LAPIS SEMICONDUCTOR

CCU8
User Manual

Program Development Support Software

ISSUE DATE: Aug. 2013

NOTICE

No copying or reproduction of this document, in part or in whole, is permitted without the consent of LAPIS Semiconductor Co., Ltd.

The content specified herein is subject to change for improvement without notice.

The content specified herein is for the purpose of introducing LAPIS Semiconductor's products (hereinafter "Products"). If you wish to use any such Product, please be sure to refer to the specifications, which can be obtained from LAPIS Semiconductor upon request.

Examples of application circuits, circuit constants and any other information contained herein illustrate the standard usage and operations of the Products. The peripheral conditions must be taken into account when designing circuits for mass production.

Great care was taken in ensuring the accuracy of the information specified in this document. However, should you incur any damage arising from any inaccuracy or misprint of such information, LAPIS Semiconductor shall bear no responsibility for such damage.

The technical information specified herein is intended only to show the typical functions of and examples of application circuits for the Products. LAPIS Semiconductor does not grant you, explicitly or implicitly, any license to use or exercise intellectual property or other rights held by LAPIS Semiconductor and other parties. LAPIS Semiconductor shall bear no responsibility whatsoever for any dispute arising from the use of such technical information.

The Products specified in this document are intended to be used with general-use electronic equipment or devices (such as audio visual equipment, office-automation equipment, communication devices, electronic appliances and amusement devices).

The Products specified in this document are not designed to be radiation tolerant.

While LAPIS Semiconductor always makes efforts to enhance the quality and reliability of its Products, a Product may fail or malfunction for a variety of reasons.

Please be sure to implement in your equipment using the Products safety measures to guard against the possibility of physical injury, fire or any other damage caused in the event of the failure of any Product, such as derating, redundancy, fire control and fail-safe designs. LAPIS Semiconductor shall bear no responsibility whatsoever for your use of any Product outside of the prescribed scope or not in accordance with the instruction manual.

The Products are not designed or manufactured to be used with any equipment, device or system which requires an extremely high level of reliability the failure or malfunction of which may result in a direct threat to human life or create a risk of human injury (such as a medical instrument, transportation equipment, aerospace machinery, nuclear-reactor controller, fuel-controller or other safety device). LAPIS Semiconductor shall bear no responsibility in any way for use of any of the Products for the above special purposes. If a Product is intended to be used for any such special purpose, please contact a ROHM sales representative before purchasing.

If you intend to export or ship overseas any Product or technology specified herein that may be controlled under the Foreign Exchange and the Foreign Trade Law, you will be required to obtain a license or permit under the Law.

Windows is a registered trademark of Microsoft Corporation (USA) in USA and other countries, and other product names and company names are trademarks or registered trademarks.

TABLE OF CONTENTS

1. OVERVIEW

CCU8 is an optimizing ‘C’ Compiler. It incorporates the features that are fundamental to the ‘C’ language and that exist in most ‘C’ compilers.

CCU8 compiles one or more input ‘C’ source files and outputs an assembly file for each of the input file. The input source file is a text file containing a standard ‘C’ source program. The output file is a text file containing relocatable assembly code.

Salient features of CCU8 are listed below:

1. ‘C’ language supported by CCU8 is implemented according to ANSI standard. Variations from the standard are imposed due to architectural constraints.
2. Variety of command line options are provided.
3. Facilities to write interrupt handling routines are available.
4. Set of pragmas are provided to utilize the architectural features.
5. Emulation libraries are provided to support **long**, **float** and **double** types.
6. Support for inline assembly.

2. OPERATING ENVIRONMENT

2.1 HARDWARE

MACHINES : IBM-PC/ AT/ Pentium compatible and clones

OPERATING SYSTEM : Windows XP, Windows Vista or Windows 7

2.2 SYSTEM CONFIGURATION

CCU8 requires the following information to be included in the CONFIG.SYS file.

files=30

This information must be added to the CONFIG.SYS file before invoking CCU8.

2.3 ENVIRONMENT VARIABLES

CCU8 uses two environment variables - INCLU8 and TMP.

INCLU8 can be used to specify directories to search the include files specified with #include preprocessor directive.

The INCLU8 environment variable can be defined using the DOS command SET. The SET command has the following format:

SET INCLU8=path

CCU8 uses temporary files during the process of compilation. The path for these temporary files can be specified in the TMP environment variable. The following line may be included in the file **autoexec.bat** that enables **CCU8** to create temporary files during compilation in the given **path**.

SET TMP=PATH

Example :

SET TMP=C:\RAMDRIVE

CCU8 uses ‘C:\RAMDRIVE’ as the path for its temporary files

If the environment variable TMP is not specified, compiler creates temporary files in the current directory.

2.4 INSTALLATION

CCU8 ‘C’ compiler comprises of the following executables:

- CCU8
- CC1U8
- CC2U8

CCU8 is a compiler loader and executes **CC1U8** and **CC2U8**. All the three executables should be present in the same directory.

3. INVOKING CCU8 AND COMMAND LINE OPTIONS

3.1 INVOCATION OF CCU8

CCU8 may be invoked by specifying the following command line:

C:\> CCU8 <CR>

C:\> CCU8 @[path] response_file <CR>

C:\> CCU8 /T string [options....] [path]file [file ....] <CR>

Note : Path of CCU8 executable should be included in PATH environment variable if CCU8 is invoked from a different path.

CCU8 shall perform a check to find version conflict between CCU8 executable and CC1U8/CC2U8 executables. If the product version number of CCU8 executable is different from product version number of CC1U8 executable or CC2U8 executable, a fatal error message is issued.

Each filename is an input 'C' source file and the name should have either “.C”, “.c”, “.H” or “.h” extension. If CCU8 encounters any other extension a fatal error message is issued and the compilation process is terminated. The *file* may have pathname.

CCU8 creates an assembly file as output for each of the files specified in the command line. The output file contains U8 assembly mnemonics.

By default, the output file has the same name as the input file with an extension “.asm”. The output file is created in the current working directory, by default.

Following are the command line options:

/T	specify the operand string for TYPE instruction
/MS	small memory model (default)
/ML	large memory model
/near	near data access specifier (default)
/far	far data access specifier
/nofar	restrict FAR access
/Ot	optimize for speed
/Ol	enable loop optimizations (default)
/Om	enable maximum optimizations
/Og	enable global optimizations (default)
/Od	disable optimizations
/Oa	enable alias checks
/Orp	enable optimization by register push/ pop routine
/Orpn	disable optimization by register push/ pop routine
/LE	generate list file
/Fa	assembly listing file
/CT	list calltree in a file
/Zg	generate function prototypes
/LP	preprocessed output in a file
/I	include file directory
/PC	preprocessed output with comment
/D	define macro
/U	undefine macro
/ST	generate stack probe routine
/SS	change stack size
/SD	generate debug information
/SL	change maximum identifier length
/J	default char type is unsigned
/PF	use comma as delimiter for pragma arguments
/SYS	change compiler segment naming strategy
/W	set warning level
/Wc	change the specified warning(s) to error(s)
/Wa	change all warnings to errors
/Za	disable extensions
/NOWIN	do not use ROM WINDOW area
/Ff	use fast emulation library
/KJ	support SHIFT-JIS Kanji characters in strings
/Lv	register/stack information for local variables
/Zs	local variable on stack

/Zp	Special packing for structure/union
/V	version information
@	response file

Command line options are explained in more detail in section 3.2.

On invocation, the following copyright message is displayed:

```
CCU8 C Compiler, Ver.3.20
Copyright  2008 - 2011 LAPIS Semiconductor Co., Ltd.
```

For the command line,

```
C:\> CCU8 <CR>
```

the following usage is displayed:

```
CCU8 C Compiler, Ver.3.20
Copyright  2008 - 2011 LAPIS Semiconductor Co., Ltd.

Usage:  CCU8 @[path]response_file
        CCU8 /T string [Options...] [path]filename...
        /T string Specify the operand string for TYPE instruction

                        -MEMORY MODEL-
/MS small model                 /near near data access specifier
/ML large model                 /far far data access specifier
/nofar restrict FAR access

                        -OPTIMIZATION-
/Ot optimize for speed          /Ol enable loop optimizations
/Om enable maximum optimizations    /Og enable global optimizations
/Od disable optimizations       /Oa Enables alias checks
/Orp - enable optimization by register push/ pop routine
/Orpn - disable optimization by register push/ pop routine

                        -OUTPUT FILES-
/LE generate list file          /Fa[filename] assembly listing file
/CT <filename> list calltree in a file  /Zg generate function prototypes

                        -PREPROCESSOR-
/LP preprocessed output in a file   /I <directory> include file directory
/PC preprocessed output with comments   /D <identifier>[=[string]] define macro
/U <identifier> undefine macro
(Press <return> to continue)
                        -STACK-
/ST generate stack probe routine    /SS <constant> change stack size
```

-DEBUG-
/SD generate debug information

-MISCELLANEOUS-
/J default char type is unsigned
/PF use comma as delimiter for pragma arguments
/SL<constant> change maximum identifier length
/SYS change compiler segment naming strategy
/W <constant> set warning level
/Wc <warning number> [,warning number,…]
change the specified warning(s) to error(s)
/Wa change all warnings to errors
/NOWIN do not use ROM WINDOW area
/Ff use fast emulation library
/Za disable extensions
/KJ support SHIFT-JIS Kanji characters in strings
/Lv register/stack information for local variables/arguments
/Zs local variable on stack
/Zp Special packing for structure/union
/V version information
@ <filename> response file

3.2 COMMAND LINE OPTIONS

This section describes various options that may be specified in the command line. All command line options are case sensitive. Options /I, /Fa, /D, /U, /Wc, /Wa and /W may be specified more than once in the command line. If any option other than /I, /Fa, /D, /U, /Wc, /Wa or /W is specified more than once in the command line, CCU8 issues fatal error message. Options /Fa, /D, /U, /Wc, /Wa and /W may also be specified between source files in the command line.

3.2.1 Machine Model Options

This section describes the machine model option /T.

3.2.1.1 TYPE STRING

Syntax : /T string

Any string may be specified with /T option. The string is not validated by CCU8. CCU8 outputs the specified string in the assembly file using TYPE pseudo instruction. This parameter is compulsory unless one of the preprocessor options /LP or /PC is specified.

Example 3.1

```
C:\> CCU8 /Tmu8 test.c  <CR>
```

/Tmu8 in the above example, instructs CCU8 to output TYPE pseudo instruction as follows:

```
type (mu8)
```

3.2.2 Memory Model Options

CCU8 supports the following memory models:

1. Small memory model
2. Large memory model

Memory model can be specified by the corresponding command line options. One of these options may be specified in the command line. If more than one option is specified, CCU8 issues a fatal error message. The memory model options are described in detail in this section.

3.2.2.1 /MS OPTION

Syntax : /MS

The **/MS** option compiles programs in small memory model. The small memory model uses one physical code segment for code and 256 physical segments for data (table or otherwise). This option is the default memory model option. If no memory model option is specified in the command line, programs are compiled in this model.

Example 3.2

```
C:\> CCU8 /Tmu8 /MS test.c <CR>
```

The command line option /MS in the above example, instructs CCU8 to compile the source file “test.c” in small memory model.

3.2.2.2 /ML OPTION

Syntax : /ML

The **/ML** option compiles programs in large memory model. The large memory model uses 16 physical code segments for code and 256 physical segments for data (table or otherwise).

Example 3.3

```
C:\> CCU8 /Tmu8 /ML test.c <CR>
```

The command line option /ML in the above example, instructs CCU8 to compile the source file “test.c” in large memory model.

Example 3.4

```
C:\> CCU8 /Tmu8 /MS /ML test.c <CR>
```

For the above command line option, CCU8 issues a fatal error because more than one memory model option is specified.

3.2.2.3 /near OPTION

Syntax : /near

/near option instructs CCU8 to treat all the data (table or otherwise) as near data, if no specifier is provided for it. This option is the default data access option. If no data access option is specified in the command line, programs are compiled in this data access option.

Example 3.5

```
C:\> CCU8 /Tmu8 /near test.c <CR>
```

The command line option /near in the above example, instructs CCU8 to treat all the data (table or otherwise) defined in the source file “test.c” as near data.

3.2.2.4 /far OPTION

Syntax : /far

/far option instructs CCU8 to treat all the data (table or otherwise) as far data, if no specifier is provided for it.

Example 3.6

```
C:\> CCU8 /Tmu8 /far test.c <CR>
```

The command line option /far in the above example, instructs CCU8 to treat all the data (table or otherwise) defined in the source file “test.c” as far data.

Example 3.7

```
C:\> CCU8 /Tmu8 /near /far test.c <CR>
```

For the above command line option, CCU8 issues a fatal error because more than one data access option is specified.

3.2.3 Optimization Options

The optimizing capabilities available with CCU8 can reduce the target storage space and/or target execution time. This is achieved by eliminating unnecessary instructions and rearranging the code.

The various options for controlling optimizations are shown in the following table:

TABLE 3.1	
OPTION	**OPTIMIZING PROCEDURE**
/Od	Disables optimization
/Ol	Enables loop optimization
/Og	Enables global optimization
/Oa	Enables alias checks
/Om	Maximizes optimization
/Ot	Speed optimization

The following optimizations are always performed by examining only short sections of the input program. These optimizations cannot be suppressed by specifying /Od option.

1. Constant propagation
2. Constant folding
3. Common subexpression elimination
4. Use of basic algebraic identities
5. Algebraic transformation
6. Peephole optimizations.

The following optimizations will be performed always unless suppressed by /Od option.

TABLE 3.2
1. Eliminating dead code
2. Eliminating dead variables
3. Optimizing jumps
4. Optimization using algebraic identities

Other options have no control over these optimizations.

By default the following optimizations are carried out when command line option is not specified:

1. Loop optimizations
2. Global constant folding
3. Global common sub expression elimination
4. Optimizations mentioned in the table 3.2.

3.2.3.1 /Od OPTION

Syntax : /Od

/Od option instructs the compiler not to perform any optimization. This option may be useful when a source program is compiled for debugging. Some of the optimizations will still be performed.

This option increases the size of the generated code and executable time.

Other optimization options cannot be specified with this option. If specified, a fatal error message indicating the illegal combination of optimization options is issued.

Example 3.8

```
C:\> CCU8 /Od /Tmu8 test.c <CR>
```

For the above command line, output file “test.asm” with unoptimized code is created.

Example 3.9

```
C:\> CCU8 /Od /Ol /Tmu8 test.c <CR>
```

A fatal error “Illegal combination of optimization options” is issued for the above command line.

3.2.3.2 /Og OPTION

Syntax : /Og

When /Og option is specified, CCU8 performs only the global optimizations. The global optimizations performed by CCU8 are:

1. Global constant propagation
2. Global constant folding
3. Global common sub expression elimination
4. Code sinking
5. Code hoisting.

/Og option enables CCU8 to perform constant propagation, constant folding and common sub-expression elimination, code sinking and hoisting by examining an entire function.

Example 3.10

```
C:\> CCU8 /Og /Tmu8 test.c <CR>
```

Loop optimizations and alias checks are not performed for the above command line. However, other optimizations shown in table 3.2 are performed.

3.2.3.3 /Ol OPTION

Syntax : /Ol

When /Ol option is specified, CCU8 performs only those optimizations that involve loops.

Following loop optimizations are performed:

1. Loop invariant code motion
2. Loop variant code motion
3. Strength reduction in loops
4. Induction variable elimination
5. Loop unrolling.

Example 3.11

```
C:\> CCU8 /Ol /Tmu8 test.c <CR>
```

/Ol option enables CCU8 to perform loop optimizations. Global optimizations and alias checks are not performed for the above command line. However, other optimizations shown in table 3.2 are performed.

Example 3.12

```
C:\> CCU8 /Ol /Og /Tmu8 test.c <CR>
```

/Ol option enables CCU8 to perform loop optimizations and /Og enables global optimizations. Alias checking is not performed for the above command line. Other optimizations shown in table 3.2 are performed.

3.2.3.4 /Oa OPTION

Syntax : /Oa

/Oa option enables the compiler to perform alias checks, which result in safe optimization.

Aliases are multiple names (that is, symbolic references) for the same memory location in a program. When /Oa option is specified, CCU8 detects and maintains the information about aliases. It then uses this information, during optimizations.

If /Oa option is not specified, the size of the output may be reduced or the speed of the output may be increased. However, the user is highly recommended to use the /Oa option to get a safe output. The user may ignore this option, only when, aliases are not used in the program.

Example 3.13

```
C:\> CCU8 /Oa /Tmu8 test.c <CR>
```

/Oa option enables CCU8 to perform alias checks. Loop optimizations and global optimizations are not performed for the above command line. Optimizations shown in table 3.2 are performed.

3.2.3.5 /Om OPTION

Syntax : /Om

/Om option enables CCU8 to perform maximum possible optimizations. When /Om option is specified, CCU8 performs all the optimizations iteratively until no more optimization can be performed. When /Om is specified, /Og and /Ol options are redundant. Global optimizations and loop optimizations will be carried out, when /Om is specified.

Example 3.14

C:\> CCU8 /Om /Tmu8 test.c <CR>

/Om option enables CCU8 to perform all the optimizations iteratively. Alias checks are not performed.

Example 3.15

C:\> CCU8 /Om /Og /Tmu8 test.c <CR>

/Om option enables CCU8 to perform all optimizations iteratively. /Og option in the above command line is redundant since global optimizations are also performed because of /Om option.

3.2.3.6 /Ot OPTION

Syntax : /Ot

/Ot option enables CCU8 to perform optimization for speed. This also enables CCU8 to perform global optimizations and loop optimizations. By default, alias checks are not performed. Sometimes, the speed optimization increases the output code size. The options /Om, /Ot and /Od are mutually exclusive.

Example 3.16

C:\> CCU8 /Ot /Tmu8 test.c <CR>

/Ot option enables CCU8 to perform optimization for speed.

3.2.3.7 /Orp OPTION

Syntax : /Orp

/Orp enables CCU8 to perform optimization by register push/ pop routine. When /Orp option is specified in command line, CCU8 generates the code which calls subroutines for saving/restoring the registers.

When /Om option is specified, CCU8 treats as /Orp option is specified automatically. /Orp option does not become valid unless /Orp option is specified expressly, when /Om option is not specified. When /Od option is specified, /Orp option cannot be specified. When both /Od option and /Orp option is specified, CCU8 issues an error.

/Orp and /Orpn are exclusive options. When both /Orp and /Orpn are specified in command line, CCU8 issues an error.

In other than /Om and a /Od option, it becomes valid only when /Orp option is specified expressly.

[Note]

Code size can be reduced when /Orp option is specified.

However, saving and restoring of the register which is not used within a function are also performed in this case.

For this reason, a stack consumption and the number of processing cycles of the entrance and exit for a function will increase.

3.2.3.8 /Orpn OPTION

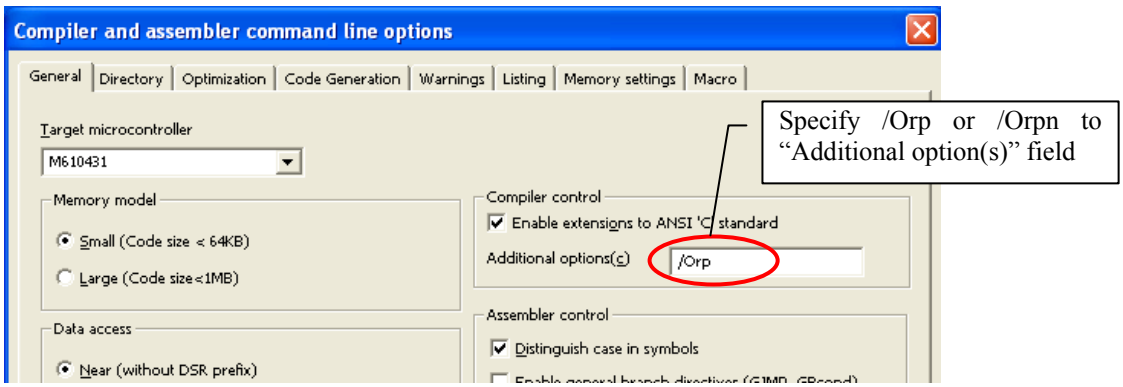
Syntax : /Orpn

/Orpn disnables CCU8 to perform optimization by register push/ pop routine. When /Orpn option is specified in command line, CCU8 does not generate the code which calls subroutines for saving/restoring the registers (It becomes the same output as CCU8 V3.31 or before).

/Orp and /Orpn are exclusive options. When both /Orp and /Orpn are specified in command line, CCU8 issues an error.

[Note]

When you specify /Orp or /Orpn by IDEU8, please specify /Orp or /Orpn to “Additional option(s)” field of “Compiler control” of “Compiler and assemble command line options” dialog.



3.2.3.9 SUMMARY OF OPTIMIZATION OPTIONS

The actions performed when the above optimization options are specified is summarized in the following table:

TABLE 3.3

Optimization Options		**Loop Optimizations**	**Global Optimizations**
Default	no /Oa	Performed – unsafe	Performed – unsafe (except hoisting/sinking)
	/Oa	Performed – safe	Performed – safe (except hoisting/sinking)
/Od	no /Oa	Not performed	Not performed
	/Oa	Error	Error
/Ol	no /Oa	Performed - unsafe	Not performed
	/Oa	Performed - safe	Not performed
/Og	no /Oa	Not performed	Performed - unsafe
	/Oa	Not performed	Performed - safe
/Om	no /Oa	Performed - unsafe	Performed - unsafe
	/Oa	Performed - safe	Performed - safe
/Ot	no /Oa	Performed - unsafe	Performed - unsafe
	/Oa	Performed - safe	Performed - safe

A combination of /Ol, /Og and /Oa optimization options may be specified.

It shows the relation of the optimization option and optimization items.

Optimization item \ Optimization option			/Od	/Ol	/Og	/Om	/Ot	nothing (default)
Local optimization			ON	ON	ON	ON	ON	ON
Peephole optimization			ON	ON	ON	ON	ON	ON
Loop optimization			-	ON	-	ON	ON	ON
Global optimization			-	-	ON	ON	ON	ON
	Code sinking Code hoisting		-	-	ON	ON	ON	-
Speed optimization			-	-	-	-	ON	-
Other optimization			-	ON	ON	ON	ON	ON
Optimization of register push/pop		**/Orp** *2	- *1	ON	ON	ON	ON	ON
		nothing	-	-	-	ON	-	-
Suppress function	**Suppress optimization of register push/pop**	**/Orpn** *2	- *1	ON *3	ON *3	ON *3	ON *3	ON *3
		nothing	-	-	-	-	-	-
	Suppress optimization by alias checking	/Oa	- *1	ON	ON	ON	ON	ON
		nothing	ON *4	-	-	-	-	-
	Suppress register allocation for local variable	/Zs	ON *4	ON	ON	ON	ON	ON
		nothing	ON *4	-	-	-	-	-

*1 : The gray field shows combination is improper. (CCU8 issues an error.)

*2 : /Orp and /Orpn cannot be specified concurrently. (CCU8 issues an error.)

*3 : When /Ol, /Og, /Ot, or nothing(default) is specified, optimization of register push/pop does not perform even if /Orpn is not specified. When /Om is only specified, /Orpn has effect.

*4 : When /Od is specified, CCU8 treats as /Oa and /Zs are specified automatically.

3.2.3.10 COMBINATION OF OPTIMIZATION OPTIONS

The following table summarizes the combination of the optimization options:

TABLE 3.4

No.	Combinations	Validity	Optimizations Performed If The Combination Is Valid
1.	/Od /Og	Invalid	-
2.	/Od /Ol	Invalid	-
3.	/Od /Oa	Invalid	-
4.	/Od /Om	Invalid	-
5.	/Od /Ot	Invalid	-
6.	/Om /Ot	Invalid	-
7.	/Og /Ol	Valid	Loop Optimizations Global Optimizations Other Optimizations
8.	/Og /Oa	Valid	Global Optimizations Alias Checking Other Optimizations
9.	/Og /Om	Valid	Loop Optimizations Global Optimizations Other Optimizations
10.	/Og /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Other Optimizations
11.	/Ol /Oa	Valid	Loop Optimizations Alias Checking Other Optimizations
12.	/Ol /Om	Valid	Loop Optimizations Global Optimizations Other Optimizations
13.	/Ol /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Other Optimizations
14.	/Oa /Om	Valid	Loop Optimizations Global Optimizations Alias Checking Other Optimizations

No.	Combinations	Validity	Optimizations Performed If The Combination Is Valid
15.	/Oa /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Alias Checking Other Optimizations
16.	/Og /Ol /Oa	Valid	Loop Optimizations Global Optimizations Alias Checking Other Optimizations
17.	/Og /Ol /Om	Valid	Loop Optimizations Global Optimizations Other Optimizations
18.	/Ol /Oa /Om	Valid	Loop Optimizations Global Optimizations Alias checking Other Optimization
19.	/Ol /Og /Oa /Om	Valid	Loop Optimizations Global Optimizations Alias checking Other Optimizations
20.	/Og /Ol /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Other Optimizations
21.	/Ol /Oa /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Alias checking Other Optimization
22.	/Ol /Og /Oa /Ot	Valid	Loop Optimizations Global Optimizations Speed Optimization Alias checking Other Optimizations

3.2.4 Output Files

3.2.4.1 ERROR LISTING OPTION

Syntax : /LE

/LE option enables CCU8 to generate listing of source files along with error messages, if any. The complete source code is listed with line numbers. The name of the listing file is the same as input file with an extension ".LER".

This option cannot be specified along with any of the preprocessor options /LP or /PC. If specified, fatal error is issued.

Information about the size of stack used in each function is also output in the list file, if no error messages or fatal error messages are generated.

Example 3.17

```
C:\> CCU8 /LE  /Tmu8 test.c <CR>
```

CCU8 generates a listing file "test.ler" for the above command line. The output listing file contains the source with line numbers and the generated errors, if any.

3.2.4.2 CALLTREE OPTION

Syntax : /CT filename

This option enables CCU8 to generate a listing of function calls. Call tree listing file contains an indented listing showing function names at the left margin and calls in each function.

Filename must be specified along with the option /CT. If file name is not specified, CCU8 issues an error message. No default extension is assumed by CCU8.

This option cannot be specified along with any of the preprocessor options /LP or /PC. If specified, fatal error is issued.

Example 3.18

```
C:\> CCU8 /CT test.cal /Tmu8 test.c <CR>
```

/CT option in the above command line enables CCU8 to generate a call tree listing file "test.cal". An indented listing of function names and the calls in each function is output in "test.cal".

Example 3.19

```
C:\> CCU8 /CT test.cal /Tmu8 t1.c t2.c <CR>
```

/CT option in the above command line enables CCU8 to generate a calltree listing file "test.cal". Calltree listing of both the files t1.c and t2.c is output in "test.cal" one after the other. However, the function information is not carried from one file to another.

Example 3.20

```
C:\> CCU8 /CT test.cal /LP test.c <CR>
```

A fatal error is issued by CCU8 for the above command line, since /CT and /LP options are mutually exclusive.

3.2.4.3 ASSEMBLY LISTING FILE

Syntax : /Fa[path]

/Fa option enables CCU8 to generate assembly listing file in the specified name, path or a directory. If a file name with or without path is specified with /Fa option then assembly listing will be output in that file. If the specified file name has no extension then the output file will have ".ASM" extension.

If a directory is alone specified with /Fa option, then the assembly listing will be created in that specified directory with the default file name.

The argument 'path' is optional. If no argument is specified with /Fa option, then the assembly listing will be created in the current directory with the default file name.

If the file or path specified with /Fa option is invalid, CCU8 issues fatal error message.

If a directory is specified with /Fa option then that will be considered for all the source files following that /Fa option if no other /Fa option is specified in between.

/Fa option can be specified any number of times in the command line for a source file. If more than one /Fa option is specified before a source file then only the latest /Fa option will be considered for that source file.

/Fa option may also be specified between source files in the command line.

Example 3.21

```
C:\> CCU8 /Fa..\ /Tmu8 test.c <CR>
```

For the above command line, assembly list file "test.asm" will be created in the parent directory of the current working directory.

Example 3.22

```
C:\> CCU8 /Fad:\asm\ /Tmu8 test1.c test2.c<CR>
```

For the above command line, the assembly listing output files "test1.asm" and "test2.asm" will be created in "d:\asm" directory.

Example 3.23

```
C:\> CCU8 /Faout1 /Faout2 /Tmu8 test.c <CR>
```

For the above command line, the assembly listing will be in the name “out2.asm”.

3.2.4.4 FUNCTION PROTOTYPE OPTION

Syntax : /Zg

/Zg option enables CCU8 to generate listing of function prototypes. The name of the listing file is same as the input file with an extension “.PRO”.

This option cannot be specified along with one of the preprocessor options /LP or /PC. If specified, fatal error is issued.

Example 3.24

```
C:\> CCU8 /Zg /Ot test.c <CR>
```

CCU8 generates a listing file “test.pro” for the above command line. The output listing file contains the prototypes for body defined functions.

3.2.5 Preprocessor Options

3.2.5.1 /LP OPTION

Syntax : /LP

This option enables CCU8 to generate listings of preprocessed output of each of the input files. When this option is specified CCU8 acts as a text processor that manipulates the text of the source files. It performs the following functions :

1. Macro expansion
2. Comment removal
3. File inclusion
4. Conditional compilation
5. Line control
6. Error generation

by processing the preprocessor directives inside the source files.

The name of the preprocessed file is same as the input file with an extension ".I". When this option is specified, source files are not compiled.

List file option (/LE), calltree option (/CT) and function prototype listing (/Zg) cannot be specified with /LP option. The input filename may have any extension (including empty extension) except ".i", when this option is specified.

Example 3.25

```
C:\> CCU8 /LP test.c <CR>
```

/LP in the above example instructs the compiler to create a preprocessed output file test.i. Comments will be stripped.

Example 3.26

```
C:\> CCU8 /LP /LE test.c <CR>
```

A fatal error is generated for the above command line, since /LE option and /LP option are mutually exclusive.

3.2.5.2 /PC OPTION

Syntax : /PC

The preprocessor, while preprocessing, normally removes all comments present in the source file. /PC option instructs the compiler to preserve comments during preprocessing. CCU8 produces a preprocessed output listing with the comments specified in the source file. All other functions are similar to that of /LP option.

The preprocessor options /PC and /LP are mutually exclusive. Only one of these options may be specified in the command line. When both these options are specified together, CCU8 issues a fatal error "Duplicate preprocessor option".

The name of the preprocessed file is the same as input file but with an extension ".i". When this option is specified, source files are not compiled.

List file option (/LE), calltree option (/CT) and function prototype listing (/Zg) cannot be specified with /PC option. The input filename may have any extension (including empty extension) except ".i", when this option is specified.

Example 3.27

```
C:\> CCU8 /PC test10.inp <CR>
```

/PC in the above example instructs the compiler to create a preprocessed output file test10.i. Comments are preserved in the output file.

Example 3.28

```
C:\> CCU8 /PC /LP key.c <CR>
```

A fatal error message is generated for the above command line, since the options /LP and /PC are mutually exclusive.

3.2.5.3 /I OPTION

Syntax : /I <directory>

A directory to search the included files can be given with /I option. This option temporarily overrides or changes the effect of environment variable INCLU8. CCU8 searches the directory specified in this option first, before searching the standard places given in INCLU8 environment variable.

Only one directory name can be specified with an /I option. If more than one directory names are to be specified, /I option should be used repeatedly.

Example 3.29

```
C:\> CCU8 /I \include /Tmu8 test.c <CR>
```

The above command line instructs CCU8 to search the included files in the directory "\include" before searching in the directories specified using the environment variable INCLU8.

Example 3.30

```
C:\> CCU8 /I include /I lib /LP test.c <CR>
```

For the above command line, CCU8 searches the included file in the directory "include" first. And if not found, it searches in the directory "lib". If still not found, CCU8 searches in the directories specified by the environment variable INCLU8.

3.2.5.4 /D OPTION

Syntax : /D <identifier>[=[string]]

where 'identifier' is the macro and 'string' is the replacement text.

A macro without argument can be defined in the command line using /D option. The macro processing is same as if it is specified in the source file.

If the argument specified with /D option is not an identifier then CCU8 issues fatal error message.

Example 3.31

```
C:\> CCU8  /Tmu8 /DVALUE(a) test14.c <CR>
```

For the above command line, CCU8 issues fatal error message since ‘VALUE(a)’ is not an identifier.

Whitespaces may or may not be specified in between ‘/D’ and the macro. If only identifier is specified with /D option then the replacement text for the macro is ‘1’.

Whitespaces cannot be specified between the identifier and ‘=’. If the argument ends with ‘=’ then the replacement text for the macro is empty.

Whitespaces cannot be specified between ‘=’ and the replacement string.

/D option can be specified before each source file name in the command line. The macro defined with /D option is considered for all the files specified after that /D option in the command line.

Example 3.32

```
C:\> CCU8  /Tmu8  /DVALUE1  test15.c  /DVALUE2=  test16.c <CR>
```

The macro ‘VALUE1’ is defined as 1 and it is considered for both ‘test15.c’ and ‘test16.c’. Whereas, the macro ‘VALUE2’ is defined with no replacement text and it is considered only for the file ‘test16.c’.

3.2.5.5 /U OPTION

Syntax : /U <identifier>

A previously defined command line macro can be undefined using /U option in the command line.

An identifier must be specified along with /U option. If identifier is not specified, CCU8 issues fatal error message.

Whitespaces may or may not be specified between ‘/U’ and the identifier.

/U option may be specified any number of times in the command line. /U option can be specified anywhere in the command line, before the first source file name, after the last source file name or in between two source file names.

If the argument specified with /U option is not an identifier, CCU8 issues fatal error message.

CCU8 ignores /U option if the identifier specified along with it is not previously defined using /D option.

/U option does not have any effect on predefined macros and macros defined by user in the source file.

Example 3.33

```
C:\> CCU8  /Tmu8  /DVALUE=1  test15.c test16.c /UVALUE  test17.c <CR>
```

The macro ‘VALUE’ is defined as 1 and it is considered for ‘test15.c’ and ‘test16.c’. But it is undefined after ‘test16.c’ and not considered for ‘test17.c’.

3.2.6 Stack

3.2.6.1 STACK SIZE OPTION

Syntax : /SS <constant>

/SS option sets the size of the program stack. CCU8 outputs the size specified in this option using the pseudo instruction STACKSEG. This enables the linker RLU8, at a later stage, to allocate memory for the program stack.

If this option is not specified, CCU8 sets a default stack size of 1024 bytes.

The constant must be a decimal constant. The valid range of the constant is an even value between 0 and 65535 (exclusive of both). The space between /SS and the constant is optional.

Example 3.34

```
C:\> CCU8 /SS 2048  /Tmu8 test.c <CR>
```

STACKSEG pseudo instruction with size 2048 is output by CCU8 for the above command line.

Example 3.35

```
C:\> CCU8 /SS0x0800   /Tmu8 test.c <CR>
```

A fatal error is issued by CCU8 for the above command line, since only a decimal constant is expected as a parameter for /SS option.

3.2.6.2 STACK CHECKING OPTION

Syntax : /ST

When /ST option is specified, stack probes are added in the assembly output by CCU8.

A “stack probe” is a short routine called on entry to function, to verify if there is enough room in the program stack to allocate local variables required by the function. The stack probe routine jumps to a ‘C’ function ‘_stack_error’, when it determines that the required size is not available in the stack. The function ‘_stack_error’ has to be defined by the user.

When this option is not specified, stack probe routine is not called, and stack overflow may occur without being diagnosed.

Example 3.36

```
C:\> CCU8 /ST /Tmu8 test.c <CR>
```

Calls to stack probe routine are generated at the entry code of each function in “test.c” for the above command line.

3.2.7 Debugging Options

3.2.7.1 /SD OPTION

Syntax : /SD

If /SD option is specified, CCU8 generates the necessary information for the ‘C’ source level debugger. Files compiled without /SD option cannot be debugged using the debugger at source level.

Debugging information are embedded in the assembly output as text.

Example 3.37

```
C:\> CCU8 /SD /Tmu8 test.c <CR>
```

For the above command line, the debug information is embedded in the assembly output as text.

3.2.8 Miscellaneous Options

3.2.8.1 /SL OPTION

Syntax : /SL <constant>

/SL option sets the maximum length of an identifier. The constant must be an integer in the range 31 to 254, inclusive of both.

If this option is not specified, CCU8 assumes the maximum length of an identifier as 31.

Example 3.38

C:\> CCU8 /SL 40 /Tmu8 test.c <CR>

In the above example, CCU8 takes maximum length of an identifier as 40 characters. If in "test.c" any identifier is encountered whose length exceeds 40 characters, only first 40 characters will be considered as identifier name and a warning message will be given.

Example 3.39

C:\> CCU8 /Tmu8 test.c <CR>

In the above example, CCU8 takes maximum length of an identifier as 31 characters (default maximum identifier length).

Example 3.40

C:\> CCU8 /SL 1023 /Tmu8 test.c <CR>

A fatal error is issued for the above command line, since the expected range of the constant value is between 31 and 254, inclusive of both.

Example 3.41

C:\> CCU8 /SL /Tmu8 test.c <CR>

A fatal error is issued for the above command line, since a constant value is expected after /SL.

3.2.8.2. /J OPTION

Syntax : /J

/J option instructs CCU8 to treat default '**char**' type as '**unsigned char'** type. If /J option is specified in the command line, CCU8 treats all '**char**' type without '**signed'** specifier as '**unsigned char'** type.

Example 3.42

```
char chr ;
```

By default, CCU8 treats the variable ‘chr’ as ‘**signed char’** type. If /J option is specified in the command line, CCU8 treats the variable ‘chr’ as ‘**unsigned char’** type.

3.2.8.3 /PF OPTION

Syntax : /PF

The default pragma argument delimiter is whitespace. This can be changed to “,” (comma) by specifying /PF option in the command line.

The following is the pragma syntax, when /PF option is not specified:

```
#pragma pragma_keyword  [ argument1  argument2  ...]
```

If /PF option is specified in the command line, the pragma syntax is as follows:

```
#pragma pragma_keyword  [ argument1,  argument2,  ...]
```

Example 3.43

```
#pragma interrupt  function_name, address
```

The above pragma syntax is valid if /PF is specified in the command line. Otherwise, CCU8 issues a warning message and ignores pragma.

3.2.8.4 /SYS OPTION

Syntax : /SYS

/SYS option directs the compiler to change the segment naming strategy. This option may be used during compiling system files.

Example 3.44

```
C:\> CCU8 /SYS /Tmu8 test.c <CR>
```

In the above example, CCU8 uses a different segment naming strategy while compiling “test.c”.

3.2.8.5 /W OPTION

Syntax : /W <constant>

/W option directs the compiler to issue warnings according to the level specified. The constant must be a decimal constant between 0 and 3, inclusive of both. /W option can be specified with any command line option.

If arguments other than 0, 1, 2, 3 is specified in /W option, fatal error message is issued.

If no arguments are specified along with /W option, fatal error message is issued.

Whitespaces may or may not be specified between /W and its argument.

/W option can be specified any where in the command line, before the first source file name, after the last source file name or in between two source file names. If /W option is not specified, 1 is assumed is as default level.

Example 3.45

```
C:\> CCU8 /Tmu8 /W3  test.c <CR>
```

In the above example, CCU8 issues warnings occurred upto level 3 while compiling “test.c”.

Example 3.46

```
C:\> CCU8  /Tmu8  /W0  test20.c  /W3  test16.c <CR>
```

In the above example, CCU8 issues no warnings occurred while compiling “test20.c” and warnings occurred upto level 3 while compiling “test16.c”.

3.2.8.6 /Wc OPTION

Syntax : /Wc <warning number> [,warning number,…]

/Wc option directs the compiler to issue error(s) instead warning(s) irrespective of the warning level. The warning(s) are specified by the warning number in the command line. /Wc option must be followed by atleast one warning number. More than one warning number can be specified by using the ‘,’ separator. /Wc option can be specified with any command line option. If /Wc option specified along with /Wa option, then /Wc option does not have any effect and ignored.

If no warning number(s) is specified, fatal error message is issued.

If no ‘,’ separator is found in between the warning numbers, fatal error message is issued.

If the specified warning number(s) is invalid, fatal error message is issued.

White spaces may or may not be specified between /Wc and its argument.

/Wc option can be specified any where in the command line, before the first source file name, or in between two source file names.

Example 3.47

```
C:\> CCU8 /Tmu8 /Wc W5017 test.c <CR>
```

In the above example, CCU8 issues W5017 warning as error if this warning occured while compiling “test.c”.

Example 3.48

```
C:\> CCU8  /Tmu8  /W0   /Wc W5025, W5033,W6000  test16.c <CR>
```

In the above example, CCU8 issues errors for the warnings W5025,W5033,W6000 if these warnings occur, while compiling “test16.c”. If any other warning occurs, while compiling “test16.c”, no warning will be issued since /W0 is specified.

Example 3.49

```
C:\> CCU8  /Tmu8   /Wc W5025, W5033, W6000  /W2 test16.c <CR>
```

In the above example, CCU8 issues errors for the warnings W5025,W5033,W6000 if these warnings occur, while compiling “test16.c”. If any other warnings upto level 2 occur, while compiling “test16.c”, warnings will be issued since /W2 is specified.

Example 3.50

```
C:\> CCU8  /Tmu8   /Wc W5025  /W2 test16.c  /W1 /Wc W6000 test20.c<CR>
```

In the above example, CCU8 issues error for the warning W5025 if this warning occurs and issues warnings occurring upto level 2, while compiling “test16.c”. Issues errors for the warnings W5025 and W6000 if these warnings occur and issues warnings occurring upto level 1, while compiling “test20.c”.

Example 3.51

```
C:\> CCU8  /Tmu8   /Wc W5025  /W2 test16.c  /Wa /W1 /Wc W6000 test20.c<CR>
```

In the above example, CCU8 issues error for the warning W5025 if this warning occurs and issues warnings occurring upto level 2, while compiling “test16.c”. /Wc option is ignored and issues errors for the warnings occurring upto level 1, while compiling “test20.c”, since /Wa /W1 option has been specified.

Note :

Although the warning has been changed to error, the number, the message and the description are to be referred from the warning list pertaining to the warning number.

3.2.8.7 /Wa OPTION

Syntax : /Wa

/Wa option directs the compiler to issue errors for all warnings occurring at the specified warning level (default level 1). This option takes precedence over /Wc option. /Wa option can be specified with any command line option.

/Wa option can be specified any where in the command line, before the first source file name, or in between two source file names.

Example 3.52

```
C:\> CCU8  /Tmu8  /Wa /W3 /Wc W6000 test20.c<CR>
```

In the above example, CCU8 issues errors for all the warnings occurring upto level 3, while compiling “test20.c”.

Note :

Although the warning has been changed to error, the number, the message and the description are to be referred from the warning list pertaining to the warning number.

3.2.8.8 /NOWIN OPTION

Syntax : /NOWIN

/NOWIN option directs the compiler not to use ROM WINDOW area.

Example 3.53

```
C:\> CCU8 /Tmu8 /NOWIN  /SD  /ML  test.c <CR>
```

In the above example, CCU8 compiles the file "test.c" without using the ROM WINDOW area.

Note :

This option is an option for a microcontroller without ROM WIDOW area.
At present, the microcontroller without ROM WINDOW area does not exist.
Therefore, please do not specify this option.

3.2.8.9 /Ff OPTION

Syntax : /Ff

/Ff option directs the compiler to use fast emulation library. This option is supported to reduce the execution time for float type.

Example 3.54

```
C:\> CCU8 /Tmu8 /Ff  /SD  /ML  test.c <CR>
```

In the above example, CCU8 compiles the file "test.c" using fast emulation library.

3.2.8.10 /Za OPTION

Syntax : /Za

/Za option directs the compiler to disable all extensions to ANSI ‘C’ standard, while compiling a source file. CCU8 expands the predefined macro ‘__STDC__’ to 0.

Example 3.55

```
C:\> CCU8 /Tmu8 /Za  /SD  /ML  test.c <CR>
```

In the above example, CCU8 issues necessary warnings and errors according to ANSI ‘C’ standard.

3.2.8.11 /KJ OPTION

Syntax : /KJ

/KJ option instructs the compiler to allow SHIFT-JIS Kanji characters (two byte) in strings.

Example 3.56

```
C:\> CCU8 /Tmu8 /KJ test.c <CR>
```

In the above example, SHIFT-JIS Kanji characters support is provided for strings in source file “test.c” .

3.2.8.12 /Lv OPTION

Syntax : /Lv

/Lv option instructs the compiler to output the register/stack information that are allocated to local variables/arguments. Each function shall be preceded by this information. This expanded information about local variables/arguments increases readability of the assembly code.

Example 3.57

```
C:\>CCU8 /Tmu8 /Lv test.c <CR>
```

In the above example, for each function, the information of register/stack allocated for each existing local variable/argument being output in the output file. Existing local variable refers to the local variable that is live after optimizations are performed.

Example 3.58

INPUT :

```
int g_i ;

void
fn (long a_l1, int a_i1, char a_c1)
{
        int l_i1 ;
        register int l_i2 ;

        l_i1 = fn1 (a_i1) ;
        l_i2 = fn1 (a_c1) ;

        g_i = l_i1 + l_i2 ;
}
```

The following is the code generated when /Lv command line option specified :

OUTPUT :

```
_fn     :

;;{
;;*****************************************
;;      register/stack information
;;*****************************************
        _a_i1$4 set     4
        _a_c1$6 set     6
;;*****************************************
```

```
                push    lr
                push    fp
                mov     fp,     sp
                push    er4

;;              l_i1 = fn1 (a_i1) ;
                l       er0,    _a_i1$4[fp]
                bl      _fn1
                mov     er4,    er0     ;; _ll_i1$0

;;              l_i2 = fn1 (a_c1) ;
                l       r0,     _a_c1$6[fp]
                extbw   er0
                bl      _fn1
                mov     er2,    er0     ;; _l_i2$2

;;              g_i = l_i1 + l_i2 ;
                mov     er0,    er4
                add     er0,    er2
                st      er0,    NEAR _g_i

;;}
                pop     er4
mov             sp,     fp
                pop     fp
                pop     pc
```

3.2.8.13 /Zs OPTION

Syntax : /Zs

/Zs option instructs the compiler not to allocate the local variables on register. Instead local variables shall be allocated to stack locations. However, register keyword for a local variable has higher priority than /Zs command line option, when a free register is available for allocation.

Example 3.59

C:\>CCU8 /Tmu8 /Zs test.c <CR>

In the above example, all local variables will be allocated to stack memory location. However, those local variables with register keyword will be allocated to stack memory only if a register is not available.

Example 3.60

INPUT :

```
int g_i ;
```

```
void
fn ()
{
        int l_i1 ;
        register int l_i2 ;

        l_i1 = fn1 () ;
        l_i2 = fn1 () ;

        g_i = l_i1 + l_i2 ;
}
```

The following is the code generated when /Zs command line option specified :

OUTPUT :

```
_fn     :


        push    lr
        push    fp
        mov     fp,     sp
        add     sp,     #-02
        push    er4

;;      l_i1 = fn1 () ;
        bl      _fn1
        st      er0,    -2[fp]

;;      l_i2 = fn1 () ;
        bl      _fn1
        mov     er4,    er0

;;      g_i = l_i1 + l_i2 ;
        l       er0,    -2[fp]
        add     er0,    er4
        st      er0,    NEAR _g_i

;;}
        pop     er4
        mov     sp,     fp
        pop     fp
        pop     pc
```

3.2.8.14 /Zp OPTION

Syntax : /Zp

The /Zp command line option shall instruct the compiler to change its structure / union member alignment based on the type of members. The alignment will be as follows:

Member Type	**Alignment**
char or unsigned char	Byte boundary
int or unsigned int or short or unsigned short or long or unsigned long	Word boundary
float or double	Word boundary
struct	If all members of the struct are char type then Byte boundary else word boundary
union	If any one of the member is not char type then word boundary else byte boundary
array	If array type is char then byte boundary, else word boundary
pointer	Word boundary

Note :

Structure size can be odd if all members are char type.

Structure size will be even if any one of the member is non char type

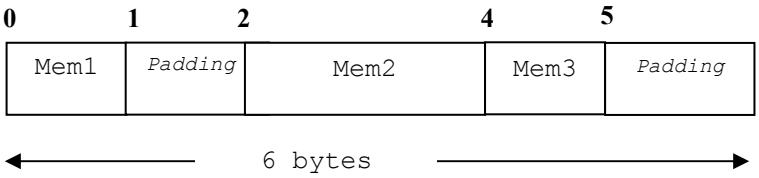
Since the copy of the structure will be performed per byte when /Zp option is specified, code size becomes large and execution speed becomes slow.

Example 3.61

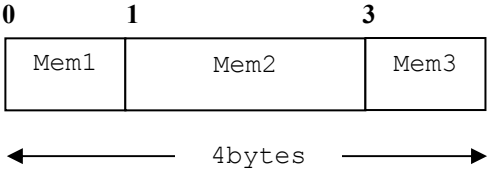
1.

```
struct ArrayMemberCharTag {

unsigned char  mem1;
unsigned char  mem2[2];
unsigned char  mem3;

} ArrayMemberChar;
```

Without Zp



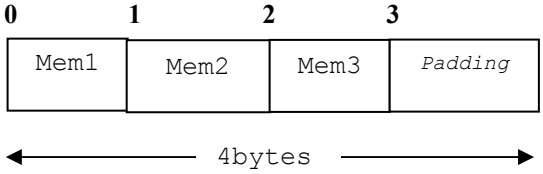
With Zp



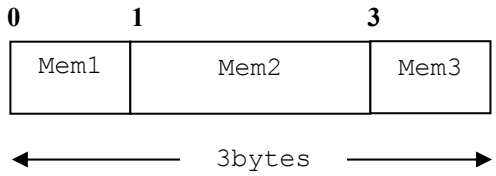
2.

```
struct MemberCharTag {

unsigned char  mem1;
unsigned char  mem2;
unsigned char  mem3;

} MemberChar;
```

Without Zp



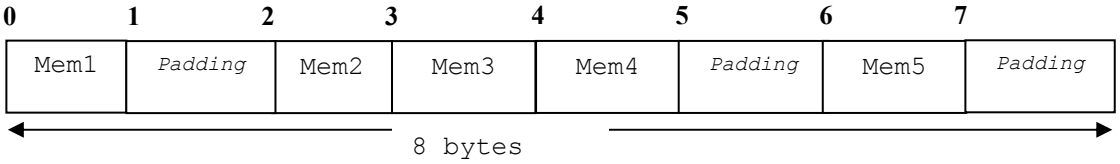
With Zp



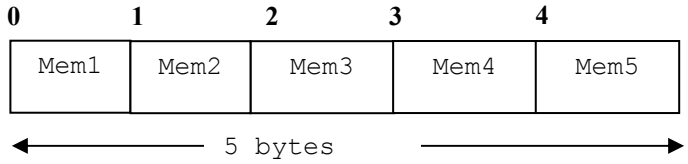
3.

```
struct ArrayMemberCharTag {

   unsigned char  mem1;
   struct ArrayMemberCharTag2 {

      unsigned char  mem2;
      unsigned char  mem3;
      unsigned char  mem4;

   }ArrayMemberChar2;

   unsigned char  mem5;
} ArrayMemberChar;
```

Without Zp



With Zp



3.2.8.15 /V OPTION

Syntax : /V

/V option shall instruct the compiler to display the file versions of CCU8, CC1U8 and CC2U8 executables. This option shall be effective only when it is specified as a first command line option. Otherwise this option shall be ignored.

Example 3.62

```
The Product Version number of CCU8 is 3.20.
The File Version of CCU8.EXE is 3.20.1
The File Version of CC1U8.EXE is 3.20.1
The File Version of CC2U8.EXE is 3.20.1

C:\>CCU8 /V  <CR>

   CCU8 C Compiler, Ver.3.20
   Copyright  2008 - 2011 LAPIS Semiconductor Co., Ltd.

      CCU8.EXE    Ver.3.20.1
      CC1U8.EXE   Ver.3.20.1
      CC2U8.EXE   Ver.3.20.1
```

3.2.8.16 @ OPTION

Syntax : @ <filename>

The @ command line option shall instruct the compiler to accept command line options from the given filename (response file). The response files are text files containing command line options. The response file can appear anywhere on the command line immediately following the @ mark. No space should be provided between '@' and the filename or the path. No default extension is assumed. The response file may also include the path.

CCU8 will search the response file in the following directories in the following order :

- Current directory
- Directories listed in the PATH environment variable
- Directory containing CCU8.

A response file should consist of lines with command line options and file names just as they would appear on the command line. These lines can also contain comments and MS-DOS environment variables.

For example, %env_var% shall be expanded to the contents of the specified environment variable. If the environment variable TMP has been set with set TMP=C:\tmp, and the response file contains the line /D TMP=%TMP%, the line will be expanded to /D TMP=C:\TMP in **CCU8**.

Each command line option can have its own line, complete with comment. Comments should start with a sharp (#), semicolon (;), or two consecutive slashes(\\).

Everything from these delimiters through to the newline shall be equivalent to a single space.

3.2.8.17 /nofar OPTION

Syntax : /nofar

/nofar option instructs CCU8 to suppresses output of DSR save and return codes within the interrupt and SWI functions when only data memory space of physical segment #0 is used. CCU8 does not permit FAR access when this option is specified in the command line.

FAR access is not allowed with /nofar option. If the variable is declared with FAR pragma CCU8 isues error as FAR access is not allowed with /nofar option. CCU8 also does not allow the pointers which are qualified by __far or declared by #pragma FAR.

The combination of /nofar and /far is not allowed.

Please refer to “5.1.2 Suppress the Use of DSR in Interrupt Function” and “Suppress the Use of DSR in SWI Function” about the example outputs when /nofar is specified.

3.2.8.18 /Zc OPTION

Syntax : /Zc

In RLU8 Ver.1.50, the feature which does not link the function/table which is not referred to has been added.

In order to enable this feature, from CCU8 Ver.3.30, the segment to which function/table belongs is separated not per file unit but per function / const variable.

By this feature, useless function and table are no longer linked, and in order that code size becomes small, it operates by default.

/Zc option is specified in order to link the function/table which is not referred to as well as the operation before V3.21.

By the existence of /Zc option, it shows below the difference of the segment name which a compiler outputs.

/Zc option	Function/Table Types	Default Segment Name
ON	Normal Function (Small model)	$$NCOD*filename*
	Normal Function (Large model)	$$FCOD*filename*
	Interrupt service routine	$$INTERRUPTCODE
	Const variable (near)	$$NTAB*filename*
	Const variable (far)	$$FTAB*filename*
OFF	Normal Function (Small model)	$$*funcname*$*filename*
	Normal Function (Large model)	$$*funcname*$*filename*
	Interrupt service routine	$$*funcname*$*filename*
	Const variable (near)	$$TAB*constname*$*filename*
	Const variable (far)	$$TAB*constname*$*filename*

funcname is a Function Name, *constname* is a Const variable Name, *filename* is a File Name.

The segments except the function/interrupt service routine/table is not affected by /Zc option.

3.2.9 Invalid Combination Of Options

The following are invalid combinations of command line options:

1. /LP and /PC
2. /LE and preprocessor options (/LP or /PC)
3. /CT and preprocessor options (/LP or /PC)
4. /Fa and preprocessor options (/LP or /PC)
5. /Zg and preprocessor options (/LP or /PC)
6. /Om and /Ot
7. /Od and other optimization options (/Ol, /Og, /Oa, /Om and /Ot)
8. /nofar and /far

4. MEMORY MODELS

This section describes about the various memory models and data access specifiers supported by CCU8 and the additional memory model qualifiers provided.

4.1 MEMORY MODELS

CCU8 supports the following memory model options:

Small memory model

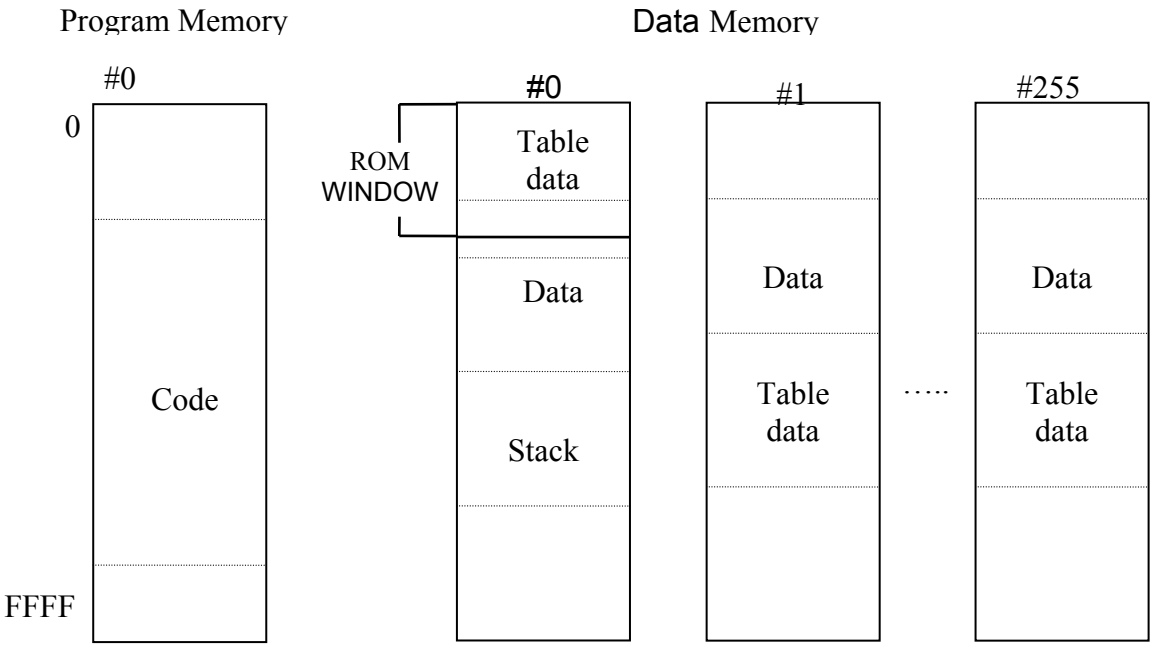
Large memory model

Command line options corresponding to the memory models are as follows:

/MS option for Small memory model

/ML option for Large memory model

4.1.1 Small Memory Model

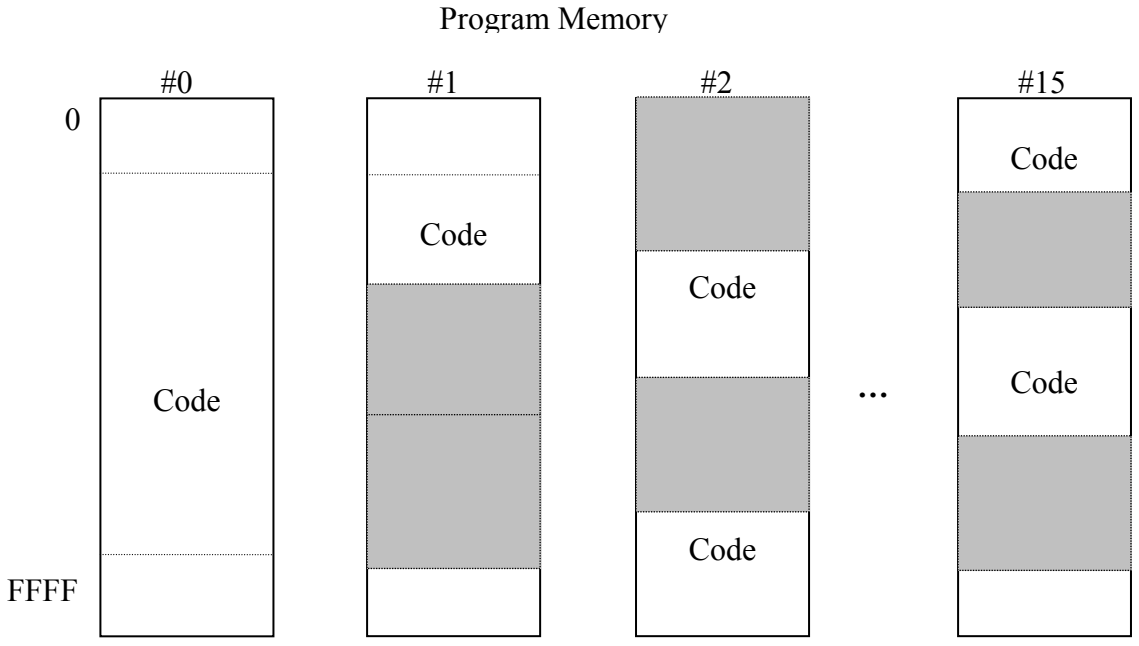


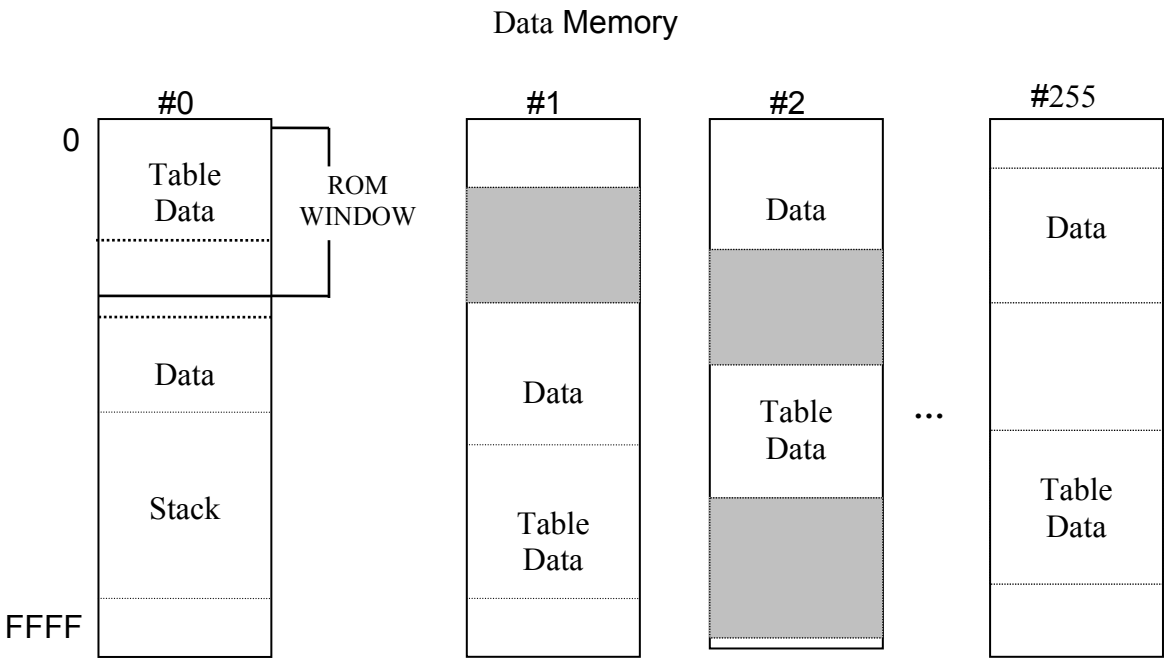
Segment Register usage of small memory model

Access Region	Segment Register	Changes	Notes
Code	CSR	Never	
Data	DSR	Changes only for far data	CCU8 adds code for setting up DSR
Table data	DSR	Changes only for far data	CCU8 adds code for setting up DSR

The small memory model consists of only one code segment and more than one data segments. CCU8 assigns code to physical segment #0 of program memory and data (table or otherwise) to physical segment #0 - #255 of data memory. ROM WINDOW area lies only at the physical segment #0 of data memory, that will be used to store table data also. If there is no far data (table or otherwise), the data memory address space for physical segments #1 and higher will be empty.

4.1.2 Large Memory Model





Segment Register usage of large memory model

Access Region	Segment Register	Changes	Notes
Code	CSR	Yes	Hardware will automatically save and restore CSR with PC/LR/ELR
Data	DSR	Changes only for far data	CCU8 adds code for setting up DSR
Table data	DSR	Changes only for far data	CCU8 adds code for setting up DSR

The large memory model consists of more than one code segment and more than one data segment. It will assign code to physical segment #0 - #15 of program memory and data (table or otherwise) to physical segment #0 - #255 of data memory. For physical segment #1 and higher, program memory and data memory shares the same address space with individual physical segments assigned to one or the other. The shaded address space will be assigned to

the other type of memory and it will not be available. If there is no far data, the data memory address space for physical segments #1 and higher will be empty.

4.2 DATA ACCESS

CCU8 uses a fixed pointer size determined by the data to be accessed (__near/__far).

Near data will be present in physical data segment #0. Hence, near data access will be done with two byte pointers. These two byte pointers are called as near pointers.

Far data will be present in physical data segment #0 and above. Hence, far data access will be done with three-byte pointers. These three-byte pointers are called as large pointers.

The type for pointers declared without data access specifier is determined by the /near or /far command line options or NEAR / FAR pragmas.

To treat particular data items as far data, __far specifier will be used with /near command line option. This specifier will override the default from the command line option, causing CCU8 to insert the DSR setup code for accessing the far data.

On the other hand, there will be a means to override the /far command line option to make data near. __near specifier will be used, causing CCU8 to skip the DSR setup code for accessing the near data.

The __near and __far specifiers cannot be used with functions.

All these data access specifiers are mutually exclusive.

5. PRAGMAS

Syntax :

#pragma pragma_keyword arguments

The directive **#pragma** directs CCU8 to define architecture specific instructions in the assembly listing file. Pragma with instructions not recognized by the compiler are ignored after issuing a warning message. Pragma keywords are not case sensitive. The pragmas supported by CCU8 are explained in this section.

By default, the delimiter which separates the pragma arguments is whitespace. This default delimiter can be changed to "," (comma) by specifying /PF option in the command line.

5.1 INTERRUPT PRAGMA

Syntax:

a. /PF option specified:

#pragma INTERRUPT function_name, address, [category]

b. /PF option not specified:

#pragma INTERRUPT function_name address [category]

The pragma **interrupt** is used to specify interrupt handling functions coded in 'C'. If a function is defined in 'C' source program with **function_name** specified in this pragma, then it is treated as an interrupt handling routine. This pragma must appear before definition of the function specified in the pragma. If this pragma appears after the definition, pragma is ignored after issuing a warning message.

The **function_name** in this pragma specifies the name of the interrupt handling function. The **function_name** must be followed by an interrupt vector **address**. The value must be an even address in the range, 0x8 and 0x7e or must be 0x4, inclusive of both. The **address** is followed by **category** and it is optional. The value of **category** must be either 1 or 2. If the value of **category** is 1, it indicates prohibit multiple interrupt. In this case if "**__EI**" built-in function is specified in this function or any calls to another Interrupt/SWI function, compiler issues

error. If the value of **category** is 2, it indicates allow multiple interrupt. If category is not specified, **category** defaults to 2 and it indicates allow multiple interrupt.

If this pragma is used more than once with the same interrupt vector address but different function names, compiler issues a warning and takes the first pragma as valid. However, same function name may be specified with different interrupt vector addresses.

The **function name** must be a static function. If it is not a static function, **CCU8** issues warning message and treat it as static function.

CCU8 pushes all registers used in interrupt handling function at the entry to this function and it pops the corresponding registers at the exit.

CCU8 issues warning for the following cases:

- If the specified symbol is not a function.
- If function “main” is specified in this pragma.
- If a function specified in this pragma is not declared in the file being compiled.
- If a function specified in this pragma has arguments or returns a value.
- If a function specified in this pragma is already specified in a pragma directive other than **interrupt**.
- If the pragma is specified after the function definition.
- If a function specified in this pragma is used in an expression.
- If the specified address is not in the range 0x8 and 0x7E, inclusive of both or other than value 0x04.
- If an odd address is specified.
- If the value of category is other than 1 and 2.
- If the function is called in source file.

The following are erroneous cases :

Example 5.1

INPUT

```
int a ;
# pragma interrupt a 0x10
```

In the above example, variable ‘a’ is not a function.

Example 5.2

INPUT

```
static void function () ;
# pragma interrupt function 0x09
```

In the above example, an odd address is specified.

Example 5.3

INPUT

```
static int int10 (void) ;
# pragma interrupt int10 0x10
```

In the above example, ‘int10’ has return value.

5.1.1 Preserving Register Contents

To ensure that a program runs correctly after an interrupt is serviced, CCU8 saves the registers that may be used during the interrupt handling process.

5.1.1.1 INTERRUPT FUNCTION WITH CATEGORY 1 WITHOUT FUNCTION CALL

Example 5.4

INPUT

```
static void intr1_1a () ;
# pragma interrupt intr1_1a 0xA 1
int a, b, c ;
static void
intr1_1a ()
{

        a = b + c ;
}
```

The following is the code generated for the above interrupt function definition:

OUTPUT

```
        rseg $$INTERRUPTCODE

_intr1_1a       :
        push    xr0

;;      a = b + c ;
```

```
                l       er0,    NEAR _b
                l       er2,    NEAR _c
                add     er0,    er2
                st      er0,    NEAR _a

        ;;}
                pop     xr0
                rti
```

5.1.1.2 INTERRUPT FUNCTION WITH CATEGORY 1 WITH FUNCTION CALL

Example 5.5

INPUT

```
static void intr () ;
# pragma interrupt intr 0xA 1
int a, b, c ;

static void
intr ()
{
        a = b + c ;
        fn1 () ;
}
```

OUTPUT

```
                rseg $$INTERRUPTCODE

        _intr   :
                push    lr, ea
                push    xr0
                l       r0,     DSR
                push    r0

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

        ;;      fn1 () ;
```

```
        bl      _fn1

;;}
        pop     r0
        st      r0,     DSR
        pop     xr0
        pop     ea, lr
        rti
```

5.1.1.3 INTERRUPT FUNCTION WITH CATEGORY 2 WITHOUT FUNCTION CALL

Example 5.6

INPUT

```
static void intr1_2a () ;
/* # pragma interrupt intr1_2a 0xA
*/
# pragma interrupt intr1_2a 0xA 2
int a, b, c ;

static void
intr1_2a ()
{
        a = b + c ;
        __EI () ;
}
```

The following is the code generated for the above interrupt function definition:

OUTPUT

```
_intr1_2a       :

        push    elr, epsw
        push    xr0

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

;;      __EI () ;
        ei

;;}
        pop     xr0
        pop     psw, pc
```

5.1.1.4 INTERRUPT FUNCTION WITH CATEGORY 2 WITH FUNCTION CALL

Example 5.7

INPUT

```
static void intr1_2b () ;
# pragma interrupt intr1_2b 0xA
/* or
# pragma interrupt intr1_2b 0xA 2
*/
int a, b, c ;

static void
intr1_2b ()
{
        a = b + c ;
        __EI () ;

        fn1 () ;
}
```

The following is the code generated for the above interrupt function definition:

OUTPUT

```
_intr1_2b       :

        push    elr, epsw, lr, ea
        push    xr0
        l       r0,     DSR
        push    r0

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

;;      __EI () ;
        ei

;;      fn1 () ;
        bl      _fn1

;;}
        pop     r0
        st      r0,     DSR
```

```
pop     xr0
pop     ea, lr, psw, pc
```

5.1.2 Suppress the Use of DSR in Interrupt Function

When command line option /nofar is specified, CCU8 suppresses output of DSR save and return codes within the interrupt function when only data memory space of physical segment #0 is used.

5.1.2.1 INTERRUPT FUNCTION WITH CATEGORY 1 WITH FUNCTION CALL WITH NOFAR OPTION

Example 5.8

INPUT

```
static void InterruptFunction () ;
# pragma interrupt InterruptFunction 0xA 1
int a, b, c ;

static void
InterruptFunction ()
{
        a = b + c ;
        fn1 () ;
}
```

Compiling the above example with /nofar option

OUTPUT

```
rseg $$INTERRUPTCODE

_ InterruptFunction         :
        push    lr, ea
        push    xr0

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

;;      fn1 () ;
```

```
        bl      _fn1

;;}
        pop     xr0
        pop     ea, lr
        rti
```

5.1.2.2 INTERRUPT FUNCTION WITH CATEGORY 2 WITH FUNCTION CALL WITH NOFAR OPTION

Example 5.9

INPUT

```
static void InterruptFunction_2 () ;
# pragma interrupt InterruptFunction_2 0xA
/* or
# pragma interrupt InterruptFunction_2 0xA 2
*/
int a, b, c ;

static void
InterruptFunction_2 ()
{
        a = b + c ;
        __EI () ;

        fn1 () ;
}
```

The following is the code generated for the above interrupt function definition when /nofar option is specified in command line:

OUTPUT

```
_ InterruptFunction_2    :

        push    elr, epsw, lr, ea
        push    xr0

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

;;      __EI () ;
        ei
```

```
;;      fn1 () ;
        bl      _fn1

;;}
        pop     xr0
        pop     ea, lr, psw, pc
```

5.2 SWI PRAGMA

Syntax:

a. /PF option specified:

#pragma SWI function_name, address, [category]

b. /PF option not specified:

#pragma SWI function_name address [category]

The pragma **swi** is used to specify software interrupt handling functions coded in 'C'. If a function is defined in 'C' source program with **function_name** specified in this pragma, then it is treated as an software interrupt handling routine. This pragma must appear before definition of the function specified in the pragma. If this pragma appears after the definition, pragma is ignored after issuing a warning message. Extern functions may be specified in this pragma.

The **function_name** in this pragma specifies the name of the software interrupt handling function. The **function_name** must be followed by an interrupt vector **address**. The value must be an even address in the range, 0x80 and 0xfe, inclusive of both. The **address** is followed by **category** and it is optional. The value of **category** must be either 1 or 2. If the value of **category** is 1, it indicates prohibit multiple interrupt. In this case if "**__EI**" built-in function is described in this function or any calls to another SWI function is provided, compiler issues error. If the value of **category** is 2, it indicates allow multiple interrupt. If category is not specified **category** defaults to 2 and it indicates allow multiple interrupt.

If this pragma is used more than once with the same interrupt vector address but different function names, compiler issues a warning and takes the first pragma as valid. However, same function name may be specified with different interrupt vector addresses.

CCU8 pushes registers R4- R15 used in software interrupt handling functon at the entry to this function and it pops the corresponding registers at the exit. When the software interrupt function does not return a value and does not have argument variable, R0-R3 will also be preserved.

CCU8 issues warning for the following cases:

- If the specified symbol is not a function.
- If function “main” is specified in this pragma.
- If a function specified in this pragma is not declared in the file being compiled.
- If a function specified in this pragma has arguments or returns a value.
- If a function specified in this pragma is already specified in a pragma directive other than swi.
- If the pragma is specified after the function definition.
- If a function specified in this pragma is used in an expression.
- If the specified address is not in range 0x80 to 0xfe, inclusive of both.
- If an odd address is specified.
- If the value of category is other than 1 and 2.
- If a function referred before it is specified in SWI pragma
- If a function specified in SWI pragma is assigned to a pointer to function

CCU8 issues error for the following cases:

- If '__EI' built-in function is "called" in a category 1 SWI function
- If the category is 1 and any calls to another SWI function is provided

The following are erroneous cases:

Example 5.10

INPUT

```
int x;
# pragma SWI x 0x80
```

In the above example, variable ‘x’ is not a function.

Example 5.11

INPUT

```
static char function (int) ;

# pragma SWI function 0x09
```

In the above example, an odd address is specified.

Example 5.12

INPUT

```
int a;
void sfn1(void);
void (*fn)() = sfn1;
# pragma SWI sfn1 0x84 2            //Warning
void sfn1()
{
        a += 10;
}
void func1()
{
        (*fn)();
}
```

In the above example, the function ‘sfn’ is already referenced.

5.2.1 Preserving Register Contents

To ensure that a program runs correctly after a software interrupt is serviced, CCU8 saves the registers that may be used during the software interrupt handling process.

5.2.1.1 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 1 WITHOUT FUNCTION CALL

Example 5.13

INPUT

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
   a = b + c ;
}
```

The following is the code generated for the above software interrupt function definition:

OUTPUT

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    xr0

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

;;}
        pop     xr0
        rti
```

5.2.1.2 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 1 WITH FUNCTION CALL

Example 5.14

INPUT

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
   a = b + c ;
   function2();
}
```

The following is the code generated for the above software interrupt function definition:

OUTPUT

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    lr, ea
        push    xr0
        push    r4
        l       r4,     DSR
        push    r4

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

;;   function2();
        bl      _function2

;;}
        pop     r4
        st      r4,     DSR
        pop     r4
        pop     xr0
        pop     ea, lr
        rti
```

5.2.1.3 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 2 WITHOUT FUNCTION CALL

Example 5.15

INPUT

```
void function () ;
# pragma SWI function 0x80 2
int a, b, c ;
void
function()
{
   a = b + c ;

   __EI () ;
}
```

The following is the code generated for the above software interrupt function definition:

OUTPUT

```
            rseg $$INTERRUPTCODE

_function       :

;;{
            push    elr, epsw
            push    xr0

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

;;   __EI () ;
            ei

;;}
            pop     xr0
            pop     psw, pc
```

5.2.1.4 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 2 WITH FUNCTION CALL

Example 5.16

INPUT

```
void function () ;
void function2 () ;

# pragma SWI function 0x80 2
int a, b, c ;

void function()
{
   a = b + c ;

   __EI () ;

   function2 () ;
}
```

The following is the code generated for the above software interrupt function definition:

OUTPUT

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    elr, epsw, lr, ea
        push    xr0
        push    r4
        l       r4,     DSR
        push    r4

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

;;  __EI () ;
        ei

;;  function2 () ;
        bl      _function2
```

```
;;}
        pop     r4
        st      r4,     DSR
        pop     r4
        pop     xr0
        pop     ea, lr, psw, pc
```

5.2.2 Suppress the Use of DSR in SWI Function

When command line option /nofar is specified, CCU8 suppresses output of DSR save and return codes within the SWI function when only data memory space of physical segment #0 is used.

5.2.2.1 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 1 WITH FUNCTION CALL AND WITH NOFAR OPTION

Example 5.17

INPUT

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
   a = b + c ;
   function2();
}
```

The following is the code generated for the above software interrupt function definition when /nofar option is specified in the command line:

OUTPUT

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    lr, ea
        push    xr0

;;   a = b + c ;
        l       er0,    NEAR _b
        l       er2,    NEAR _c
        add     er0,    er2
```

```
        st      er0,    NEAR _a

;;  function2();
        bl      _function2

;;}
        pop     xr0
        pop     ea, lr
        rti
```

5.2.2.2 SOFTWARE INTERRUPT FUNCTION WITH CATEGORY 2 WITH FUNCTION CALL AND WITH NOFAR OPTION

Example 5.18

INPUT

```
void function () ;
void function2 () ;

# pragma SWI function 0x80 2
int a, b, c ;
void
function()
{
   a = b + c ;

   __EI () ;

   function2 () ;
}
```

The following is the code generated for the above software interrupt function definition when /nofar option is specified in command line:

OUTPUT

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    elr, epsw, lr, ea
        push    xr0

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

```
;;   __EI () ;
            ei

;;   function2 () ;
            bl          _function2

;;}
            pop         xr0
            pop         ea, lr, psw, pc
```

5.3 INLINE PRAGMA

Syntax:

a. /PF option specified:

#pragma INLINE function_name [, function_name ...]

b. /PF option not specified:

#pragma INLINE function_name [function_name ...]

The pragma **inline** is used to specify functions, which can be inlined instead of calling that function.

This pragma must appear before the definition of that function. If this pragma appears after the definition of the function, CCU8 issues a warning message.

A list of function names may be specified in this pragma. CCU8 issues warning message if symbols other than functions are specified in this pragma. The functions specified in this pragma are treated as **static** functions. So, functions specified in inline pragma should be defined in the same file.

A function specified in this pragma is not expanded (inlined) in the following cases:

- If the function has variable number of arguments.
- If the function is defined before the inline pragma specification.
- If the function contains switch statement (the case that generates switch-jump table).

  *1: A switch-jump table is generated when satisfying all the following conditions.

  1. A control expression type is either of char(signed /unsigned)/ short/int.
  2. There are six or more cases.

3. ((maximum of case) – (minimum of case)) / (the number of cases) is smaller than 4.

- If there is any asm block in the function.
- If the function is too big. (If the number of the dag node exceeds 99(*2) or the number of sub-function calling exceeds 3 in the target function)

  *2: In description of C language, when an expression including one operation is assumed to be one line, it becomes about 30 lines.

- If the function has no definition prior to its call.
- If the inline expansion level exceeds inline depth.
- If the inline function is called recursively when inlinerecursionon pragma is not specified.
- If the function is called indirectly via function pointer.
- If the optimization option Od is active then the function will not be inlined.

If all the inline function calls are expanded then code for the function body will not be generated. CCU8 outputs warning message if an inline function call is not expanded.

However, a function specified in this pragma is not expanded (inlined) and no warning message is generated in the following cases:

- If the calling function has ‘Od’ as the active optimization option. Which can be set either through the command line or the optimization pragma.
- If the function is called before specifying inline pragma.
- If the function is called indirectly via function pointer.

CCU8 issues warning for the following cases:

- If the specified symbol is not a function.
- If function “main” is specified in this pragma.
- If a function specified in this pragma is not defined in the file being compiled.
- If a function specified in this pragma is already specified in a pragma other than **inline**.
- If a call to inline function is not expanded (inlined).
- If the pragma is specified after the function definition.

Example 5.19

INPUT

```
int var ;
# pragma inline fn
int fn (int arg)
{
        return (arg*arg) ;
}
void fn1()
{
        var = fn (var) ;
}
```

OUTPUT

```
                type (mu8)
                model small, near
                $$NCOD5_5 segment code 2h #0h
                CFILE 0000H 00000011H "5_5.c"

                rseg $$NCOD5_5

_fn1    :
;;{
                push    lr

;;              var = fn (var) ;
                l       er0,    NEAR _var
                mov     er2,    er0
                bl      __imulu8sw
                st      er0,    NEAR _var

;;}
                pop     pc

                public _fn1
                _var comm data 02h #00h
                extrn code near : _main
                extrn code : __imulu8sw

                end
```

Example 5.20

INPUT

```
# pragma inline fn

void fn ()
{
        fn () ;
}

fn1 ()
{
        fn () ;
}
```

The inline function “fn” is not expanded since “fn” is called recursively when inline recursion is off.

5.4 ABSOLUTE PRAGMA

Syntax:

a. /PF option specified:

#pragma ABSOLUTE name, [segment:]offset

b. /PF option not specified:

#pragma ABSOLUTE name [segment:]offset

The pragma **absolute** assigns an absolute address to a global variable or static local variable.

If physical segment address is not specified, it is considered as zero. CCU8 issues a warning message when physical segment address other than zero is specified for near variables.

The physical segment address can take any value between 0 and 0xff, while the offset can take a value between 0 and 0xffff.

Absolute pragma can be specified for a variable before or after its declaration. Variables already initialized cannot be used in this pragma, however, this pragma can appear before the variable's initialization. If this pragma is used more than once for the same variable, CCU8 flags a warning and assigns the address specified with the latest pragma. If extern variables are specified in this pragma, this pragma is ignored.

The valid range of absolute address is as follows:

- Segment address 0x0 (for near variables)
0x0 to 0xff (for far and large variables)
- Offset address 0x0 to 0xffff

CCU8 issues warnings for the following cases :

- If the symbol specified in this pragma is not a global or static local variable.
- If the variable is already specified in any pragma.
- If the variable specified in this pragma is not declared within the same file.
- If an odd address is specified for initialized variables.
- If specified address is not in absolute range.
- If the pragma is specified after variable initialization.
- If the variable specified in segment #1h or higher is qualified by __near specifier

Example 5.21

INPUT

```
int acc ;
# pragma absolute acc 0x40
```

OUTPUT

```
        public _acc

        dseg #00h at 040h
_acc :
        ds      02h
        extrn code near : _main
```

Example 5.22

INPUT

```
long __far la ;
# pragma absolute la 0x2:0x1000
```

OUTPUT

```
        public _la

        dseg #02h at 01000h
_la :
        ds      04h
        extrn code near : _main
```

Following example illustrates an erroneous case:

Example 5.23

INPUT

```
# pragma absolute abs_data_var 100
void fn (void)
{
        int     abs_data_var    ;
}
```

In the above example, local variable “abs_data_var” is specified in the pragma.

5.5 ROMWIN PRAGMA

Syntax:

a. /PF option specified:

#pragma ROMWIN start_address, end_address

b. /PF option not specified:

#pragma ROMWIN start_address end_address

The pragma **romwin** instructs the compiler about the ROM window boundaries.
The valid range of ROM window boundaries is as follows:

- The start_address should be greater than or equal to zero
- The end_address should be greater than the start_address
- The maximum value allowed for end_address is 0xffff

Example 5.24

INPUT

#pragma ROMWIN 0 0x8fff

In this example, ROM window boundaries are set from 0h to 0x8fffh.

OUTPUT

romwindow 0h, 08fffh

Example 5.25

INPUT

#pragma ROMWIN 0x1000 0x9fff

In this example, ROM window boundaries are set from 0x1000 to 0x9fff

OUTPUT

romwindow 1000h, 09fffh

CCU8 issues warning for the following erroneous cases :

- If the end_address is less than start_address

  Example 5.26

  ```
  #pragma ROMWIN 0x3000 0x2fff
  ```

- If the max address of end_address is greater than 0xffff.

  Example 5.27

  ```
  #pragma ROMWIN 0 0x1ffff
  ```

5.6 NOROMWIN PRAGMA

Syntax:

```
#pragma NOROMWIN
```

The pragma **noromwin** instructs the compiler not to use ROM window area. The **romwin** pragma and **noromwin** pragma are mutually exclusive. If both the pragmas are defined, first defined pragma will be valid. CCU8 issues warning and ignores the latter pragma.

5.7 NVDATA PRAGMA

Syntax:

a. /PF option specified:

```
#pragma NVDATA variable [, variable ..]
```

b. /PF option not specified:

```
#pragma NVDATA variable [ variable ..]
```

The pragma **nvdata** instructs the compiler to allocate one or more global variables or **static** local variables given by the list of variables within the nonvolatile RAM region.

Local variables cannot be allocated in nonvolatile RAM region, because they are allocated in stack. CCU8 ignores **'const'** qualifier for a variable specified in this pragma after issuing a warning message and makes the variable's type as NVDATA. If a variable does not have a

data access specifier (__near or __far), CCU8 uses the default specified by the command line options /near or /far.

CCU8 issues warning for the following cases :

- If the specified symbol is not a global or **static** local variable.
- If a variable specified in this pragma is not declared in the file being compiled.
- If the **'const'** qualified variable is specified.
- If the variable is already specified in any pragma.
- If the variable is initialized before this pragma directive.

Example 5.28

INPUT

```
int nvdata_var ;
# pragma nvdata nvdata_var
```

OUTPUT

```
        $$NNVDATA5_14 segment nvdata 2h  #0h
CFILE 0000H 00000003H "5_14.c"
        public _nvdata_var
        extrn code near : _main

        rseg $$NNVDATA5_14
_nvdata_var :
        dw      00h
```

5.8 CHECKSTACK PRAGMAS

Syntax :

```
#pragma CHECKSTACK_ON

#pragma CHECKSTACK_OFF
```

The pragma **checkstack_on** instructs the compiler to add a call to stack probe routine in entry code of functions defined after this pragma.

The pragma **checkstack_off** instructs the compiler not to add a call to the stack probe routine in entry code of functions defined after this pragma.

These two pragmas are processed irrespective of /ST option in the command line.

Example 5.29

INPUT

```
# pragma CHECKSTACK_ON
void fn (void)
{
        fn1 (0) ;
        fn2 (1) ;
}
```

CCU8 generates the following code for function “fn” :

OUTPUT

```
_fn     :

        push    lr
        mov     er0,    #04h
        bl      __chstu8sw

;;      fn1 (0) ;
        mov     er0,    #0
        bl      _fn1

;;      fn2 (1) ;
        mov     er0,    #1
        bl      _fn2

;;}
        pop     pc

        extrn code near : _fn2
        extrn code near : _fn1
        public _fn
        extrn code near : _main
        extrn code : __chstu8sw

        end
```

5.9 OPTIMIZATION PRAGMAS

5.9.1 OPTIMIZATION PRAGMA

Syntax:

a. /PF option specified:

#pragma OPTIMIZATION [argument [,argument ..]]

b. /PF option not specified:

#pragma OPTIMIZATION [argument [argument ..]]

This pragma will set the optimization options for the individual functions. In optimization pragma, the priority higher than an optimization option is given.

These options determine the type of optimization to be performed on the function.

The arguments can be among of the following options.

1. Od : Disable optimization
2. Ol : Optimize loops
3. Og : Perform Global Optimization
4. Om : Do maximum optimization possible
   (When this option is specified, Orp is specified automatically)
5. Ot : Do speed optimization
6. default : Do the compiler default optimizations.
7. Oa : Perform alias checks
8. Orp : Optimize register push/pop
9. Orpn : Do not optimize register push/pop

Note : All the options are case sensitive

The pragma remains in effect till a second optimization pragma is encountered or the end of the file, if none is specified.

If no argument is specified in the pragma, then the optimization options indicated by the command line are taken as the active optimization options for the subsequent functions

If Orp or Orpn is only specified, then the optimization options indicated by the command line are taken as the active optimization options subsequent functions.

But if /Od is specified in command line option and the argument of optimization pragma is Orp only, then CCU8 issues a warning and ignores the pragma.

The pragma will be ignored in the following conditions

- The pragma is specified inside a function definition.
- The phrase that is specified as an argument is not a valid argument as given in the table above.
- When multiple arguments are specified, the arguments violate any of the restrictions described below.
  - Od cannot be specified with any other arguments.
  - 'default' cannot be specified with other arguments (except Orp and Orpn).
  - Om and Ot cannot be specified concurrently.
  - Orp and Orpn cannot be specified concurrently.

It shows the relation of the arguments of optimization pragma and optimization items.

argument (option) / optimization item			Od	Ol	Og	Om	Ot	default
Local optimization			ON	ON	ON	ON	ON	ON
Peephole optimization			ON	ON	ON	ON	ON	ON
Loop optimization			-	ON	-	ON	ON	ON
Global optimization			-	-	ON	ON	ON	ON
	Code sinking Code hoisting		-	-	ON	ON	ON	-
Speed optimization			-	-	-	-	ON	-
Other optimization			-	ON	ON	ON	ON	ON
Optimization of register push/pop		**Orp** *2	- *1	ON	ON	ON	ON	ON
		nothing	-	-	-	ON	-	-
Suppress function	**Suppress optimization of register push/pop**	**Orpn** *2	- *1	ON *3	ON *3	ON *3	ON *3	ON *3
		nothing	-	-	-	-	-	-
	Suppress optimization by alias checking	Oa	- *1	ON	ON	ON	ON	ON
		nothing	ON *4	-	-	-	-	-

*1 : The gray field shows combination is improper. (CCU8 issues a warning.)

*2 : Orp and Orpn cannot be specified concurrently. (CCU8 issues a warning.)

*3 : When Ol, Og, Ot, or default is specified,optimization of register push/pop does not perform even if Orpn is not specified. When Om is only specified, Orpn has effect.

*4 : When Od is specified, CCU8 treats as Oa is specified automatically.

The OPT_ON and OPT_OFF pragmas are affected in the same way by this pragma as the command line option.

Example 5.30

INPUT

source.c
command line : ccu8 /Tu8 /Om source.c

```
int a, b, c, d;

#pragma OPT_OFF                 // Invalid by the command line option /Om

void Om_func()          //  Performs optimization by /Om
{
        a = b + c;
        if (a != 0)
        {
                d = b + c;      // Deletes the common expressions
        }
 }

#pragma OPTIMIZATION default  // Equivalent to the case where a
                              // default value is used for the
                              // command line option

                              // #pragma OPT_OFF is valid
void Od_func()                // Suppresses optimization
{
        a = b + c;
        if (a != 0)
        {
                d = b + c;
        }
}

#pragma OPTIMIZATION Ol Og   // Equivalent to the case where the
                              // command line options are /Ol /Og

#pragma OPT_ON                        // #pragma OPT_ON is valid

void default_func()           // Performs default optimization
{
```

```
        a = b + c;
        if (a != 0)
        {
                d = b + c;          // Deletes the common expressions
        }
}

#pragma OPTIMIZATION Od                 // Equivalent to the case where the
                                        // command line option is /Od

                                        // #pragma OPT_ON is invalid
void Od_func_again()                    // Suppresses optimization
{
        a = b + c;
        if (a != 0)
        {
                d = b + c;
        }
}
```

At compilation with optimization where the inline function definition and the calling function are different, inline expansion is performed only when both the inline function definition and the calling function are compiled with optimization that allows inline expansion. For the functions for which inline expansion was not performed, the compiler generates the entity codes at the end.

Example 5.31

INPUT

Source2.c

command line : ccu8 /Tu8 /Om source2.c

```
int Om_val, Od_val;

#pragma inline Om_Ifunc Od_Ifunc
void Om_Ifunc()     /* Generates the entity when inline expansion is not performed */
{
        Om_val = 0;
}

#pragma OPTIMIZATION Od  /* Suppresses optimization */
void Od_Ifunc()     /* Generates the entity since inline expansion is not performed */
{
        Od_val = 0;
}
```

```
#pragma OPTIMIZATION     /* Restarts optimization */
void Om_func()
{
        Od_Ifunc();       /* Inline expansion is not performed */
        Om_Ifunc();       /* <---Inline expansion is performed */
}

#pragma OPTIMIZATION Od  /* Suppresses optimization */
void Od_func()
{
        Od_Ifunc();       /* Inline expansion is not performed */
        Om_Ifunc();       /* Inline expansion is not performed */
}
```

OUTPUT:

```
_Od_Ifunc       :

;;{

;;      Od_val = 0;
        mov     er0,    #0
        st      er0,    NEAR _Od_val

;;}
        rt

_Om_func        :

;;{

;;      Od_Ifunc();  /* Inline expansion is not performed */
        bl      _Od_Ifunc

;;      Om_Ifunc();  /* <--Inline expansion is performed */
        mov     er0,    #0
        st      er0,    NEAR _Om_val

;;}
        rt

_Od_func        :

;;{
        push    lr

;;      Od_Ifunc(); /* Inline expansion is not performed */
        bl      _Od_Ifunc

;;      Om_Ifunc(); /* Inline expansion is not performed */
        bl      _Om_Ifunc
```

```
;;}
        pop     pc

_Om_Ifunc       :   ;<----Generates the entity also, since  inline expansion has not been
                    performed.
;;{

;;      Om_val = 0;
        mov     er0,    #0
        st      er0,    NEAR _Om_val

;;}
        rt
```

CCU8 issues warning for the following cases and the pragma is ignored:

- If the pragma is specified within the function body.
- If the phrase that is specified as an argument is not a valid argument.
- If option ‘Od’ is specified with some other option.
- If the option ‘default’ is specified with some other option except Orp and Orpn.
- If options ‘Om’ and ‘Ot’ are specified concurrently.
- If options ‘Orp’ and ‘Orpn’ are specified concurrently.

Example 5.32

INPUT

```
source1.c
command line : ccu8 /Tu8 /Om source1.c
int a, b, c, d;

void Om_func()
{
        a = b + c;
        if (a != 0) {
                d = b + c; /* Deletes the common expressions */
        }
}

#pragma OPTIMIZATION Od  /* Suppresses subsequent optimization */
void Od_func()
{
        a = b + c;
```

```
        if (a != 0) {
                d = b + c;
        }
}
#pragma OPTIMIZATION    /* Resets optimization to the original state */

void Om_func_again()
{
        a = b + c;
        if (a != 0) {
                d = b + c; /* Deletes the common expressions */
        }
}
```

OUTPUT:

Snip of the code generated by CCu8.

```
_Om_func        :

;;{

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

;;      if (a != 0) {
        mov     er0,    er0
        beq     _$L1

;;              d = b + c; /* Deletes the common expressions */
        st      er0,    NEAR _d

;;      }
_$L1 :

;;}
        rt


_Od_func        :

;;{

;;      a = b + c;
        l       er0,    NEAR _b
        l       er2,    NEAR _c
        add     er0,    er2
```

```
            st      er0,    NEAR _a

;;          if (a != 0) {
            mov     er0,    er0
            beq     _$L4

;;                  d = b + c; <- Suppresses optimization
            l       er0,    NEAR _b
            add     er0,    er2
            st      er0,    NEAR _d

;;          }
_$L4 :

;;}
 rt

_Om_func_again :

;;{

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

;;          if (a != 0) {
            mov     er0,    er0
            beq     _$L7

;;                  d = b + c; /* Deletes the common expressions */
            st      er0,    NEAR _d

;;          }
_$L7 :

;;}
rt
```

5.9.2 OPT_ON AND OPT_OFF PRAGMAS

Syntax :

```
#pragma OPT_ON
#pragma OPT_OFF
```

The pragma **opt_on** instructs the compiler to enable optimizations for functions that are defined after this pragma. The enabled optimization consists of the default optimizations applied in the absence of any optimization command line options. This pragma is ignored if the command line option /Od is specified.

The pragma **opt_off** instructs the compiler not to perform optimizations for functions defined after this pragma. The disabled optimization consists of the default optimizations applied in the absence of any optimization command line options. This pragma is ignored if the command line option /Om is specified.

5.10 ASM AND ENDASM PRAGMAS

Syntax :

```
# pragma ASM
...                          /* assembly instruction block */
#pragma ENDASM
```

The pragmas “asm” and “endasm” are similar to the directives “#asm” and “#endasm”. Any text can be given inside “#pragma asm” and “#pragma endasm”. CCU8 does not process this block of text. This block will be output in the assembly listing file as given in the source file.

CCU8 issues warning for the following case:

- If an **endasm** pragma is specified without its corresponding **asm** pragma.

CCU8 issues fatal error message for the following case:

- If an **asm** pragma is specified without its corresponding **endasm** pragma.

The following example shows the usage of “#pragma asm - # pragma endasm”

Example 5.33

INPUT

```
fn ()
{
# pragma asm
        cplc                    ;;      invert carry flag
        di                      ;;      disable interrupt
# pragma endasm
}
```

CCU8 generates the following function body for function “fn”:

OUTPUT

```
_fn     :

;;# pragma asm
        cplc                    ;;      invert carry flag
        di                      ;;      disable interrupt

;;}
        rt
```

The following are erroneous cases:

Example 5.34

INPUT

```
fn ()
{
# pragma endasm
# pragma asm
        cplc                    ;;      invert carry flag
        di                      ;;      disable interrupt
# pragma endasm
}
```

CCU8 issues a warning message for the first **endasm** pragma because, it is specified without its corresponding **asm** pragma.

Example 5.35

INPUT

```
fn ()
{
# pragma asm
        cplc                    ;;      invert carry flag
        di                      ;;      disable interrupt
}
```

CCU8 issues a fatal error message for the **asm** pragma because, its corresponding **endasm** pragma is not specified.

5.11 INLINEDEPTH PRAGMA

Syntax :

#pragma INLINEDEPTH constant

The pragma **inlinedepth** specifies the number of levels of inline function calls that can be expanded. The constant can take values between 0 and 255. The pragma **inlinedepth** pragma can be specified anywhere in a source file and any number of times. The pragma **inlinedepth** has effect on a source line, if the pragma declaration precedes that source line.

If **inlinedepth** pragma is not specified, inline depth is set to 8, by default. Inline functions are not expanded, if inline depth is specified as 0.

CCU8 issues warning for the following cases:

- If the constant value is not between 0 and 255.
- If a token type other than constant is specified in **inlinedepth** pragma.

Example 5.36

INPUT

```
int     a, b ;

#pragma inline fn1 fn2
#pragma inlinedepth 3

void fn1 ()
{
        a ++ ;
}

void fn2 ()
{
        b ++ ;
        fn1 () ;
}

void fn ()
{
        fn2 () ;
}
```

CCU8 generates the following code for function “fn”:

OUTPUT

```
_fn     :

;;      fn2 () ;
        l       er0,    NEAR _b
        add     er0,    #1
        st      er0,    NEAR _b
        l       er0,    NEAR _a
        add     er0,    #1
        st      er0,    NEAR _a

;;}
        rt
```

5.12 INLINERECURSION PRAGMAS

Syntax :

#pragma INLINERECURSIONON

#pragma INLINERECURSIONOFF

inlinerecursion pragma specifies whether a call to a recursive inline function is to be expanded or not. The recursion in the inline function may be direct or indirect.

inlinerecursion pragma can be specified anywhere in a source file and any number of times.

inlinerecursion pragma has effect on a source line if the pragma declaration precedes that source line.

A call to a recursive inline function is expanded if **inlinerecursionon** pragma is specified. The number of levels a recursive call expanded is determined by the specified inline depth. When the expansion level of a recursive call is exceeded, warning is issued, and it is not expanded after that.

A call to a recursive inline function is not expanded if **inlinerecursionoff** pragma is specified.

If **inlinerecursion** pragma is not specified, call to a recursive inline function is not expanded, by default.

Example 5.37

INPUT

```
int     a ;

#pragma Inline              fn
#pragma Inlinedepth         3
#pragma Inlinerecursionon

void fn ()
{
        a ++ ;
        fn () ;
}

main ()
{
        fn () ;
}
```

CCU8 generates the following code for the above example:

OUTPUT

```
_main   :
;;{

;;      fn () ;
        l       er0,    NEAR _a
        add     er0,    #1
        add     er0,    #1
        add     er0,    #1
        st      er0,    NEAR _a
        bl      _fn

;;}
_$$end_of_main :
        bal     $


_fn     :

;;}
        l       er0,    NEAR _a
        add     er0,    #1
        add     er0,    #1
        add     er0,    #1
        add     er0,    #1
        st      er0,    NEAR _a
        b       _fn
```

5.13 STACKSIZE PRAGMA

Syntax :

#pragma STACKSIZE constant

The pragma **stacksize** sets the stack size. The constant specifies the size of the stack in bytes. Any even value between 0x0 and 0xffff may be specified as the stack size. This pragma and the command line option /SS behave in the same way.

If /SS option is specified in the command line then this pragma will be ignored without giving warning message. CCU8 issues warning message if the pragma is specified more than once in the source file and the first time specified pragma stack size will be taken. This pragma is valid only if the source file has “main” function definition.

CCU8 issues warning for the following cases:

- If the pragma is specified more than once in the source file.

The following example shows erroneous case:

Example 5.38

INPUT

```
# pragma STACKSIZE 3001
```

For the above pragma, CCU8 issues warning message since the stacksize pragma specifies an odd number as stacksize.

5.14 NEAR AND FAR PRAGMAS

Syntax :

```
#pragma NEAR
#pragma FAR
```

The pragma NEAR/FAR can be used for applying the default data specifier for part(s) of a source file.

The pragma NEAR specifies __near as the default data specifier for data (table or otherwise) variables following it.

The pragma FAR specifies __far as the default data specifier for data (table or otherwise) variables following it.

These pragmas do not override the explicitly specified data specifier __near or __far for data (table or otherwise) variables.

The NEAR and FAR pragmas take precedence over the /near and /far command line options, if there is a mismatch between the two.

Data Model shown in the output file depends upon the Command Line data model specification (explicit or otherwise). However, the data model used for compilation may be more than one depending upon the use of pragma NEAR/FAR inside the source file. The interpretation by compiler in such cases is as follows:

(a) When #pragma NEAR/ FAR is not specified in the source file

The compilation of code is done in 'near' or 'far' mode as specified in the command line (default is 'near'). In this case, the entire source file is compiled in this data model.

(b) When #pragma NEAR/ FAR is specified in the source file once

The compilation of code is done in 'near' or 'far' mode as specified in the command line (default is 'near') till pragma NEAR/ FAR is encountered. Thereafter, the pragma specification takes precedence and the compilation is done in the data model specified by the pragma.

(c) When #pragma NEAR/ FAR is specified in the source file more than once

Multiple occurrences of pragmas NEAR/ FAR are permissible. In such a case, whenever a new pragma occurs, the compilation of the code that follows this new pragma is done in the newly specified data model till another pragma NEAR/FAR is encountered or the end of file is reached.

Example 5.39

```
                <code in part 1>
#pragma NEAR
                <code in part 2>
#pragma FAR
                <code in part 3>
#pragma NEAR
                <code in part 4>
```

In the above example, when the file is compiled with /far command line option, the following interpretation is done:

In part 1 compiler treats all default data variables as far data.
In part 2 compiler treats all default data variables as near data.
In part 3 compiler treats all default data variables as far data.
In part 4 compiler treats all default data variables as near data.

Example 5.40

INPUT

```
# pragma NEAR
int a, b ;
void
func1 ()
{
        a = b ;
}
# pragma FAR
int d, e ;
void
func2 ()
{
        d = e ;
}
```

CCU8 generates the following code for the above example:

OUTPUT

```
_func1  :

;;      a = b ;
        l       er0,    NEAR _b
        st      er0,    NEAR _a

;;}
        rt

_func2  :
```

```
;;      d = e ;
        l       er0,    FAR _e
        st      er0,    FAR _d

;;}
        rt
```

5.15 FASTFLOAT PRAGMA

Syntax :

#pragma FASTFLOAT

The pragma **fastfloat** instructs the compiler to use fast emulation library.

This pragma is supported to reduce the execution time for float type. This pragma is ignored if the command line option /Ff is specified. CCU8 issues warning message if this pragma is specified more than once.

5.16 SEGMENT PRAGMAS

The pragma **segment** instructs the compiler to modify the segment which allocates function and variable.
The following pragmas are prepared as pragma **segment**.

SEGCODE pragma	: allocates the function to specified segment
SEGINTR pragma	: allocates the interrupt function to specified segment
SEGINIT pragma	: allocates the initialized variable to specified segment
SEGNOINIT pragma	: allocates the non-initialized variable to specified segment
SEGCONST pragma	: allocates the variable with const qualifier to specified segment
SEGNVDATA pragma	: allocates the variable located to nonvolatile memory to specified segment

5.16.1 SEGCODE PRAGMA

The SEGCODE pragma allows the user to specify a relocatable segment name or an absolute address for the assembly language output of normal functions (Non-interrupt function).

Syntax :

Variant 1 : #pragma SEGCODE "SegmentName"
Variant 2 : #pragma SEGCODE [segment:]Address
Variant 3 : #pragma SEGCODE

The first variant specifies the relocatable segment name for the assembly language output of normal functions (Non-interrupt function).

The second variant specifies the starting address for assigning the assembly language output of normal functions (Non-interrupt function). The address which can be specified is shown below.

Small memory model : 0x0:0x0004 - 0x0:0xfffe

Large memory model : 0x0:0x0004 - 0xf:0xfffe

The third varant specifies default segment name for an assembly code of all functions, except interrupt functions.The default segment name is shown below.

/Zc command line option not spesified:

$$*funcname*$*filename*

/Zc command line option spesified:

Small memory model : $$NCODfilename

Large memory model : $$FCODfilename

Rules to identify functions affected by 'segcode' pragma:

A function 'X' shall have a body

A function 'X' shall not be an inline function

A function 'X' shall not be defined by either SWI or INTERRUPT pragma

The above mentioned rules are applicable to functions that are in SEGCODE pragma scope

Scope of segcode pragma:

Scope of segcode pragma can be terminated by another segcode pragma of any variant.
Let's say,

Current effective segcode pragma is variant 1 (i.e. segment name)
This effect will be terminated by any of the following variant

Segcode with segment name (variant 1)

Segcode with absolute address (variant 2)

Default Segcode (variant 3)

More than one segcode pragmas can't have either same segment name or absolute address. The second segcode pragma will be ignored.

OTHER POINTS

SEGCODE pragma affects only bodies of the function

SEGCODE pragma does not apply to function prototypes

SEGCODE pragma does not affect the INLINE functions

Unexpanded code goes to the default segment

Expanded code goes to same as calling function

SEGCODE pragma does not affect the interrupt functions, those are defined with SWI or INTERRUPT pragmas

The pragma keyword 'SEGCODE' is not a case sensitive

If the pragma ‘SEGCODE’ is declared witin the function body, it affects for the next function or subsequent ones.

Example 5.41

INPUT

```
int a;

#pragma Segcode "Segname1" /* Segment name is set to "Segname1"
                              Segment is Relocatable */
void fn1(void)
{
         a += 10;
}

#pragma Segcode 0x1000     /* Segment name is default
                              Segment is allocated to 0x1000 */
void fn2(void)
{
         a += 20;
}
```

```
#pragma Segcode "Segname3" /* Segment name is set to "Segname3"
                              Segment is Relocatable */
void fn3(void)
{
         a += 30;
}

#pragma Segcode           /* Segment name is default
                              Segment is Relocatable */
void fn4(void)
{
         a += 40;
}

#pragma Segcode 0x2000    /* Segment name is default
                              Segment is allocated to 0x2000 */
void fn5(void)
{
         a += 50;
}

#pragma Segcode           /* Segment name is default
                              Segment is Relocatable */
void fn6(void)
{
         a += 60;
}
```

5.16.2 SEGINTR PRAGMA

The SEGINTR pragma allows the user to specify a relocatable segment name or an absolute address for the assembly language output of interrupt functions.

Syntax :

Variant 1 : #pragma SEGINTR "SegmentName"
Variant 2 : #pragma SEGINTR address
Variant 3 : #pragma SEGINTR

The first variant specifies the relocatable segment name for the assembly language output of interrupt functions.

The second variant specifies the starting address for assigning the assembly language output of interrupt functions. The address which can be specified is 0x0004 – 0xfffe.

The third variant specifies the assembly language output of interrupt functions are assigning with default segment name. The default segment name is shown below.

/Zc command line option not spesified:

$$*funcname*$*filename*

/Zc command line option spesified:

$$INTERRUPTCODE

Rules to identify functions affected by 'segintr' pragma:

- A function 'X' shall have a body
- A function 'X' shall be defined by either SWI or INTERRUPT pragma

The above mentioned rules are applicable to functions that are in SEGINTR pragma scope

Scope of segintr pragma:

Scope of segintr pragma can be terminated by another segintr pragma of any variant.
Let's say,
Current effective segintr pragma is variant 1 (i.e. segment name)
This effect will be terminated by any of the following variant

- Segintr with segment name (variant 1)
- Segintr with absolute address (variant 2)
- Default segintr (variant 3)

More than one segintr pragmas can't have either same segment name or absolute address. The second segintr pragma will be ignored.

OTHER POINTS

- This SEGINTR pragma affects only bodies of the function
- SEGINTR pragma does not apply to function prototypes
- SEGINTR pragma does not affect the functions, those are defined with SWI or INTERRUPT pragmas. It affects only bodies of the interrupt function
- The pragma keyword 'SEGINTR' is not a case sensitive
- The pragma 'SEGINTR' can be declared within the function body

Example 5.42

INPUT

```
int a;
static void intr_fn1(void);
static void intr_fn2(void);
static void intr_fn3(void);
static void intr_fn4(void);
static void intr_fn5(void);
static void intr_fn6(void);

#pragma interrupt intr_fn1 0x8  1
#pragma interrupt intr_fn2 0xA  1
#pragma interrupt intr_fn3 0xC  1
```

```
#pragma interrupt intr_fn4 0xE  1
#pragma interrupt intr_fn5 0x10 1
#pragma interrupt intr_fn6 0x12 1

#pragma Segintr "Segname1" /* Segment name is set to "Segname1"
                              Segment is Relocatable */
static void intr_fn1(void)
{
         a += 10;
}

#pragma Segintr 0x1000     /* Segment name is default
                              Segment is allocated to 0x1000 */
static void intr_fn2(void)
{
         a += 20;
}

#pragma Segintr "Segname3" /* Segment name is set to "Segname3"
                              Segment is Relocatable */
static void intr_fn3(void)
{
         a += 30;
}

#pragma Segintr            /* Segment name is default
                              Segment is Relocatable */
static void intr_fn4(void)
{
         a += 40;
}

#pragma Segintr 0x2000     /* Segment name is default
                              Segment is allocated to 0x2000 */
static void intr_fn5(void)
{
         a += 50;
}

#pragma Segintr            /* Segment name is default
                              Segment is Relocatable */
static void intr_fn6(void)
{
         a += 60;
}
```

5.16.3 SEGINIT PRAGMA

SEGINIT pragma is used to allocate the initialized global variables, local static and global static variables to a user specified segment.

Syntax:

a. /PF option specified:

Variant 1 : #pragma Seginit [__near/__far] , “Name”
Variant 2 : #pragma Seginit [__near/__far] , Address
Variant 3 : #pragma Seginit [__near/__far]

b. /PF option not specified:

Variant 1 : #pragma Seginit [__near/__far] “Name”
Variant 2 : #pragma Seginit [__near/__far] Address
Variant 3 : #pragma Seginit [__near/__far]

The first variant directs the compiler to group all the subsequently affected variables in a relocatable data segment whose name is specified in the pragma, and also create a table segment with the name “Name”+”TAB” for the initial values.
The compiler will also generate an $$init_info.This will be used for initializing the variables with the initial values by the startup routine.

The second variant directs the compiler to assign all the affected variables, in the pragma scope, to the address starting at the address given in the pragma.
The address range depends upon the data access type of the pragma.

__near : the allowed range is 0x0:0x0 - 0x0:0xffff

__far : the allowed range is 0x0:0x0 - 0xff:0xffff

The third variant returns variable assignment to the default state.
#pragma Seginit __near will cancel the effect of any previous Seginit __near pragma
and #pragma Seginit __far will cancel the effect of any previous Seginit __far pragma.

Data Access Specifier [{__near | __far}] is optional .If the data access specifier is omitted in the pragma, the data access specifier currently active will be taken as the data access for the pragma, that is preference is given to.

1. The nearest ‘near’ or ‘far’ pragma specified earlier in the source file
2. Or, if none of the ‘near’/ ‘far’ pragmas is specified, the data access indicated by the command line option. (The default data access specifier is near)

Example 5.43

Command line Option is /far

```
#pragma far
#pragma Seginit __near “Name”
```

Data access for the pragma is __near

Example 5.44

Command line Option is /near

```
#pragma far
#pragma Seginit “Name”
```

Data Access for the pragma is __far

Example 5.45

Command line Option is /far

```
#pragma Seginit “Name”
```

Data Access Specifier is __far

Rules for identifying the variables that will be affected by the pragma:

The following rules are applicable for determining which variable should go to the Seginit segment.

If the variable is qualified by __near

- *Implicitly through nearest ‘near’ pragma or the command line option*
- *Or explicitly through the __near keyword in the pragma declaration.*

And the variable satisfies the following conditions

1. The variable is an initialized variable.
2. The variable is a global, global static or local static variable.
3. The variable is not specified with an absolute or nvdata pragma.

Then the variable will be grouped under the active Seginit near segment.

If the variable is qualified by __far

- *Implicitly through nearest FAR pragma or the command line option*
- *Or explicitly through the __far keyword in the variable declaration.*

And the variable satisfies the following conditions

1. The variable is an Initialized variable.
2. The variable is a global, global static or local static variable.
3. The variable is not specified with an absolute or nvdata pragma.

Then the variable will be grouped under the active Seginit far segment.

Scope of the pragma:

When a Seginit pragma is specified its effect will remain till

1. Any other Seginit pragma with the same data access specifier is specified.
2. Segnvdata (Name or Address variant) pragma with the same access specifier is specified
3. Or, if no other pragma (Seginit or Segnvdata) is specified, till the end of the file.

Example 5.46

```
#pragma Seginit __near “SegNear1”
#pragma Seginit __far “SegFar1”
int __near Var1 =1;                    // Grouped with SegNear1
int __far  Var2 =2;                    // Grouped With SegFar1

#pragma Seginit __near “SegNear2”      //Scope of SegNear1 is over
int __near Var3 =3;                    // Grouped with SegNear2
int __far Var4 =4;                     // Grouped With SegFar1

#pragma Seginit __far “SegFar2”        //Scope of SegFar1 is over
int __near Var5 =5;                    // Grouped With SegNear2;
int __far Var6 =6;                     // Grouped With SegFar2

#pragma Seginit __near 0x0:0x100       //Scope of SegNear2 is over
int __near Va7 =7;                     // grouped with Seginit __near  0x0:0x100
int __far Var8 =8;                     // Grouped With SegFar2

#pragma Segnvdata __far “Segnvdata1”   // Scope of Seginit __far is over
int __near Var9 =9;                    // Grouped with Seginit __near  0x0:0x100
int __far Var10 =10;                   // Grouped with Segnvdata1 Segment

#pragma Seginit __near                 //Scope of Seginit __near 0x0:0x100  is over
int __near Var11 =11;                  //Grouped with the default segment for near
int __far Var12 =12;                   // Grouped with Segnvdata1 Segment
```

Note :
The maximum length of a segment name who can specify by **seginit** pragma is the value which subtracted 3 from the greatest length that CCU8 can recognize as an identifier.
This is because the three characters "TAB" are attached after the segment name specified as the name of the segment generated as a table for initialization by **seginit** pragma. In addition, the length of an identifier can be set up by a command line option /SL. A default is 31 characters.

5.16.4 SEGNOINIT PRAGMA

SEGNOINIT pragma is used to specify a segment name or a segment starting address for uninitialized global, static global and static local variables.

Syntax:

a. /PF option specified:
Variant 1 : #pragma Segnoinit [__near/__far] , “Name”
Variant 2 : #pragma Segnoinit [__near/__far] , Address
Variant 3 : #pragma Segnoinit [__near/__far]

b. /PF option not specified:
Variant 1 : #pragma Segnoinit [__near/__far] “Name”
Varinat 2 : #pragma Segnoinit [__near/__far] Address
Variant 3 : #pragma Segnoinit [__near/__far]

The first variant instructs the compiler to assign the affected variables to a relocatable data segment. The name of relocatable data segment is same as specified in pragma.

The second variant directs the compiler to assign the affected variables starting at the specified address.

The third variant returns variable assignment to the default state. Default state is as follows.

Global variables without initializers are defined as communal symbols. The segment name used for all static variables without initializers, whether inside or outside functions is $$NVAR*filename* for the SMALL model or $$FVAR*filename* for the Large model.

Data Access Specifier [{__near | __far}] is optional .If the data access specifier is omitted in the pragma, the data access specifier currently active will be taken as the data access for the pragma, that is preference is given to.

1. The nearest ‘near’ or ‘far’ pragma specified earlier in the source file

2. Or, if none of the ‘near’/ ‘far’ pragmas is specified, the data access indicated by the command line option.
3. If none of the above is specified, default data access specifier 'near' will be considered.

Example 5.47

Command line Option is /far

```
#pragma far
#pragma Segnoinit __near "Name"
int __near var;
```

Data Access for the pragma is __near

Example 5.48

Command line Option is /near

```
#pragma far
#pragma Segnoinit "Name"
int __far var;
```

Data Access for the pragma is __far

Example 5.49

Command line Option is /far

```
#pragma Segnoinit "Name"
int __far var;
```

Data Access Specifier for pragma is __far

Example 5.50

No command line Option is specified)

```
#pragma Segnoinit "Name"
int __near var;
```

Data Access Specifier for pragma is __near

Rules for identifying the variables that will be affected by the pragma:

If the variable is qualified by __near and the variable satisfies the following conditions

- The variable is a global , global static , or local static variable

- The variable is uninitialized
- The variable is not qualified with const
- The variable is not specified with NVDATA or ABSOLUTE pragma.

Then the variable will be grouped under the active Segnoinit near segment.

If the variable is qualified by __far and the variable satisfies the following conditions

- The variable is a global , global static , or local static variable
- The variable is uninitialized
- The variable is not qualified with const
- The variable is not specified with NVDATA or ABSOLUTE pragmas.

Then the variable will be grouped under the active Segnoinit far segment.

Scope of the pragma

If a SEGNOINIT pragma is specified, its effect will remain until

1. Any other SEGNOINIT pragma with the same data access specifier is specified
2. Any other SEGNVDATA pragma (either name or address variant) with same data access specifier is specified
3. End of current source code file, if there is no SEGNOINIT or SEGNVDATA pragma with same data access type.

Example 5.51

INPUT

```
#pragma SEGNOINIT __near "SegNear1"
#pragma SEGNOINIT __far "SegFar1"
int __near Var1 ; // Grouped with SegNear1
int __far  Var2 ;   // Grouped With SegFar1

#pragma SEGNOINIT __near "SegNear2"    //Scope of SegNear1 is over
int __near Var3 ; // Grouped with SegNear2
int __far Var4 ;    // Grouped With SegFar1


#pragma SEGNOINIT __far "SegFar2"   //Scope of SegFar1 is over
int __near Var5 ; // Grouped With SegNear2;
int __far Var6 ;    // Grouped With SegFar2

#pragma SEGNOINIT __near 0x0:0x100                  //Scope of SegNear2 is over
int __near Var7 ; // Assigned to address  0x0:0x100
```

```
int __far Var8 ;    // Grouped With SegFar2

#pragma Segnvdata __far "Segnvdata1"        // Scope of SEGNOINIT __far is over
int __near Var9 ; // Assigned to address  0x0:0x102
int __far Var10 ;  // Grouped with Segnvdata1 Segment

#pragma SEGNOINIT __near                    //Scope of SEGNOINIT __near 0x0:0x100 is
over
int __near Var11 ;              // Assigned to default segment
int __far Var12 ;  // Grouped with Segnvdata1 Segment
```

OUTPUT

```
            rseg Segnvdata1
_Var10 :
            dw          00h
_Var12 :
            dw          00h

            dseg #00h at 0100h
_Var7 :
            ds          02h

            dseg #00h at 0102h
_Var9 :
            ds          02h

            rseg SegNear1
_Var1 :
            ds 02h


            rseg SegFar1
_Var2 :
            ds 02h
_Var4 :
            ds 02h


            rseg SegNear2
_Var3 :
            ds 02h
_Var5 :
            ds 02h


            rseg SegFar2
_Var6 :
            ds 02h
_Var8 :
```

```
ds 02h

_Var11 comm data 02h #00h

end
```

5.16.5 SEGCONST PRAGMA

SEGCONST pragma is used to group global, global static or local static variables qualified with ‘const’ in a user specified segment.

Syntax:

a. /PF option specified:
Variant 1 : #pragma Segconst [__near/__far] , “Name”
Variant 2 : #pragma Segconst [__near/__far] , Address
Variant 3 : #pragma Segconst [__near/__far]

b. /PF option not specified:
Variant 1 : #pragma Segconst [__near/__far] “Name”
Variant 2 : #pragma Segconst [__near/__far] Address
Variant 3 : #pragma Segconst [__near/__far]

The first variant directs the compiler to group all the subsequent affected variables into a relocatble table segment whose name is given in the pragma.

The second variant directs the compiler to group all the subsequent affected variables into absolute table segments whose address starts at the address specified in the pragma.
The address range depends upon the data access type of the pragma.

__near : the allowed range is 0x0:0x0004 - 0x0:0xffff

__far : the allowed range is 0x0:0x0004 - 0xff:0xffff

The third variant returns variable assignment to the default state.
#pragma Segconst __near will cancel the effect of any previous Segconst __near pragma
and #pragma Segconst __far will cancel the effect of any previous Segconst __far pragma

Data Access Specifier [{__near | __far}] is optional .If the data access specifier is omitted in the pragma, the data access specifier currently active will be taken as the data access for the pragma, that is preference is given to.

1. The nearest ‘near’ or ‘far’ pragma specified earlier in the source file

2. Or, if none of the ‘near’/ ‘far’ pragmas is specified, the data access indicated by the command line option. (The default data access specifier is near)

Example 5.52

Command line Option is /far

```
#pragma far
#pragma Segconst __near “Name”
```

Data access for the pragma is __near

Example 5.53

Command line Option is /near

```
#pragma far
#pragma Segconst “Name”
```

Data Access for the pragma is __far

Example 5.54

Command line Option is /far

```
#pragma  Segconst “Name”
```

Data Access Specifier is __far

Rules for identifying the variables that will be affected by the pragma:

The following rules are applicable for determining which variable should go to the Segconst segment.

If the variable is qualified by __near

- *Implicitly through nearest ‘near’ pragma or the command line option*
- *Or explicitly through the __near keyword in the pragma declaration.*

And the variable satisfies the following conditions

1. The variable is qualified by a const.
2. The variable is a global, global static or local static variable.
3. The variable is not specified with an absolute pragma.

Then the variable will be grouped under the active SEGCONST near segment.

If the variable is qualified by __far

- *Implicitly through nearest FAR pragma or the command line option*
- *Or explicitly through the __far keyword in the variable declaration.*

And the variable satisfies the following conditions

1. The variable is qualified by a const.
2. The variable is a global, global static or local static variable.
3. The variable is not specified with an absolute pragma.

Then the variable will be grouped under the active SEGCONST far segment.

Scope of the pragma

When a SEGCONST pragma is specified its effect will remain till

1. Any other Segconst pragma with the same data access specifier is specified.
2. Or, if no other Segconst pragma is specified, till the end of the file.

Example 5.55

```
#pragma Segconstt  __near “SegNear”
#pragma Segconst  __far “SegFar”

cosnt int __near  nearvar1 =1 ;  /* Will be grouped with SegNear */
const int __far  farvar1 =1 ;    /* Will be grouped with SegFar */

#pragma Segconst  __near         /* Scope of SegNear is over */

const int __near nearvar2 =2 ;   /* Will be grouped with the default
                                    segment for near */
const int __far  farvar2 =2 ;    /* Will be grouped with SefFar */

#pragma Segconst  __far          /* Scope of SegFar is over */

const int __near  nearvar3 =3 ;  /* Will be grouped with the default
                                    segment for near */
const int _near  farvar3 =3 ;    /* will be grouped with the default
                                    segment for far */
```

5.16.6 SEGNVDATA PRAGMA

SEGNVDATA pragma is used to assign a group of variables to a segment name or a segment starting address assigned to non volatile memory. This pragma assigns for a group of global, static global or static local variables.

Syntax:

a. /PF option specified:
Variant 1 : #pragma Segnvdata [__near/__far] , “Name”
Variant 2 : #pragma Segnvdata [__near/__far] , Address
Variant 3 : #pragma Segnvdata [__near/__far]

b. /PF option not specified:
Variant 1 : #pragma Segnvdata [__near/__far] “Name”
Variant 2 : #pragma Segnvdata [__near/__far] Address
Variant 3 : #pragma Segnvdata [__near/__far]

The first variant instructs the compiler to assign, subsequently defined variables with the specified data access type to the specified relocatable segment, the name of relocatable data segment is same as specified in pragma.

The second variant directs the compiler to assign the affected variables starting at the specified address. The valid range depends on the specified data access type.

__near : 0x0:0x0000 - 0x0:0xffff

__far : 0x0:0x0000 - 0xff:0xffff.

The third variant returns variable assignment to the default state. Default state is as follows.

Global variables without initializers are defined as communal symbols. The segment name used for all static variables without initializers, whether inside or outside functions is $$NVAR*filename* for the SMALL model and $$FVAR*filename* for the Large model. For variables with initializers, the segment name is $$NINITVAR for __near data and $$FINITVAR*filename* for __far data.

Data Access Specifier [{__near | __far}] is optional .If the data access specifier is omitted in the pragma, the data access specifier currently active will be taken as the data access for the pragma, that is preference is given to.

1. The nearest ‘near’ or ‘far’ pragma specified earlier in the source file
2. Or, if none of the ‘near’/ ‘far’ pragmas is specified, the data access indicated by the command line option.

3. If none of the above is specified, default data access specifier 'near' will be considered.

Rules for identifying the variables that will be affected by the pragma:

If the variable is qualified by _near and the variable satisfies the following conditions

- The variable is a global , global static , or local static variable
- The variable is not qualified with const
- The variable is not specified with NVDATA or ABSOLUTE pragma.

Then the variable will be grouped under the active SEGNVDATA __ near segment.

If the variable is qualified by __far and the variable satisfies the following conditions

- The variable is a global , global static , or local static variable
- The variable is not qualified with const
- The variable is not specified with NVDATA or ABSOLUTE pragma

Then the variable will be grouped under the active SEGNVDATA __far segment.

Scope of the pragma

If a SEGNVDATA pragma is specified its effect will remain until

- Any other SEGNVDATA pragma with same data access specifier is specified
- SEGNOINIT pragma (either name or address variant) with the same data access specifier is specified.
- SEGINIT pragma (either name or address variant) with the same data access specifier is specified.
- End of current source code file, if there is no SEGNVDATA, SEGNOINIT or SEGINIT pragma with same data access type.

Example 5.56

INPUT

```
#pragma SEGNVDATA __near "SegNear1"
#pragma SEGNVDATA __far "SegFar1"
int __near Var1  = 1;                  /* Grouped with SegNear1 */
int __far  Var2  =  1 ;                /* Grouped With SegFar1 */

#pragma SEGINIT  __near "SegNear2"  /* Scope of SegNear1 is over */
                                       /* but scope of  SegFar1 is still continue*/
int __near Var3  = 1 ;                 /* Grouped with SegNear2 */
```

```
int __far Var4 ;                        /* Grouped With SegFar1 */

#pragma SEGNVDATA __far "SegFar2"   /*Scope of SegFar1 is over */
int __near Var5 = 1 ;         /* Grouped With SegNear2; */
int __far Var6 ;              /* Grouped With SegFar2 */
```

The following is the code generated for the above example

OUTPUT

```
            rseg    SegNear1
_Var1 :
            dw      01h

            rseg    SegFar1
_Var2 :
            dw      01h

            rseg   SegFar1
_Var4 :
            dw      00h

            rseg SegFar2
_Var6 :
            dw      00h

            rseg $$init_info
            dw $$INITTAB_1
            dw $$INITVAR_1
            dw 4
            db seg $$INITTAB_1
            db seg $$INITVAR_1

            rseg   SegNear2TAB
$$INITTAB_1 :
            dw      01h
            dw      01h

            rseg SegNear2
$$INITVAR_1 :
_Var3 :
            ds      02h
_Var5 :
            ds      02h
```

5.17 SEGDEF PRAGMA

Syntax:

a. /PF option specified:

#pragma SEGDEF "segment_name", "segment_type"

b. /PF option not specified:

#pragma SEGDEF "segment_name" "segment_type"

SEGDEF pragma instructs the compiler that segment definition pseudo instructions are output to an assembly language file. The SEGDEF pragma is used when segment symbols that are referenced by __segbase_n, __segbase_f, and __segsize functions, are not defined in the same compilation unit. Therefore, when segment symbols of the same name are defined in the same compilation unit, the compiler ignores the pragma.

If the segment type for segment symbols having same name are different, CCU8 ignores the pragma by issuing a warning message, otherwise pragma is ignored without issuing any warning

Segment name specified with SEGDEF pragma is either default segment, automatically generated by the compiler or defined by the user using the SEGCODE, SEGINTR, SEGINIT, SEGNOINIT, SEGCONST, and SEGNVDATA pragmas

As a segment type, CODE, DATA, TABLE, or NVDATA can be specified. Segment types can be specified either in uppercase or in lowercase characters. When any other segment type is specified, CCU8 ignores the pragma, and issues a warning message to the pragma.

Even if the segment symbol that is defined by the SEGDEF pragma is not referenced within the same compilation unit, the CCU8 outputs segment definition pseudo instructions without issuing a warning message.

The physical segment attribute of the segment definition instruction that is generated by the SEGDEF pragma is ANY. This is because the pragma is needed only to reference a segment symbol. Two examples using SEGDEF pragma are given below.

Example 5.57

INPUT

```
#pragma SEGDEF "SEGRAM" "DATA"
```

OUTPUT

```
SEGRAM  segment data any
```

Example 5.58

INPUT

```
--------module1.c--------------
#pragma Segconst  "SEGMENT1"  // Segment definition
const int siGVar;
--------module2.c--------------
#pragma SEGDEF "SEGMENT1" "table" // Indicates that Segment with name
                                  // "SEGMENT1" has been defined in other
                                  // compilation unit
void fn(void)
{
 unsigned int uisize = __segsize("SEGMENT1"); // Segment symbol is
}                                                    //  referenced
```

Pseudo Segment definition for SEGDEF pragma is generated in one compilation unit module2.c , it is defined in another compilation unit Module1.c

OUTPUT

```
--------module1.asm--------------
 SEGMENT1 segment table 2h #0h  // Actual segment definition


 rseg SEGMENT1
_siGVar :
 dw  00h
--------module2.asm--------------


 SEGMENT1  segment table any   // Segment pseudo definition is generated
                               // for SEGDEF pragma


 l er0,  $$S1
 .
 .
         rseg $$NTABModule2
 $$S1 :
  DW  SIZE  SEGMENT1
```

6. OUTPUT FILES

The different output files with their default extensions are listed below.

TABLE 6.1	
Output File	**Default Extension**
*Assembly Output	.ASM
Source/Error Listing	.LER
**Calltree Listing	-
Preprocessed Output	.I
Function Prototype File	.PRO

* indicates that the assembly file name extension may be changed using /Fa option in the command line.

** indicates that calltree listing file has no default extension.

Command line options to obtain corresponding output file is listed below.

TABLE 6.2	
Output File	**Command Line Option**
*Assembly Output	/Fa
Source/Error Listing	/LE
Calltree Listing	/CT
Preprocessed Output	/LP or /PC
Function Prototype File	/Zg

* indicates that CCU8 generates assembly file with default assembly file name, if /Fa option is not specified in the command line.

6.1 ASSEMBLY OUTPUT

The output file produced by CCU8 is an assembly file which contains UniCORE8 assembly mnemonics.

This section explains the conventions followed by the compiler in generating the output code.

6.1.1 Comment Section

The start of the output assembly file has a comment section. It contains the following information:

1. Compile Options
2. Version Number
3. File Name

6.1.1.1 Compile Options

The compile options specified along with the file name in the command line are listed in a sequence.

Example 6.1

COMMAND LINE

```
C:\>CCU8 /Tmu8 /MS /SS 10000 test.c
```

For the above command line, the compile options are output in the comment section as follows:

OUTPUT

```
;; Compile Options : /Tmu8 /MS /SS 10000
```

6.1.1.2 Version Number

The compiler version in which the source file is compiled, is output in the comment section.

Example 6.2

```
;; Version Number  : Ver.3.00.1
```

6.1.1.3 File Name

The source file name, as specified by the user in the command line, is output in the comment section.

Example 6.3

COMMAND LINE

```
C:\>CCU8 /Tmu8 /MS ..\source\test.c
```

For the above command line, the source file name is output in the comment section as follows:

OUTPUT

```
;; File Name       : ..\source\test.c
```

6.1.2 Assembler Initialization Pseudo-Instructions

This section contains the pseudo-instructions output by CCU8, which are required by RASU8.

6.1.2.1 TYPE INSTRUCTION

The TYPE pseudo-instruction is generated at the beginning of the output. The string specified with /T option is output with this pseudo instruction.

Example 6.4

COMMAND LINE

```
C:\>CCU8 /Tmu8 test.c <CR>
```

For the above command line, the following pseudo-instruction is output in “test.asm”:

OUTPUT

```
type (mu8)
```

6.1.2.2 MODEL PSEUDO-INSTRUCTION

The MODEL pseudo-instruction is used to specify the hardware memory model and the default data memory access in the assembly listing file. One of the following is output based on the hardware memory model:

small	for small memory model
large	for large memory model

One of the following is output based on the default data memory access:

near	for near data access specifier
far	for far data access specifier

Example 6.5

COMMAND LINE

```
C:\>CCU8  /MS  /Tmu8 test.c <CR>
```

For the above command line, the following pseudo-instruction is output in “test.asm”:

OUTPUT

model small, near

Example 6.6

COMMAND LINE

C:\>CCU8 /MS /far /Tmu8 test.c <CR>

For the above command line, the following pseudo instruction is output in “test.asm”:

OUTPUT

model small, far

6.1.2.3 NOFAR INSTRUCTION

The NOFAR pseudo-instruction is generated at the position immediately following the MODEL pseudo instruction. The pseudo instruction is generated when /nofar option is specified in command line.

Example 6.7

COMMAND LINE

C:\>CCU8 /Tmu8 /nofar test.c <CR>

For the above command line, the following pseudo-instruction is output in “test.asm”:

OUTPUT

nofar

6.1.2.4 SEGMENT DEFINITION PSEUDO INSTRUCTION

This section contains the definitions of all the relocatable segments, that have been used in the assembly output file.
In RLU8 Ver.1.50, the feature which does not link the function/table which is not referred to has been added. In CCU8 Ver.3.30, the generated segment name in having /Zc command line option or not are different.

Segment Name	Description
$$*funcname*$*filename*	Specifies segment definition for near or far function. (Only /Zc command line option not specified.)
$$NCOD*filename*	Specifies segment definition for near function. (Only /Zc command line option specified.)
$$FCOD*filename*	Specifies segment definition for far function. (Only /Zc command line option specified.)
$$INTERRUPTCODE	Specifies segment definition for interrupt functions. This segment resides in code segment #0. (Only /Zc command line option specified.)
$$TAB*constname*$*filename*	Specifies segment definition for const qualified near or far variable. (Only /Zc command line option not specified.)
$$NTAB*filename*	Specifies segment definition for const qualified near variable. (Only /Zc command line option specified.)
$$FTAB*filename*	Specifies segment definition for const qualified far variable. (Only /Zc command line option specified.)
$$NVAR*filename*	Specifies segment definition for uninitialized near static variable (global/local).
$$FVAR*filename*	Specifies segment definition for uninitialized far static variable (global/local).
$$NINITVAR	Specifies segment definition for initialized near variable without the const qualifier.
$$NINITTAB	Specifies initialization values for initialized near variable without the const qualifier.
$$FINITVAR*filename*	Specifies segment definition for initialized far variable without the const qualifier.
$$FINITTAB*filename*	Specifies initialization values for initialized far variable without the const qualifier.
$$init_info	Specifies initialization information table for far variable without the const modifier.
$$NNVDATA*filename*	Specifies segment definitions for near variable specified in nvdata pragma.

$$FNVDATA*filename*	Specifies segment definitions for far variable specified in nvdata pragma.
$$content_of_init	Specifies segment definition for initialised variable without the const qualifier that is specified in absolute pragma.

Each segment definition contains the name of the segment and the properties associated with that segment.

Example 6.8

$$NCOD*file* segment code 2h #0h

The above segment definition indicates that, the segment '$$NCOD*file*' is allocated in 0^{th} physical code segment of boundary 2.

6.1.3 Procedure Section

This section contains the assembly instructions and assembly directives, generated for all the functions defined in the source file.

The contents of this section can be further classified as follows:

1. relocatable segment definition
2. function name label
3. C source level debug information
   - CGLOBAL directive
   - CSGLOBAL directive
   - CARGUMENT directive
   - CLOCAL directive
   - CSLOCAL directive
   - CLABEL directive
   - CSTRUCTTAG directive
   - CSTRUCTMEM directive
   - CLINE directive
   - CBLOCK directive
   - CBLOCKEND directive
   - CFILE directive
   - CUNIONTAG directive
   - CUNIONMEM directive
   - CENUMTAG
   - CENUMMEM directive
   - CTYPEDEF directive

- CFUNCTION directive
- CFUNCTIONEND directive
- C source line

4. assembly instructions for each statement

6.1.3.1 RELOCATABLE SEGMENT DEFINITION

A function is placed in a segment which is determined by the type of the function. To specify a function in a particular segment, ‘rseg’ pseudo instruction is used. For example, to specify that the function should be allocated in ‘NCOD*file*’ segment, the following is output:

```
rseg NCOD file
```

Example 6.9

INPUT

```
/* um609.c*/
void fn ()
{
}
```

OUTPUT

```
rseg $$NCODum609
```

All the functions are output in the assembly file, in the order they appear in the source file. If the segment in which the current function is to be allocated is same as that for the previous function, ‘rseg’ directive is not output.

6.1.3.2 FUNCTION NAME LABEL

Each beginning of a function is marked by the function name followed by a colon (:). This label indicates that the assembly instructions following this label are part of this function code. The function name is preceded by a ‘_’.

Example 6.10

INPUT

```
int func ()
{
}
```

The function name label is output as follows for the function ‘func’ in the above example:

OUTPUT

```
_func :
```

6.1.3.3 ‘C’ SOURCE LEVEL DEBUG INFORMATION

6.1.3.3.1 CGLOBAL directive

This directive is output for each global variable definitions.

SYNTAX

CGLOBAL usg_typ attrib size “global_name” hierarchy

6.1.3.3.2 CSGLOBAL directive

This directive is output for each static global variable definitions.

SYNTAX

CSGLOBAL attrib size “static_global_name” hierarchy

6.1.3.3.3 CARGUMENT directive

This directive is output for each argument variable definitions.

SYNTAX

CARGUMENT attrib size offset “argument_name” hierarchy

6.1.3.3.4 CLOCAL directive

This directive is output for each local variable definitions.

SYNTAX

CLOCAL attrib size offset block_id “local_name” hierarchy

6.1.3.3.5 CSLOCAL directive

This directive is output for each static local variable definitions.

SYNTAX

CSLOCAL attrib size alias_no block_id “static_local_name” hierarchy

6.1.3.3.6 CLABEL directive

This directive is output for each label definitions.

SYNTAX

CLABEL label_no "label_name"

6.1.3.3.7 CSTRUCTTAG directive

This directive is output for each structure variable definitions.

SYNTAX

CSTRUCTTAG fn_id block_id su_id tot_mem tot_size “tag_name”

6.1.3.3.8 CSTRUCTMEM directive

This directive is output for each structure member variable definitions.

SYNTAX

CSTRUCTMEM attrib size offset “member_name” hierarchy

6.1.3.3.9 CLINE directive

CLINE directive is output for each executable statement, for which assembly instructions have been generated.

SYNTAX

CLINE line_atr line_no start_column end_column

The CLINE directive is followed by the line attribute, line number, start column number and end column number of the ‘C’ statement in the source file. The line attribute field will be 0 for the first line of a ‘C’ source code statement. CCU8 will use this field to distinguish among the multiple CLINE directives resulting from the optimizer’s splitting of a single source code line into several parts.

6.1.3.3.10 CBLOCK/CBLOCKEND directives

For each ‘{’ in the source file, a CBLOCK directive is output. Along with CBLOCK the function id, block number and the line number is also output. Similarly, for each ‘}’ in the source file, a CBLOCKEND directives is output along with the function id, block number and the corresponding line number (specified in CBLOCK directive).

SYNTAX

CBLOCK fn_id block_id line_no
CBLOCKEND fn_id block_id line_no

6.1.3.3.11 CFILE directive

To distinguish the output of include files and source file, CFILE directive is output. CFILE directive is followed by the file number. On encountering an include file this directive is output along with file number associated with the include file.

SYNTAX

CFILE file_id total_line “filename”

6.1.3.3.12 CUNIONTAG directive

This directive is output for each union variable definitions.

SYNTAX

CUNIONTAG fn_id block_id un_id tot_mem tot_size “tag_name”

6.1.3.3.13 CUNIONMEM directive

This directive is output for each union member variable definitions.

SYNTAX

CUNIONMEM attrib size “member_name” hierarchy

6.1.3.3.14 CENUMTAG directive

This directive is output for each enum variable definitions.

SYNTAX

CENUMTAG fn_id block_id enum_id tot_mem “tag_name”

6.1.3.3.15 CENUMMEM directive

This directive is output for each enum member variable definitions.

SYNTAX

CENUMMEM value “member_name”

6.1.3.3.16 CTYPEDEF directive

This directive is output for each typedef variable definitions.

SYNTAX

CTYPEDEF fn_id block_id attrib “type_name” hierarchy

6.1.3.3.17 CFUNCTION directive

Each function name label is preceded by ‘CFUNCTION’ directive. Each CFUNCTION directive has a function number associated with it, which is output along with the directive.

SYNTAX

CFUNCTION fn_id
CFUNCTIONEND fn_id

6.1.3.3.18 ‘C’ source line

For each executable line for which assembly instructions are output, the corresponding ‘C’ statement is output as comments.

Example 6.11

INPUT

```
a = fn ();
```

For the above ‘C’ statement, the ‘C’ source line is output in the assembly file as follows:

OUTPUT

```
;; a = fn () ;
```

6.1.3.4 ASSEMBLY INSTRUCTIONS

One or more assembly instructions are generated for a ‘C’ statement.

Example 6.12

INPUT

```
int b, c ;
void
fn ()
{
        b = fun1 () ;
        c += b ;
}
```

OUTPUT

```
…
;;      b = fun1 () ;
        bl      _fun1
        st      er0,    NEAR _b

;;      c += b ;
        l       er0,    NEAR _c
        l       er2,    NEAR _b
        add     er0,    er2
        st      er0,    NEAR _c
…
```

6.1.4 Symbol Declaration Section

This section contains the symbol declarations for different types of variables specified in the source file.

The three types of symbol declarations are as follows:

1. comm
2. public
3. extrn

Uninitialized global data variables, which are not specified in pragmas, are output using the 'comm' pseudo instruction.

Example 6.13

INPUT

```
long a;
```

OUTPUT

```
_a comm data 04h #00h
```

In the above example, 'a' is assigned a location in 0^{th} data segment with size 4 bytes.

Initialized global data variables are output using 'public' pseudo instruction.

Example 6.14

INPUT

```
int a = 7 ;
```

OUTPUT

```
public _a
```

In the above example, the variable 'a' is output as public.

A function which has been called but whose body is not defined in the current file, is output as '**extern**'. Similarly, variables that have been declared as **'extern'** in source are also output as 'extrn'.

Example 6.15

INPUT

```
extern int a ;

main ()
{
        a = 1 ;

        fn () ;
}
```

OUTPUT

```
.....
extrn code near : _fn
.....
extrn data near : _a
```

In the above example, the body of the function ‘fn’ is not defined and therefore, it is output as ‘extrn’. Also, ‘a’ has been declared as ‘**extern’**. Therefore, no storage is allocated and output as ‘extrn’.

The memory initialization pseudo instructions DW and DB and memory allocation pseudo instruction DS are used to output the initialized global data variables. CCU8 follows similar methods to output static, non-static and aggregate (array, structure/union) initialized global data variables.

Example 6.16

INPUT

```
long var = 10 ;
const int cint = 20 ;
```

OUTPUT

```
        rseg $$NINITTAB
        dw      0ah
        dw      00h

        rseg $$NTABum616
_cint :
        dw      014h

        rseg $$NINITVAR
_var :
        ds      04h
```

In the above example, 4 bytes are allocated in '$$NINITVAR' data segment using DS pseudo instruction. The initial value is output in '$$NINITTAB' using DW pseudo instructions. Similarly, for the **const** variable 'cint' DW pseudo instruction is used to allocate and initialize memory.

Initialization of global data and **static** variables are performed by allocating memory for these variables in a RAM segment and defining those initial values in a ROM segment. Startup code copies these initial values from the ROM segment to the RAM segment before the function "main" is invoked.

A sample output of an assembly file is given below:

Example 6.17

INPUT

```
int     a, b ;
int     c = 10 ;

void fn ( void )
{
        b = fn1 () ;

        a = b * c ;

        return ;
}
```

OUTPUT

```
;; Compile Options : No command line options
;; Version Number  : <version>
;; File Name       : usr617.int

        type (mu8)
        model small, near
        $$NINITVAR segment data 2h #0h
        $$NINITTAB segment table 2h any
        $$NCODusr617 segment code 2h #0h
CFILE 0000H 00000009H "usr617.c"

        rseg $$NCODusr617

_fn     :
;;{
        push    lr

;;      b = fn1 () ;
        bl      _fn1
        st      er0,    NEAR _b

;;      a = b * c ;
```

```
                l       er2,    NEAR _c
                bl      __imulu8sw
                st      er0,    NEAR _a

;;}
                pop     pc

                extrn code near : _fn1
                public _c
                public _fn
                _a comm data 02h #00h
                _b comm data 02h #00h
                extrn code near : _main

                rseg $$NINITTAB
                dw      0ah

                rseg $$NINITVAR
_c :
                ds      02h
                extrn code : __imulu8sw

                end
```

6.2 ERROR LISTING

Source listings are helpful in debugging programs as they are being developed. These listings are also useful for documenting the structure of finished programs.

The source listing contains the numbered source code lines of each function in the source file, along with diagnostic messages that were generated. Any error or warning messages issued during compilation appear in the listing after the line that caused the error, as shown in the following example:

Example 6.18

INPUT

```
int  a ;
int b ;

void fn ()
{
        output_fn () ;

        if ( a == b [1] )
                return a ;
}
```

The following list file is generated when the above program “um618.c” is compiled in /LE /Tmu8 options:

OUTPUT

```
                                                    Page :              1
                                                    Date : 09-22-2000
                                                    Time :   14:32:03
CCU8 C Compiler Ver.1.00, Source List
Source File : um618.c

Line #  Source Line

    1 int  a ;
    2 int b ;
    3
    4 void fn ()
    5 {
    6       output_fn () ;
    7       if ( a == b [1] )
***** um618.c(7) : Error : E5003 : Subscript on non array
    8             return a ;
    9 }
***** um618.c(8) : Error : E5038 : Void function returning value
   10


Error(s)      : 2
Warning(s)  : 0
```

If the source file compiles without an error or fatal error, then the list file is generated without errors. The following example shows a complete source listing without errors.

Example 6.19

INPUT

```
int a ;
int b ;

void
begin ( x, y)
int x ;
int y ;
{
     function () ;
     end ( x, y) ;
     return ;
}

int
end ( x, y)
```

```
int x ;
int y ;
{
    int z ;
    z = function1 () ;
    function2 () ;
    z += x + y ;
    return (z) ;
}
```

OUTPUT

```
                                                    Page :         1
                                                    Date : 02-09-2001
                                                    Time :   11:51:17
CCU8 C Compiler Ver.1.00, Source List
Source File : um619.c


 Line #  Source Line

     1 int a ;
     2 int b ;
     3
     4 void
     5 begin ( x, y)
     6 int x ;
     7 int y ;
     8 {
     9 function () ;
    10 end ( x, y) ;
    11 return ;
    12 }
    13
    14 int
    15 end ( x, y)
    16 int x ;
    17 int y ;
    18 {
    19 int z ;
    20 z = function1 () ;
    21 function2 () ;
    22 z += x + y ;
    23 return (z) ;
    24 }
    25

Error(s)     :  0
Warning(s)   :  0
                                                    Page :         2
                                                    Date : 02-09-2001
```

Time : 11:51:17
CCU8 C Compiler Ver.1.00, Source List
Source File : um619.c

STACK INFORMATION

FUNCTION	LOCALS	CONTEXT	OTHERS	TOTAL
_begin	0	6	0	6
_end	2	8	0	10

6.3 CALLTREE LISTING

The calltree listing file produces an indented listing showing the procedure names at the left margin. Calls are shown indented three spaces per level.

If a path has already been viewed, it is shown as ellipsis (...). A recursive call is shown as an asterisk (*). If a call to an undefined procedure is made, a question mark(?) appears.

Example 6.20

INPUT

```
void
fn ()
{
}

void
fn1 ()
{
        fn () ;
        fn1 () ;
        fn2 () ;
}
```

For the above source file “um620.c”, the calltree listing generated by CCU8 is shown below.

```
CCU8 C Compiler, Ver.1.00, Calltree Listing

Source File : um620.c

fn
fn1
|  fn...
|  fn1*
|  fn2?
```

In the above example, ellipsis follows function “fn” because calltree for function “fn” is listed previously. An asterisk follows function “fn1” because it is called recursively. A question mark follows “fn2” because definition of function “fn2” was not encountered prior to that function call.

When more than one source file is specified for compilation, the calltree listing of each source file is output in the same calltree file. However, the calltree information of one source file is not carried to another source file.

Example 6.21

INPUT

```
/* um621a.c */
void fn ()
{
        fn () ;
}

/* um621b.c */
void fn1 ()
{
        fn () ;
}
```

In the above code, the function “fn” is defined in source file “um621a.c” and function “fn1” in “um621b.c” calls function “fn”.

OUTPUT

```
CCU8 C Compiler, Ver.1.00, Calltree Listing

Source File : um621a.c

fn
|  fn*

Source File : um621b.c

fn1
|  fn?
```

In the calltree listing of function “fn1”, a question mark follows “fn” since function “fn” was not defined in source file “um621b.c”.

6.4 FUNCTION PROTOTYPE LISTING

CCU8 creates prototype list file with an extension ".pro" and the base name derived from the source file when /Zg option is specified in the command line. The function prototype file contains prototypes for the functions defined in the source file.

Example 6.22

INPUT

```
/* um622.c */
int fn1 (long a, char *cptr)
{
        fn3 () ;
}
static void fn2 (int *iptr, char c)
{
}
```

CCU8 creates "um622.pro" when the above file is compiled in /Zg option. This function prototype file contains the following function prototypes:

```
extern  int fn1(long a,char *cptr);
static  void fn2(int *iptr,char c);
```

A function defined in the source file is declared as **extern** function in function prototype file unless it is qualified with **static**.

Example 6.23

INPUT

```
/* um623.c */
void fn1 (int *iptr, char c)
{
}
static void fn2 (int *iptr, char c)
{
}
extern void fn3 (int *iptr, char c)
{
}
```

CCU8 outputs following prototypes in file “um623.pro” when the above file is compiled in /Zg option:

```
extern  void fn1(int *iptr,char c);
static  void fn2(int *iptr,char c);
extern  void fn3(int *iptr,char c);
```

A function defined without any argument is declared as function taking **void** as its argument in function prototype file.

Example 6.24

INPUT

```
/* um624.c */
void fn1 ()
{
}
```

CCU8 outputs following prototype in file “um624.pro” when the above file is compiled in /Zg option:

```
extern  void fn1(void);
```

A function defined without return type is declared as function returning **int** in function prototype file.

Example 6.25

INPUT

```
/* um625.c */
fn1 ()
{
}
```

CCU8 outputs following prototype in file “um625.pro” when the above file is compiled in /Zg option:

```
extern  int fn1(void);
```

7. OPTIMIZATIONS

CCU8 performs a variety of optimizations that reduce the storage space or execution time required for a program. This is achieved by eliminating unnecessary instructions and rearranging code.

CCU8 performs optimizations of the following types :

1. It modifies or moves sections of code so that fewer and/or faster instructions are used.
2. It eliminates sections of code that are redundant or unused.

CCU8 performs all optimizations by default. The optimization options /Od, /Ol, /Oa, /Og, /Ot and /Om may be used to exercise greater control over the optimizations performed.

7.1 GLOBAL OPTIMIZATIONS

Global optimizations are those that are performed across different basic blocks of code. (A basic block corresponds to a sequence of executable statements through which control flows from the first statement to the last statement, sequentially).

The following optimizations are classified as global optimizations :

1. Constant propagation
2. Constant folding
3. Common sub-expression elimination
4. Code sinking
5. Code hoisting

The above optimizations may be enabled or disabled, using /Og option.

7.1.1 Constant Propagation

Variables used in expressions are replaced by their constant values.

Example 7.1

INPUT

```
int a, b, x, y, m, n ;
```

```
const_propagate ()
{
      a = 45 ;
      if ( b < x)
      {
          m = a  ;          /* changed to m = 45 */
      }
      y = a + m ;           /* changed to y = 45 + m */
}
```

Assembly code generated by CCU8 for the above function ‘const_propagate’ is shown below:

OUTPUT

```
_const_propagate :

;;          a = 45 ;
            mov     er0,    #45
            st      er0,    NEAR _a

;;          if ( b < x)
            l       er0,    NEAR _b
            l       er2,    NEAR _x
            cmp     er0,    er2
            bges    _$L1

;;                  m = a  ;            /* changed to m = 45 */
            mov     er0,    #45
            st      er0,    NEAR _m

;;          }
_$L1 :

;;          y = a + m ;         /* changed to y = 45 + m */
            l       er0,    NEAR _m
            add     er0,    #45
            st      er0,    NEAR _y

;;}
            rt
```

7.1.2 Constant Folding

The resultant constant expressions are computed at compile time and the computed result is used in the expression.

Example 7.2

INPUT

```
int a, b, x, y, m, n ;

const_folding ()
{
      a = 45 ;

      if ( b < x)
      {
          m = a + 20 ;      /* changed to m = 65 */
      }

      y = a + m ;           /* changed to y = 45 + m */
}
```

Assembly code generated by CCU8 for the above function ‘const_folding’ is shown below:

OUTPUT

```
_const_folding :

;;        a = 45 ;
          mov     er0,      #45
          st      er0,      NEAR _a

;;        if ( b < x)
          l       er0,      NEAR _b
          l       er2,      NEAR _x
          cmp     er0,      er2
          bges    _$L1

;;                m = a + 20 ;      /* changed to m = 65 */
          mov     r0,       #041h
          mov     r1,       #00h
          st      er0,      NEAR _m

;;        }
_$L1 :

;;        y = a + m ;       /* changed to y = 45 + m */
          l       er0,      NEAR _m
          add     er0,      #45
          st      er0,      NEAR _y

;;}
          rt
```

7.1.3 Common Sub-Expression Elimination

Sub-expressions that are repeated more than once are eliminated. These are replaced by a temporary that hold the result of a single evaluation.

Example 7.3

INPUT

```
int a, b, x, y, m, n ;

common_sub_exp ()
{
      x = a + b ;              /* a + b is also assigned to a temporary */

      if (a < b)
          m = a + b + y ;      /* a + b is replaced by the temporary */
      else
          n = (a + b) >> 4 ;   /* a + b is replaced by the temporary */
}
```

Assembly code generated by CCU8 for the above function ‘common_sub_exp’ is shown below:

OUTPUT

```
_common_sub_exp         :

            push    er4

;;          x = a + b ;         /* a + b is also assigned to a temporary */
            l       er0,    NEAR _a
            l       er2,    NEAR _b
            add     er0,    er2
            st      er0,    NEAR _x
            mov     er4,    er0

;;          if (a < b)
            l       er0,    NEAR _a
            cmp     er0,    er2
            bges    _$L1

;;                  m = a + b + y ;     /* a + b is replaced by the temporary */
            mov     er0,    er4
            l       er2,    NEAR _y
            add     er0,    er2
            st      er0,    NEAR _m

;;          else
            bal     _$L3
_$L1 :
```

```
;;                  n = (a + b) >> 4 ; /* a + b is replaced by the temporary */
            mov     er0,    er4
            srlc    r0,     #04h
            sra     r1,     #04h
            st      er0,    NEAR _n
_$L3 :

;;}
            pop     er4
            rt
```

7.1.4 Code Sinking

If control passes to a single point, after executing same sequence of statements along different paths, the statements are sinked (moved down) to the single common point. The unnecessary copies of statements are removed.

Example 7.4

INPUT

```
int a, b, e, x, y, z, m, n ;

sink ()
{
    if ( a == b )
    {
        func () ;
        m = e + 25 ;     /* two statements are sinked */
        return (e) ;
    }

    x = y + z ;
    m = e + 25 ;         /* two statements are removed */

    return (e) ;
}
```

Assembly code generated by CCU8 for the above function ‘sink’ is shown below :

OUTPUT

```
_sink    :
;;{
            push    lr

;;if ( a == b )
            l       er0,    NEAR _a
            l       er2,    NEAR _b
            cmp     er0,    er2
            bne     _$L1
```

```
;;func () ;
        bl      _func

;;}
_$L0 :
        l       er0,    NEAR _e
        add     er0,    #25
        st      er0,    NEAR _m
        l       er0,    NEAR _e
        pop     pc

;;}
_$L1 :

;;x = y + z ;
        l       er0,    NEAR _y
        l       er2,    NEAR _z
        add     er0,    er2
        st      er0,    NEAR _x

;;return (e) ;
        bal     _$L0
```

7.1.5 Code Hoisting

This is similar to code sinking, but the direction of code movement is reversed. If control passes from a single point, and same sequence of statements are executed along different paths, the statements are hoisted (moved up) to the single common point. The unnecessary copies of statements are removed.

Example 7.5

INPUT

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

hoist ()
{
    if ( a == b )
    {
        m = x + y ;         /* statement hoisted */
        x = z ;
    }
    else
    {
        m = x + y ;         /* statement removed */
        fn1 () ;
    }
```

```
}
```

Assembly code generated by CCU8 for the above function ‘hoist’ is shown below:

OUTPUT

```
_hoist  :

;;{
        l       er0,    NEAR _x
        l       er2,    NEAR _y
        add     er0,    er2
        st      er0,    NEAR _m

;;      if ( a == b )
        l       er0,    NEAR _a
        l       er2,    NEAR _b
        cmp     er0,    er2
        bne     _$L1

;;              x = z ;
        l       er0,    NEAR _z
        st      er0,    NEAR _x

;;      else
        rt
_$L1 :

;;              fn1 () ;
        b       _fn1
```

7.2 LOOP OPTIMIZATIONS

Loop optimizations are those that are performed on statements within loops.

The following optimizations are classified as loop optimizations:

1. Loop invariant code motion
2. Loop variant code motion
3. Induction variable elimination
4. Strength reduction
5. Loop unrolling

The above optimizations may be enabled or disabled using the /Ol option.

7.2.1 Loop Invariant Code Motion

Expressions whose values do not change through each execution of a loop are termed as invariant expressions. Such expressions are detected and moved to a position outside the loop, so that they are evaluated only once.

Example 7.6

INPUT

```
unsigned int x, m, n, o, p, r, i, y [10] ;

loop_invar ()
{
    do
    {
        p = n / o ;          /* moved outside the loop */
        x = m * r + i ;      /* sub-expression m * r is moved outside the loop */
        y [i] += x ;
        i ++ ;

    } while ( x < i ) ;
}
```

Assembly code generated by CCU8 for the above function ‘loop_invar’ is shown below:

OUTPUT

```
_loop_invar     :

;;{

        push    lr
        push    xr4
        l       er0,    NEAR _n
        l       er2,    NEAR _o
        bl      __uidivu8sw
        st      er0,    NEAR _p
        l       er0,    NEAR _m
        l       er2,    NEAR _r
        bl      __imulu8sw
        mov     er6,    er0

;;      do
_$L3 :

;;              x = m * r + i ;     /* sub-expression m * r is moved outside the loop */
        mov     er0,    er6
        l       er2,    NEAR _i
        add     er0,    er2
        st      er0,    NEAR _x
```

```
;;                  y [i] += x ;
        add         er2,        er2
        l           er4,        NEAR _y[er2]
        add         er4,        er0
        st          er4,        NEAR _y[er2]

;;                  i ++ ;
        l           er0,        NEAR _i
        add         er0,        #1
        st          er0,        NEAR _i

;;      } while ( x < i ) ;
        l           er0,        NEAR _x
        l           er2,        NEAR _i
        cmp         er0,        er2
        blt         _$L3

;;}
        pop         xr4
        pop         pc
```

7.2.2 Loop Variant Code Motion

Expressions whose values change by constant step value through each execution of a loop are termed as variant expressions. Such expressions are detected and moved to a position outside the loop, so that they are evaluated once with the final values of the variables (values at loop exit).

Example 7.7

INPUT

```
int i, a ;

loop_variant_code_motion ()
{
    for ( i = 1 ; i < 11 ; i ++ )
        a += i ;
}
```

The above loop is replaced by

```
a += 55 ;
i = 11 ;
```

Assembly code generated by CCU8 for the above function loop_variant_code_motion is shown below:

OUTPUT

```
_loop_variant_code_motion        :

;;{
        l       er0,    NEAR _a
        add     er0,    #55
        st      er0,    NEAR _a
        mov     er0,    #11
        st      er0,    NEAR _i

;;}
        rt
```

7.2.3 Induction Variable Elimination

An induction variable is one whose value changes by a function of another variable or constant, within a loop. When two or more induction variables are present, variables which are a linear function of another variable are eliminated and all uses of the eliminated variable are replaced by a function of the other variable. Eliminated variables are initialized to their final values, if necessary.

Example 7.8

INPUT

```
char a [10] ;
int i, j ;

induction_var_elim ()
{
    for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
    {   /* i, j are induction variables */
        a [ i ] = j + 3 ;
    }
}
```

The above loop is transformed to

```
j = 10 ;

for ( i = 0 ; i < 10 ; i ++)
{
    a [ i ] = i + 3 ;          /* j is replaced by i */
}
```

Assembly code generated by CCU8 for the above function ‘induction_var_elim’ is shown below:

OUTPUT

```
_induction_var_elim        :

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
            mov     er0,    #0
            st      er0,    NEAR _i
            mov     er0,    #10
            st      er0,    NEAR _j

_$L3 :

;;                  a [ i ] = j + 3 ;
            l       r0,     NEAR _i
            add     r0,     #03h
            l       er2,    NEAR _i
            st      r0,     _a[er2]

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
            mov     er0,    er2
            add     er0,    #1
            st      er0,    NEAR _i

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
            cmp     r0,     #0ah
            cmpc    r1,     #00h
            blts    _$L3

;;}
            rt
```

7.2.4 Strength Reduction

Expressions, in loops, that use costly operations are modified to use cheaper operations.

Example 7.9

INPUT

```
int a [10] ;
int i ,i1;

strength_reduction ()
{
    for ( i = 0, i1 = 0; i < 10 ;  i1++, i +=2 )
    {
        a [i1] = 0 ;        /* for accessing i1th element of the ‘a’, multiplication by 2 */
                            /* is necessary, because ‘a’ is an array of ‘int’ */
    }
}
```

The above loop is transformed to

```
for ( i = 0, i1 = 0, temp = 0;  i < 10 ; temp += 2, i1 ++, i+= 2)
{
      * ( a + temp ) = 0 ;          /* variable i1 varies similar to that of loop */
                                    /* control variable i. So strength reduction is */
                                    /* performed on the variable i1. multiplication */
                                    /* inside the loop is removed by introducing  */
                                    /* temporary variable (incremented by 2 */
                                    /* instead  of 1 because 'a' is an array of int).*/
}
```

Assembly code generated by CCU8 for the above function 'strength_reduction' is shown below:

OUTPUT

```
_strength_reduction         :

;;{
            push      er4

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
            mov       er0,      #0
            st        er0,      NEAR _i
            st        er0,      NEAR _i1
            mov       er4,      er0
_$L3 :

;;        a[i1] = 0;            /* for accessing ith element of the 'a', multiplication by 2 */
            mov       er2,      #0
            st        er2,      NEAR _a[er4]

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
            l         er0,      NEAR _i1
            add       er0,      #1
            st        er0,      NEAR _i1
            add       er4,      #2
            l         er0,      NEAR _i
            add       er0,      #2
            st        er0,      NEAR _i

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
            cmp       r0,       #0ah
            cmpc      r1,       #00h
            blts      _$L3

;;}
            pop       er4
            rt
```

7.2.5 Loop Unrolling

The body of a loop which would execute a constant number of times, is expanded that many number of times, if feasible. The loop control statements are removed.

Example 7.10

INPUT

```
loop_unroll ()
{
    int i ;
    for ( i = 0 ; i < 2 ; i ++)
        function () ;
}
```

The above loop is transformed to

```
function () ;
function () ;
```

Assembly code generated by CCU8 for the above function ‘loop_unroll’ is shown below:

OUTPUT

```
_loop_unroll    :
;;{
        push    lr

;;{
        bl      _function
        bl      _function

;;}
        pop     pc
```

7.3 OTHER OPTIMIZATIONS

The other optimizations performed include :

1. Dead code elimination
2. Dead variable elimination
3. Optimization using algebraic identities
4. Optimizing jumps
5. Replacing ‘const’ variables with immediate value

7.3.1 Dead Code Elimination

Parts of code that will never be executed are referred to as ‘dead’ code. These can be statements that could be detected as dead, by looking at the input source program, or those that could be detected because of prior optimizations such as constant propagation.

Example 7.11

INPUT

```
int a, p, q, r ;

dead_code ()
{
    a = 10 ;
    r = p + q ;

    if ( a < 10)    /* if statement removed */
        fn1 () ;    /* statement removed */
}
```

Assembly code generated by CCU8 for the above function ‘dead_code’ is shown below:

OUTPUT

```
_dead_code      :

;;      a = 10 ;
        mov     er0,    #10
        st      er0,    NEAR _a

;;      r = p + q ;
        l       er0,    NEAR _p
        l       er2,    NEAR _q
        add     er0,    er2
        st      er0,    NEAR _r

;;}
        rt
```

7.3.2 Dead Variable Elimination

Variables are assigned values by expressions. The values of some variables may not be used later in the program. Such variables are referred to as ‘dead variables’. These dead variables are detected and removed. Dead variables also include variables, that are assigned values, before a previously assigned value is used. The unnecessary assignment is removed.

Example 7.12

INPUT

```
int x, m, n, r, p, q ;

dead_var ()
{
    int l ;

    x = m + n ;      /* statement removed */
    r = p * q ;
    x = r >> 2 ;
    l = x + r ;      /* statement is removed variable l is a dead variable */
}
```

Assembly code generated by CCU8 for the above function ‘dead_var’ is shown below:

OUTPUT

```
_dead_var       :
;;{
        push    lr

;;      r = p * q ;
        l       er0,    NEAR _p
        l       er2,    NEAR _q
        bl      __imulu8sw
        st      er0,    NEAR _r

;;      x = r >> 2 ;
        srlc    r0,     #02h
        sra     r1,     #02h
        st      er0,    NEAR _x

;;}
        pop     pc
```

7.3.3 Optimization using Algebraic Identities

Expressions that conform to algebraic laws are modified, so that unnecessary operations are eliminated.

Example 7.13

INPUT

```
int a, x, b, c ;
alg_transfer ()
{
        a = x + ( b - c ) ;
}
```

Assembly code generated by CCU8 for the above function ‘alg_transfer’ is shown below

OUTPUT

```
_alg_transfer      :

;;          a = x + ( b - c ) ;
            l       er0,    NEAR _b
            l       er2,    NEAR _c
            sub     r0,             r2
            subc    r1,             r3
            l       er2,    NEAR _x
            add     er0,    er2
            st      er0,    NEAR _a
;;}
            rt
```

7.3.4 Optimizing Jumps

Blocks of code are rearranged to minimize use of jump instructions. Jump instructions that jump to jump instructions are modified to reduce the number of jumps executed.

Example 7.14

```
LABEL2 :
    goto LABEL1;        /* LABEL1 is replaced by LABEL2 */
LABEL1 :
    goto LABEL2;
```

7.3.5 Replacing ‘const’ variables with immediate value

This optimization is performed by replacing variables that are qualified by const with immediate values. This optimization becomes effective when optimization other than /Od option is specified.

Example 7.15

INPUT

```
const char c = 10 ;            /* Handled using the immediate value */
static const char sc = 20 ;    /* Handled using the immediate value */

char x , y ;
void Const_Repl_Fun ()
{
        x = c ;                /* The value of ‘c’ is directly moved to x */
        y = sc ;               /* The value of ‘sc’ is directly moved to y */
}
```

Assembly code generated by CCU8 for the above function ‘Const_Repl_Fun’ is shown below

OUTPUT

```
_Const_Repl_Fun         :

;;{

;;   x = c ;
        mov     r0,     #0ah
        st      r0,     NEAR _x

;;   y = sc ;
        mov     r0,     #014h
        st      r0,     NEAR _y
;;}
        rt
```

7.4 PEEPHOLE OPTIMIZATIONS

Peephole optimizations are performed on the output assembly language instructions.

These optimizations include :

1. Removal of redundant transfer instructions
2. Optimizing relative jumps
3. Tail recursion

7.4.1 Removal Of Redundant Transfer Instructions

The generated assembly instructions are scanned for unnecessary transfers to and from registers.

Example 7.16

```
l       r0,     #20h
st      r0,     _one
l       r0,     #20h        ; this instruction is removed
st      r0,     _two
```

7.4.2 Optimizing Relative Jumps

Relative jump instructions whose targets exceed the allowed range are replaced by pairs of conditional and unconditional jump instructions. Sequential pairs of conditional and unconditional jumps are replaced by a single conditional jump instruction.

Example 7.17

```
        bne _$L2
        bal  _$L1
_$L2 :
:
:
_$L1 :
```

The above instructions are replaced by

```
        beq _$L1
        :
        :
_$L1 :
```

7.4.3 Tail Call Optimization

The generated assembly instructions are scanned for a function call before return instruction and converted to absolute jump.

Example 7.18

```
bl      _fn1
rt
```

The above instructions are replaced by

```
b       _fn1
```

7.5 LOCAL OPTIMIZATIONS

These are optimizations that are performed within a basic block :

1. Constant propagation
2. Constant folding
3. Common subexpression elimination
4. Use of basic algebraic identities
5. Algebraic transformation
6. Copy propagation

These optimizations, within a basic block, are not dependent on any optimization option. These are always enabled.

7.5.1 Constant Propagation

Variables used in expressions are analyzed and changed to constants if they can be changed.

Example 7.19

INPUT

```
int c, d ;

local_constant_prop ()
{
    c = 30 ;
    d = c ;        /* instead of c, 30 is assigned to d */
}
```

Assembly code generated by CCU8 for the above function ‘local_constant_prop’ is shown below

OUTPUT

```
_local_constant_prop      :

;;        c = 30 ;
          mov     er0,    #30
          st      er0,    NEAR _c

;;        d = c ;   /* instead of c, 30 is assigned to d */
```

```
        st      er0,    NEAR _d

;;}
        rt
```

7.5.2 Constant Folding

The result of the expression is analysed for constant and computed at compile time if possible.

Example 7.20

INPUT

```
int  d ;

local_constant_folding ()
{
    d = 30 + 20 ;          /* the resultant value 50 computed at */
                           /* compile time  is assigned to 'd' */
}
```

Assembly code generated by CCU8 for the above function 'local_constant_folding' is shown below

OUTPUT

```
_local_constant_folding    :

;;{

;;      d = 30 + 20 ;              /* the resultant value 50 computed at */
        mov     er0,    #50        /* compile time is assigned */
        st      er0,    NEAR _d

;;}
        rt
```

7.5.3 Common Sub-Expression Elimination

Code containing repeated sub-expressions is modified, so that the sub-expressions are evaluated only once.

Example 7.21

INPUT

```
unsigned int a, b, c, d, x, y ;

local_cse ()
{
```

```
    a = b + c * d ;         /* c *d is evaluated and assigned to a temporary */
    x = c * d / y ;         /* value of c * stored in the temporary is used not evaluated again */
}
```

Assembly code generated by CCU8 for the above function ‘local_cse’ is shown below:

OUTPUT

```
_local_cse      :

        push    lr
        push    er4

;;      a = b + c * d ;     /* c *d is evaluated and assigned to a temporary */
        l       er0,    NEAR _c
        l       er2,    NEAR _d
        bl      __uimulu8sw
        mov     er2,    er0
        l       er4,    NEAR _b
        add     er2,    er4
        st      er2,    NEAR _a

;;      x = c * d / y ;     /* value of c * stored in the temporary is used not evaluated again
*/
        l       er2,    NEAR _y
        bl      __uidivu8sw
        st      er0,    NEAR _x

;;}
        pop     er4
        pop     pc
```

7.5.4 Use Of Basic Algebraic Identities

Expressions that conform to algebraic laws using basic arithmetic identities (0 and 1) are modified, so that unnecessary operations are eliminated. In this process only the following expression are considered.

a+0
a-0
a*0
a*1
a/1

where, 'a' is an integral type variable.

Example 7.22

INPUT

```
int a, b, c, d ;

alg_identities ()
{
     a = b + 0 ;     /* addition is eliminated */
     c = d * 1 ;     /* multiplication is eliminated */
}
```

Assembly code generated by CCU8 for the above function ‘alg_identities’ is shown below:

OUTPUT

```
_alg_identities     :

;;          a = b + 0 ;          /* addition is eliminated */
            l          er0,      NEAR _b
            st         er0,      NEAR _a

;;          c = d * 1 ;          /* multiplication is eliminated */
            l          er0,      NEAR _d
            st         er0,      NEAR _c

;;}
            rt
```

7.5.5 Algebraic Transformation

Expressions are modified, using commutative and associative laws, for optimal use of registers.

Example 7.23

INPUT

```
int a, x, b, c ;

alg_transfer ()
{
     a = x + ( b - c ) ;
}
```

The above statement is transformed to

```
a = ( b - c ) + x ;
```

Assembly code generated by CCU8 for the above function ‘alg_transfer’ is shown below

OUTPUT

```
_alg_transfer      :

;;        a = x + ( b - c ) ;
          l         er0,      NEAR _b
          l         er2,      NEAR _c
          sub       r0,       r2
          subc      r1,       r3
          l         er2,      NEAR _x
          add       er0,      er2
          st        er0,      NEAR _a

;;}
          rt
```

7.5.6 Copy propagation

Given the assignment x <- y for some variables x and y, this optimization replaces later uses of x with uses of y, as long as intervening statements have not changed the values of either x or y. This optimization is carried out only within the straight line code. Result of this optimization may provide opportunity for other optimizations like dead variable elimination.

This optimization is performed only for basic types (signed or unsigned versions of char, short, int and long) and pointer to these basic types.

This optimization is not performed if the two variables are of same size, but differ in sign.

Example 7.24

INPUT

```
int g, j, k ;

void copy_propagation(int a)
{
          k = a ;
          g = j + k ;
}
```

The above program is transformed to

```
int g, j, k ;

void copy_propagation(int a)
{
          k = a ;
          g = j + a ;          ;; 'k' is replaced with 'a'
}
```

Assembly code generated by CCU8 for the above function ‘copy_propagation’ is shown below

OUTPUT

```
_copy_propagation       :

;;{

;;      k = a ;
        st      er0,    NEAR _k

;;      g = j + k ;
        l       er2,    NEAR _j
        add     er0,    er2
        st      er0,    NEAR _g

;;}
        rt
```

7.6 EFFECT OF ALIASING ON OPTIMIZATIONS

An ‘alias’ is a name used to refer to a memory location already referred to by a different name.

As a location can be referred to by more than one variable, performing optimization on variables becomes unsafe. By default CCU8 does not check for aliases. The default optimizations performed by CCU8 may result in unsafe code, when the following assumptions are violated :

1. If a variable is used directly, no pointers are used to reference that variable.
2. If a pointer is used to refer to a variable, that variable is not referred to directly.
3. If a pointer is used to modify a memory location, no other pointers are used to access the same memory location.

The term ‘reference’ means the use of a variable on the right-hand side or left-hand side of an assignment expression or use of a variable as an argument to a function call.

Specifying the command line option /Oa enables CCU8 to check for aliases while performing optimizations. Though this results in correct code, it reduces the extent to which optimizations are performed.

Example 7.25

INPUT

```
int a, b, c, x, y, *ptr ;
```

```
alias_check ()
{
    a = b + c ;

    if (x < a)
    {
        * ptr = 56 ;
        y = b + c ;         /* By default, alias are ignored, so */
                            /* b + c, evaluated earlier is used */
                            /* if /Oa option is specified b + c is evaluated again */
    }
}
```

In the above code fragment, by default, common sub-expression elimination is performed. Hence the sub-expression ‘b + c’, is evaluated only once and a temporary containing the value is used instead of the second evaluation.

Assuming that ‘ptr’ does not point to ‘b’ or ‘c’, the optimization performed is correct. If ‘ptr’ was pointing to ‘b’ or ‘c’, then performing common sub-expression elimination results in assigning an incorrect value to ‘y’.

When the above code fragment is compiled using /Oa option, evaluation of the sub-expression ‘b + c’ is not optimized, thus resulting in correct assignment to ‘y’.

Assembly code generated by CCU8 for the above function ‘alias_check’ in default command line option (all optimizations are performed) is shown below (No /Oa option)

OUTPUT

```
_alias_check    :

        push    bp
        push    er4

;;      a = b + c ;
        l       er0,    NEAR _b
        l       er2,    NEAR _c
        add     er0,    er2
        st      er0,    NEAR _a
        mov     er4,    er0

;;      if (x < a)
        l       er2,    NEAR _x
        cmp     er2,    er0
        bges    _$L1

;;              * ptr = 56 ;
        l       bp,     NEAR _ptr
        mov     er0,    #56
        st      er0,    [bp]
```

```
;;                  y = b + c ;          /* By default, alias are ignored, so */
        st          er4,        NEAR _y

;;          }
_$L1 :

;;}
        pop         er4
        pop         bp
        rt
```

Assembly code generated by CCU8 for the above function ‘alias_check’, when ‘/Oa’ option (perform alias check), is specified in the command line, is shown below:

```
_alias_check        :
        push        bp

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

;;          if (x < a)
        l           er2,        NEAR _x
        cmp         er2,        er0
        bges        _$L1

;;                  * ptr = 56 ;
        l           bp,         NEAR _ptr
        mov         er0,        #56
        st          er0,        [bp]

;;                  y = b + c ;          /* By default, alias are ignored, so */
        l           er0,        NEAR _b
        l           er2,        NEAR _c
        add         er0,        er2
        st          er0,        NEAR _y

;;          }
_$L1 :

;;}
        pop         bp
        rt
```

8. IMPROVING COMPILER OUTPUT

8.1 CONTROLLING OPTIMIZATIONS

CCU8 provides a number of optimization options that can improve program speed. In addition, CCU8 include pragmas to control loop and global optimizations on a local basis within a source program.

8.1.1 Default Optimization

By default, CCU8 performs all optimizations. If no optimization is to be performed, the user must specify /Od option.

8.1.2 Relaxing Alias Checking

By default, CCU8 performs unsafe optimizations. Optimizations may be made safe by specifying the command line option /Oa. But /Oa option may lead to outputs with increased size and with slower execution.

/Oa option may be omitted safely by the user, if multiple aliases referring to the same location, either directly or indirectly, are not used. /Oa may still be omitted safely even if aliases are used in the program, provided that no memory location is referenced by more than one name, within a function.

8.1.3 Controlling Optimization On A Local Basis

All the optimizations that can be controlled form the command line can also be controlled using the OPTIMIZATION pragma. The pragma can be used to enable or disable optimizations for individual functions in the source file. The arguments for the pragma can be among the following ‘Od’,’Om’,’Ot’,’Og’,’Ol’,’Oa’, and ‘default’, each representing the optimization techniques to be applied for the subsequent functions.

Please refer section 5.9 for further details.

8.1.4 Maximum Optimization

The command line option /Om enables the compiler to perform maximum optimizations. By default, all the optimizations are performed only once. But when /Om option is specified, a set of optimizations are performed iteratively, unless CCU8 is unable to perform more optimizations.

/Om option with /Oa option enables the user to obtain an output on which maximum and safe optimizations are performed.

8.1.5 Speed Optimization

The command line option /Ot enables the compiler to perform speed optimization.

/Ot option with /Oa option enables the user to obtain an output on which speed and safe optimizations are performed.

8.1.5.1 SPEED OPTIMIZATION FOR MULTIPLICATION OPERATIONS

Speed Optimization is applied for multiplication operations only when the /Ot option is specified.

In multiplication A*B where A is signed or unsigned variable and B is a positive integer constant with a value of 2n±m or 2n, the following conditions are applied for the speed optimization.

When multiplicand A is a char or an unsigned char variable, speed optimization is performed when positive integer B as a multiplier is transformed into a binary expression and the total number of bits that indicate 1 or [-1] is 3 or less.

Example 8.1

char type multiplication

INPUT

```
char A, Res;

void main()
{
        Res = A * 20;      /* 20 ---> 10100(2)  */
}
```

Compiling example 8.1 with /Ot option, the generated assembly code is given below

OUTPUT

```
;;  Res = A * 20;    /* 20 ---> 10100(2)  */
l       r0,     NEAR _A
mov     r1,     r0
sll     r0,     #02h
add     r0,     r1
sll     r0,     #02h
st      r0,     NEAR _Res
```

Assembly code generated for the above example without /Ot option is given below

OUTPUT

```
;;  Res = A * 20;    /* 20 ---> 10100(2)  */
l       r0,     NEAR _A
mov     r2,     #014h
mul     er0,    r2
st      r0,     NEAR _Res
```

When multiplicand A is an int, unsigned int, short, or unsigned short variable

Speed optimization is performed regardless of the value of positive integer B as a multiplier.

Example 8.2

int type multiplication

INPUT

```
int A, Res;

void main()
{
        Res = A * 73;       /* 73 ---> 1001001(2)  */
}
```

Generated assembly code for the above example with /Ot option is given below

OUTPUT

```
;;   Res = A * 73; /* 73 ---> 1001001(2)  */
l       er0,    NEAR _A
mov     er2,    er0
sllc    r1,     #03h
sll     r0,     #03h
```

```
add     er0,    er2
sllc    r1,     #03h
sll     r0,     #03h
add     er0,    er2
st      er0,    NEAR _Res
```

Generated assembly code for the above example without /Ot option is given below

OUTPUT

```
;;   Res = A * 73; /* 73 ---> 1001001(2)  */
l       er0,    NEAR _A
mov     r2,     #049h
mov     r3,     #00h
bl      __imulu8sw
st      er0,    NEAR _Res
```

When multiplicand A is a long variable

Speed optimization is not performed for long type multiplications. __lmulu8sw emulation routine will be used for performing long type multiplication.

8.1.5.2 SPEED OPTIMIZATION FOR DIVISION OPERATIONS

The speed optimization is performed for division operations satisfying the following conditions.

In division A / B where A is a signed variable and B is a positive integer with a value of 2n.

By checking the sign of A , the division is transformed into a shift operation.

Example 8.3

Division with signed char type variables

INPUT

```
signed char Res , A ;
void CharDivision ()
{
        Res = A / 8 ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;      Res = A / 8 ;
        l       r0,     NEAR _A
        bps     _$M1
        add     r0,     #07h
```

```
_$M1 :
        sra     r0,     #03h
        st      r0,     NEAR _Res
```

Example 8.4

Division with signed int type variables

INPUT

```
signed int Res , A ;

void IntDivision ()
{
        Res = A /128  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;      Res = A /128 ;
        l       er0,    NEAR _A
        bps     _$M1
        add     r0,     #07fh
        addc    r1,     #00h
_$M1 :
        srlc    r0,     #07h
        sra     r0,     #07h
        st      er0,    NEAR _Res
```

Example 8.5

Division with signed long type variables

INPUT

```
signed long Res , A ;
void LongDivision ()
{
        Res = A / 512  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;      Res = A / 512;
        l       er0,    NEAR _A
        l       er2,    NEAR _A+02h
        bps     _$M1
        add     r0,     #0ffh
```

```
        addc    r1,     #01h
        addc    r2,     #00h
        addc    r3,     #00h
_$M1:
        mov     r0,     r1
        mov     r1,     r2
        mov     r2,     r3
        extbw   er2
        srlc    r0,     #01h
        srlc    r1,     #01h
        sra     r2,     #01h
        st      er0,    NEAR _Res
        st      er2,    NEAR _Res+02h
```

8.1.5.3 SPEED OPTIMIZATION FOR SHIFT OPERATIONS

The speed optimization for the Shift operations is performed when /Ot option is specified.

Example 8.6

Left Shift Operation involving signed char type variables

INPUT

```
signed char Res , A , B ;
void CharShift ()
{
        Res = A << B  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;   Res = A << B ;
        l       r0,     NEAR _A
        l       r1,     NEAR _B
        cmp     r1,     #07h
        bgt     _$M1
        sll     r0,     r1
        b       _$M2
_$M1 :
        mov     r0,     #00h
_$M2 :
        st      r0,     NEAR _Res
```

Example 8.7

Right Shift Operation involving signed int type variables

INPUT

```
signed int Res , A , B ;
void IntShift ()
{
        Res = A >> B  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;   Res = A >> B ;
        l       er0,    NEAR _A
        l       er2,    NEAR _B
_$M2 :
        cmp     r2,     #07h
        cmpc    r3,     #00h
        ble     _$M1
        srlc    r0,     #07h
        sra     r1,     #07h
        add     er2,    #-7
        bne     _$M2
_$M1 :
        srlc    r0,     r2
        sra     r1,     r2
        st      er0,    NEAR_Res
```

Example 8.8

Left Shift Operation involving unsigned long type variables

INPUT

```
unsigned long Res , A , B ;
void ULongShift ()
{
        Res = A << B  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;   Res = A << B ;
        l       er0,    NEAR _A
        l       er2,    NEAR _A+02h
        l       er4,    NEAR _B
_$M2 :
        cmp     r4,     #07h
```

```
                cmpc    r5,     #00h
                ble     _$M1
                sllc    r3,     #07h
                sllc    r2,     #07h
                sllc    r1,     #07h
                sll     r0,     #07h
                add     er4,    #-7
                bne     _$M2
_$M1 :
                sllc    r3,     r4
                sllc    r2,     r4
                sllc    r1,     r4
                sll     r0,     r4
                st      er0,    NEAR _Res
                st      er2,    NEAR_Res+02h
```

For the shift operations involving bit-field members, and bit-wise AND operation upon the shift value , the optimization in the code generated will be performed in the default option as explained below.

When the Shift value is a bit-field operand of size less than or equal to 3 , the optimization is done with default option.

Example 8.9

Left Shift Operation involving bit-field member of size less than or equal to three.

INPUT

```
struct  sTag
{
        unsigned int  B : 3 ;
} sObj ;

unsigned int Res , A  ;

void BitFieldShift ()
{
        Res = A << sObj.B  ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;   Res = A << sObj.B  ;
        l       er0,    NEAR _sObj
        and     r0,     #07h
        l       er2,    NEAR _A
        sllc    r3,     r0
        sll     r2,     r0
        st      er2,    NEAR _Res
```

When the bit-wise AND operation is performed upon the shift value with value less than or equal to 7 , the optimization is done with default option.

Example 8.10

Left Shift Operation involving bit-wise AND operation upon right operand with value less than or equal to 7.

INPUT

```
unsigned int Res , A  , B ;

void BitwiseANDShift ()
{
        Res = A << ( B & 7 ) ;
}
```

Assembly code generated for the above function is given below.

OUTPUT

```
;;   Res = A << ( B & 7 ) ;
        l       er0,    NEAR _B
        and     r0,     #07h
        l       er2,    NEAR _A
        sllc    r3,     r0
        sll     r2,     r0
        st      er2,    NEAR_Res
```

8.2 REMOVING STACK PROBES

Program execution may be speeded up by removing calls to stack-checking-routines known as stack probes. Stack probes verify that a program has enough space to allocate required local variables.

The potential disadvantage in removing stack probes is that stack-overflow goes undetected. However, this technique may be useful for programs that are known not to exceed the available stack space.

By default, stack probe routines are not called. The command line option /ST enables CCU8 to call stack probe routines at the beginning of each function.

Stack checking may be controlled on local basis also by using either #pragma CHECKSTACK_ON or #pragma CHECKSTACK_OFF. Stack checking is turned off for any function following #pragma CHECKSTACK_OFF and turned on for any function following #pragma CHECKSTACK_ON.

8.3 CONTROLLING ALLOCATION OF VARIABLES

Pragmas of CCU8 may be used in controlling the allocation of variables. This enables CCU8 to use variety of addressing modes in order to improve the assembly output.

Using #pragma ABSOLUTE, a variable may be allocated anywhere in data memory. Other pragmas like NVDATA, ROMWIN etc., enables the user to allocate the given variable in any part of the memory.

Using #pragma SEGNOINIT a segment name or a segment starting address can be specified for uninitialized global, static global and local static variables.

Using #pragma SEGNVDATA a segment name or a segment starting address can be specified for global, static global and local static variables. Variables are allocated in nonvolatile memory region.

Using #pragma SEGINIT a segment name or a segment starting address can be specified for initialized global, static global and local static variables.

Using #pragma SEGCONST a segment name or a segment starting address can be specified for global, static global and local static variables declared const. Variables are allocated in read only memory region.

For the variables allocated to the segments specified by SEGNOINIT, SEGNVDATA and SEGINIT pragma, variables should not be specified with either of the ABSOLUTE or NVDATA pragmas.

For the variables allocated to the segment specified by SEGCONST pragma, variable should not be specified with ABSOLUTE pragma.

8.4 MIXED LANGUAGE PROGRAMMING

This section explains how to use U8 assembly language routines with ‘C’ programs and functions compiled using CCU8. In particular it explains how to call assembly language routines from ‘C’ programs and how to call ‘C’ language functions from an assembly language routine.

8.4.1 Preserving Register Contents

CCU8 preserves register contents as follows:

Normal functions and software interrupt functions can use registers R0-R3 freely without preserving. When a common or software interrupt function uses any registers R4-R15 (including BP and FP), it saves that register in function prologue and restores that register in function epilogue. During function call, contents of registers R4-R15 are preserved, and contents of registers R0-R3 are destroyed. If software interrupt function does not have return value and argument variable, then registers R0-R3 will be preserved.

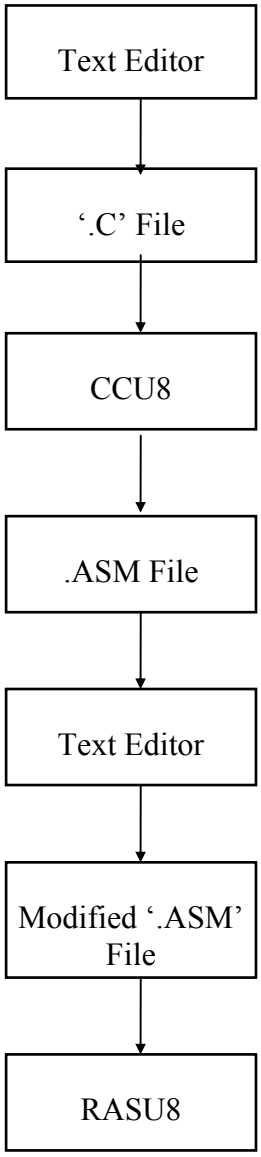
Interrupt function save all registers that are used in the function in function prologue, and restore the corresponding registers in function epilogue. So, user should conform to the above mentioned protocol during mixed language programming.

8.4.2 Combining Assembly And ‘C’ Programs

Some of the methods by which a programmer can combine an assembly language routine and a ‘C’ program are given below:

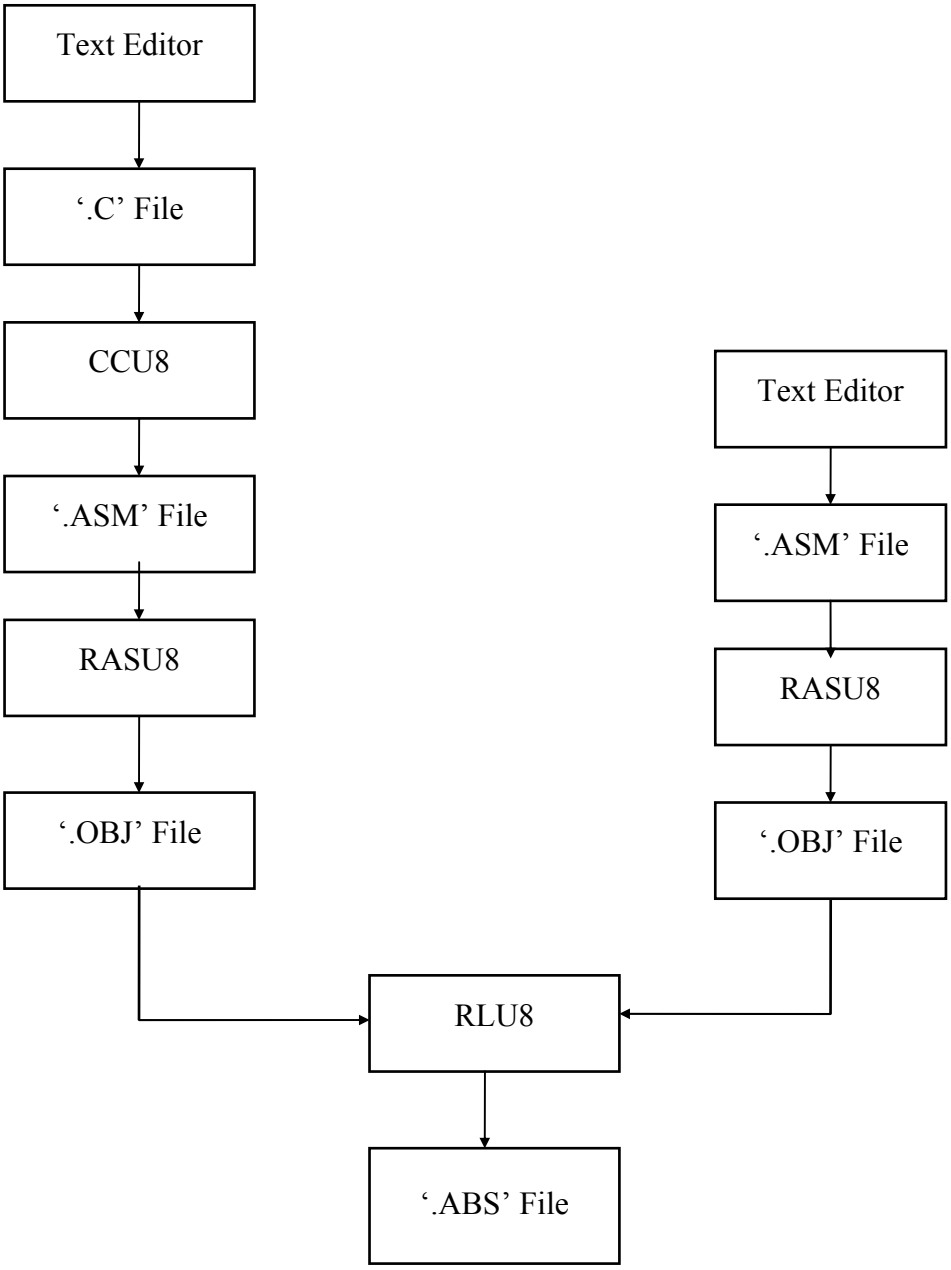
Method 1

In this method, programmer writes a ‘C’ program and then compiles the ‘C’ program using CCU8. The output produced by CCU8 is an assembly language file containing CCU8 mnemonics. Programmer can edit this file using any text editor and add the necessary assembly language routines. The resulting file can then be assembled and linked using RASU8 and RLU8 respectively to produce the absolute file.



Method 2

In this method, programmer writes a ‘C’ program and compiles it using CCU8. The compiler produces as output an ‘.ASM’ file. The programmer creates an assembly language file, containing the assembly routines to be mixed with the ‘C’ program. The two assembly program files can be assembled separately using RASU8. The result will be two ‘.OBJ’ files. These two ‘.OBJ’ files can be linked using the linker RLU8.



Method 3

Using #asm and #endasm

In this method, programmer writes assembly instructions directly in the source file using preprocessor directives **#asm** and **#endasm**. A procedure or a part of a procedure may be written in assembly language and enclosed within the two directives **#asm** and **#endasm**. CCU8 outputs whatever is specified between these two directives as it is in the output file. Since local variables may be assigned to registers, any access to local variables inside asm block, may not yield intended results. Therefore, any data passing across asm blocks must be only through global variables.

Method 4

Using #pragma asm and #pragma endasm

In this method, programmer writes assembly instructions directly in the source file using pragma directives **#pragma asm** and **#pragma endasm**. Processing of the text inside these pragma directives are same as the processing of **#asm** and **#endasm**.

Method 5

Using __asm keyword

Syntax:

__asm (string)

In this method, programmer writes assembly instructions directly in the source file using **__asm** keyword. A procedure or a part of a procedure may be specified as a string argument to **__asm** keyword. CCU8 outputs whatever is the argument to this keyword as it is in the output file.

The return value of **__asm** keyword cannot be used.

CCU8 issues error message in the following cases:

- If the specified argument is not a string.
- If more than one argument is specified.
- If return value of '**__asm**' keyword is used.

The following examples show erroneous cases:

Example 8.11

INPUT

```
void fn ()
{
        __asm (“DI\n”, “EI\n” ) ;
}
```

CCU8 outputs error message for the above program as more than one argument is specified for **__asm** keyword.

Example 8.12

INPUT

```
void fn ()
{
        return __asm (“DI\n”, “EI\n” ) ;
}
```

CCU8 outputs error message for the above program as return value of **__asm**. keyword is used.

8.4.3 Calling Conventions Of CCU8

CCU8 follows certain conventions while passing values to ‘C’ functions or while receiving values from ‘C’ language function calls. Hence, assembly language routines must follow these conventions. CCU8 passes arguments to any given function by assigning them to R0, R1, R2 and R3 registers. The type of arguments that is assigned to registers are char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float and pointer. Arguments that are not assigned to registers because of the type difference or register shortage is assigned to the stack.

The behaviour of Swi function is same as normal function call.

The following rules are followed if the arguments are assigned to registers :

- Arguments located from the left to right is assigned to registers from R0 to R3.
- One byte argument is assigned to Rn (where n = 0, 1, 2 or 3).
- Two bytes argument is assigned to Rn for lower byte and Rn+1 for higher byte (where n = 0 or 2).
- Four bytes argument is assigned to R0 for the lowest byte, R1 for the lower middle byte, R2 for upper middle byte and R3 for the highest byte.
- Near pointer argument is assigned to Rn for lower offset byte and Rn+1 for higher offset byte (where n = 0 or 2).
- Far pointer argument is assigned to R0 for lower offset byte, R1 for higher offset byte and R2 for the segment.
- R0, R1, R2 and R3 registers that are assigned arguments are saved by copying them to R8, R9, R10 and R11 in the prologue if it were necessary. Therefore, R8, R9, R10 and R11 registers are pushed into the stack in the prologue and popped from the stack in the epilogue if they were used in the function code.

Example 8.13

INPUT

```
char g_var;
void function(char c)
{
  g_var = c;
}
```

OUTPUT

```
_function       :

;;{

;;  g_var = c;
        st      r0,     NEAR _g_var

;;}
rt
```

Example 8.14

INPUT

```
# pragma SWI function 0x80
int c;
void
function(int a, int b)
```

```
{
  c = a + b ;
}
```

OUTPUT

```
_function          :
;;{
            push    elr, epsw
;;   c = a + b ;
            add     er0,    er2
            st      er0,    NEAR _c
;;}
pop     psw, pc
```

The following rules are followed if the arguments are assigned to stack :

- Pushing the value of each of the arguments into the stack from right to left
- If an argument is an expression, CCU8 computes the expression's value before pushing it onto the stack. The expression evaluation is carried out from left to right, but the arguments are pushed into the stack from right to left
- The number of bytes pushed into the stack are equal to the size of the argument
- After a function returns control, the calling routine is responsible for removing the arguments from the stack. This is achieved by adding the number of bytes pushed as arguments to SP.

Example 8.15

INPUT

```
int var1 ;
char __far * var2 ;
char var3 ;

void
fn1 ( int a, char __far * b, char c )
{
            var1 = a ;
            var2 = b ;
            var3 = c ;
}
```

OUTPUT

```
_fn1    :

            push    fp
            mov     fp,     sp
            push    er10
            mov     r10,    r2
```

```
;;      var1 = a ;
        st      er0,    NEAR _var1

;;      var2 = b ;
        l       er0,    2[fp]
        l       r2,     4[fp]
        st      er0,    NEAR _var2
        st      r2,     NEAR _var2+02h

;;      var3 = c ;
        st      r10,    NEAR _var3

;;}
        pop     er10
        mov     sp,     fp
        pop     fp
        rt
```

Example 8.16

INPUT

```
char var1 ;
long var2 ;
int var3 ;

void
fn2 ( char a, long b, int c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

OUTPUT

```
_fn2    :

        push    fp
        mov     fp,     sp
        push    er10
        mov     er10,   er2

;;      var1 = a ;
        st      r0,     NEAR _var1

;;      var2 = b ;
        l       er0,    2[fp]
        l       er2,    4[fp]
```

```
                st      er0,    NEAR _var2
                st      er2,    NEAR _var2+02h

        ;;      var3 = c ;
                st      er10,   NEAR _var3

        ;;}
                pop     er10
                mov     sp,     fp
                pop     fp
                rt
```

Example 8.17

INPUT

```
char __far * var1 ;
int var2 ;
char var3 ;

void
fn3 ( char __far * a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

OUTPUT

```
__fn3   :
        push    fp
        mov     fp,     sp

;;      var1 = a ;
        st      er0,    NEAR _var1
        st      r2,     NEAR _var1+02h

;;      var2 = b ;
        l       er0,    2[fp]
        st      er0,    NEAR _var2

;;      var3 = c ;
        st      r3,     NEAR _var3

;;}
        mov     sp,     fp
        pop     fp
        rt
```

Example 8.18

INPUT

```
char var1 ;
int var2 ;
char var3 ;

void
fn4 ( char a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

OUTPUT

```
__fn4   :
        push    fp
        mov     fp,     sp

;;      var1 = a ;
        st      r0,     NEAR _var1

;;      var2 = b ;
        st      er2,    NEAR _var2

;;      var3 = c ;
        l       r0,     2[fp]
        st      r0,     NEAR _var3

;;}
        mov     sp,     fp
        pop     fp
        rt
```

Example 8.19

INPUT

```
double var1 ;
int var2 ;
char var3 ;

void
fn5 ( double a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

OUTPUT

```
_fn5    :

        push    fp
        mov     fp,     sp
        push    xr4
        push    er8
        mov     er8,    er0

;;      var1 = a ;
        lea     2[fp]
        l       qr0,    [ea]
        lea     OFFSET _var1
        st      qr0,    [ea]

;;      var2 = b ;
        st      er8,    NEAR _var2

;;      var3 = c ;
        st      r2,     NEAR _var3

;;}
        pop     er8
        pop     xr4
        mov     sp,     fp
        pop     fp
        rt
```

Example 8.20

INPUT

```
# pragma SWI function 0x80
void
function(long a, long b, int c)
{
   function2(a, b, c);
}
```

OUTPUT

```
_function       :

;;{
        push    elr, epsw, lr, ea
        push    fp
        mov     fp,             sp
        push    xr8
        l       r8,             DSR
        push    r8
        mov     er8,    er0
        mov     er10,   er2

;;  function2(a, b, c);
        l       er0,    14[fp]
        push    er0
        l       er0,    10[fp]
        l       er2,    12[fp]
        push    xr0
        mov     er0,    er8
        mov     er2,    er10
        bl      _function2
        add     sp,             #6

;;}
        pop     r8
        st      r8,             DSR
        pop     xr8
        mov     sp,             fp
        pop     fp
        pop     ea, lr, psw, pc
```

8.4.4 Return Values

Assembly language routines that wish to return values to a ‘C’ program or receive return values from ‘C’ functions must follow CCU8 return value conventions. If the function has a return value of size less than or equal to 4 bytes, CCU8 places the return value of function in the registers R0, R1, R2, R3, depending on the size of the return value. If the return value type is structure or union or double, CCU8 passes the address of the variable to which the return value is assigned, as the first argument (i.e. ER0 shall contain the address if the function is not qualified by the __noreg qualifier). Therefore, the return value is updated in the called function.

The behaviour of Swi function is same as normal function call.

The following rules are followed if the return values are assigned to registers :

- Return values are assigned to registers from R0 to R3
- One byte return value is assigned to R0 register
- Two bytes return value is assigned to R0 for lower byte and R1 for higher byte
- Four bytes return value is assigned to R0 for the lowest byte, R1 for lower middle byte, R2 for upper middle byte and R3 for the highest byte
- Near pointer is assigned to R0 for lower offset byte and R1 for higher offset byte
- Far pointer is assigned to R0 for lower offset byte, R1 for higher offset byte and R2 for the segment.

Example 8.21

INPUT

```
int add_int (int a, int b)
{
        return ( a + b ) ;
}
long add_long (long a, long b)
{
        return ( a + b ) ;
}

double add_double ( double a, double b)
{
        return ( a + b ) ;
}
```

OUTPUT

```
_add_int:

;;      return ( a + b ) ;
        add     er0,    er2

;;}
        rt

_add_long       :

        push    fp
        mov     fp,     sp
        push    xr4

;;      return ( a + b ) ;
        l       er4,    2[fp]
        l       er6,    4[fp]
        add     er0,    er4
        addc    r2,     r6
        addc    r3,     r7

;;}
        pop     xr4
        mov     sp,     fp
        pop     fp
        rt

_add_double     :

        push    lr
        push    fp
        mov     fp,     sp
        push    xr4
        push    er8
        mov     er8,    er0

;;      return ( a + b ) ;
        lea     12[fp]
        l       qr0,    [ea]
        push    qr0
        lea     4[fp]
        l       qr0,    [ea]
        push    qr0
        bl      __daddu8sw
        add     sp,     #8
        pop     qr0
        lea     [er8]
        st      qr0,    [ea]
```

```
;;}
        pop     er8
        pop     xr4
        mov     sp,     fp
        pop     fp
        pop     pc
```

Example 8.22

INPUT

```
# pragma SWI function 0x80 1
int a;
int function()
{
   return a ;
}
```

OUTPUT

```
_function       :

;;{

;;   return a ;
        l       er0,    NEAR _a

;;}
        rti
```

8.4.5 Interrupt Handling Routines In Assembly

CCU8 allows interrupt handling routines to be written in 'C'. Interrupt handling routines must reside in physical segment 0 of the CODE memory. The appropriate interrupt vector must be initialized by the starting address of the routine. The last statement of an interrupt handling routine must be "rti" instruction.

8.4.6 Referring 'C' Variables

Assembly routines can refer to global variables used in 'C' source program. Initialized global variables can be referred by declaring them as "EXTRN" in assembly routines. Such variables should not be declared as "PUBLIC" in assembly. Uninitialized global variables can be referred by declaring them using "PUBLIC" or "EXTRN" or "COMM" pseudo

instructions. Global variables which are declared as "**extern**" in 'C' program can be referenced in assembly routines by declaring them as "PUBLIC" or "COMM".

8.5 QUALIFYING FUNCTIONS WITH '__NOREG'

A function may be qualified with **__noreg** to inform the compiler to use Stack for it's arguments.

If __noreg function specifier is specified in function prototype or function definition, CCU8 passes arguments not by register but by stack. By default, __noreg qualifier is applicable to 'main' function and variable argument functions.

The behaviour of Swi function is same as normal function call.

Example 8.23

INPUT

```
int __noreg add ( int a, int b );
int var1, var2 ;

int __noreg add ( int a, int b )
{
        int     l_ret   ;
        l_ret = a + b ;
        return ( l_ret ) ;
}

fn ()
{
        var1 = add ( var1, var2 ) ;
}
```

OUTPUT

```
_add    :
;;{
        push    fp
        mov     fp,     sp

;;      return ( l_ret ) ;
        l       er0,    2[fp]
        l       er2,    4[fp]
        add     er0,    er2

;;}
        mov     sp,     fp
```

```
                pop     fp
                rt


        _fn     :
        ;;{
                push    lr

        ;;      var1 = add ( var1, var2 ) ;
                l       er0,    NEAR _var2
                push    er0
                l       er0,    NEAR _var1
                push    er0
                bl      _add
                add     sp,     #4
                st      er0,    NEAR _var1

        ;;}
                pop     pc
```

Example 8.24

INPUT

```
# pragma SWI add 0x80
int __noreg add ( int a, int b );
int __noreg add ( int a, int b )
{
        int l_ret ;
        l_ret = a + b ;
        return ( l_ret ) ;
}
```

OUTPUT

```
        _add    :

        ;;{
                push    elr, epsw
                push    fp
                mov     fp,     sp

        ;;  return ( l_ret ) ;
                l       er0,    6[fp]
                l       er2,    8[fp]
                add     er0,    er2

        ;;}
                mov     sp,     fp
                pop     fp
                pop     psw, pc
```

8.6 BUILT-IN FUNCTIONS

8.6.1 __EI() and __DI()

CCU8 supports built-in functions **__EI()** and **__DI()**. When a built-in function is called, the body of that built-in function is inlined in the assembly listing file.

These function names are reserved keywords. CCU8 issues error message if a built-in function is defined in the source file.

CCU8 issues error message, if arguments are specified for these built-in functions.

The prototypes of these built-in functions are:

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

The built-in function "**__EI()**" expands to the assembly code **EI** and the built-in function "**__DI()**" expands to the assembly code **DI**.

Example 8.25

INPUT:

```
static void intr () ;
# pragma interrupt intr 0xA
/* or
# pragma interrupt intr 0xA 2
*/

static void
intr ()
{
        __EI () ;
}
```

The following is the code generated for the function "intr" defined in the above program:

OUTPUT

```
_intr   :
        push    elr, epsw

;;      __EI () ;
        ei

;;}
        pop     psw, pc
```

Example 8.26

INPUT

```
int __EI (int arg1) ;
```

In the above example, CCU8 issues an error as the prototype of built-in function "**__EI**" is redefined.

8.6.2 __segbase_n()

This Built-in function is used to get the segment starting address. The function returns __near pointer type

Prototype of built-in function is :

void __near * __segbase_n("segment_name")

A valid segment name is specified as a parameter to the built-in function __segbase_n. The function returns the segment starting address of the segment that is specified as a parameter.

Segment name passed to the __segbase_n() function is either default segment, automatically generated by the compiler or defined by the user using the SEGCODE, SEGINTR, SEGINIT, SEGNOINIT, SEGCONST, and SEGNVDATA pragmas

__segbase_n function can be used in both, local as well as in global scope.

In local scope function is expanded to the Assembler instructions that use the OFFSET operand of Assembler.

The following example shows the usage of __segbase_n in local scope :

Example 8.27

INPUT

```
#pragma segnoinit __near "SEGNAME"
int var;
void Function()
{
        int __near * np = __segbase_n("SEGNAME") ;
}
```

CCU8 generates the following output for above example:

OUTPUT

```
mov r0, #BYTE1 OFFSET SEGNAME      ;; Lower byte of near pointer is accessed
mov r1, #BYTE2 OFFSET SEGNAME      ;; Higher byte of near pointer is accessed


        rseg SEGNAME
_var :
        ds 02h
```

In global scope, function is used to initialize global variables, table segment is generated to store the value returned by this function

Example 8.28

INPUT

```
#pragma segnoinit __near "SEGRAM1"
int var;
unsigned int __near * nbase = __segbase_n( "SEGRAM1" ) ;
```

CCU8 generates the following output for above example:

OUTPUT

```
        rseg $$NINITTAB         ;; Table segment is generated
        dw      SEGRAM1         ;; OFFSET portion of SEGRAM1 of type address is stored

        rseg $$NINITVAR
_nbase :
        ds      02h     ;; Memory equivalent to size of pointer variable is reserved

        rseg SEGRAM1
_var :
        ds 02h          ;; Memory equivalent to size of integer variable is reserved
```

8.6.3 __segbase_f()

This Built-in function is used to get the segment starting address. The function returns __far pointer type

Prototype of built-in function is :

void __far * __segbase_f("segment_name")

A valid segment name is specified as a parameter to the built-in function __segbase_f. The function returns the segment starting address of the segment that is specified as a parameter.

Segment name passed to the __segbase_f() function is either default segment, automatically generated by the compiler or defined by the user using the SEGCODE, SEGINTR, SEGINIT, SEGNOINIT, SEGCONST, and SEGNVDATA pragmas

__segbase_f function can be used in both, local as well as in global scope.

In local scope function is expanded to the Assembler instructions that use the SEG operand and the OFFSET operand of Assembler.

Example 8.29

INPUT

```
#pragma segnoinit __far "SEGNAME1"
int siVar;
void fn(void)
{
        int __far * fp = __segbase_f("SEGNAME1") ;
}
```

CCU8 generates the following output for above example:

OUTPUT

```
mov r0, #BYTE1 OFFSET SEGNAME1  ;; Lower byte of near pointer is accessed
mov r1, #BYTE2 OFFSET SEGNAME1  ;; Higher byte of near pointer is accessed
mov r2, #SEG SEGNAME1    ;; Physical segment address portion of SEGNAME1 is accessed
.....
.....
.....
 rseg SEGNAME1
_siVar :
 ds 02h
```

In global scope, function is used to initialize global variables, table segment is generated to store the value returned by this function

Example 8.30

INPUT

```
#pragma segnoinit __far "SEGNAME1"
int __far var;
unsigned int __far  * fbase = __segbase_f( "SEGNAME1" ) ;
```

CCU8 generates the following output for above example:

OUTPUT

```
.....
.....
public _fbase
```

```
            .....
            .....

            rseg $$NINITTAB
            dw      OFFSET (SEGNAME1) :: Offset of  the address is accessed
            db      SEG (SEGNAME1) :; Physical segmsnt address is accessed

            rseg $$NINITVAR
_fbase :
            ds      03h             ;; Memory for far pointer variable is reserved

            rseg SEGNAME1
_var :
            ds 02h
```

8.6.4 __segsize()

This Built-in function is used to get the segment size. The size, returned by the function is of unsigned int type.

Prototype of built-in function is :

unsigned int __segsize("segment_name")

A valid segment name is specified as a parameter to the built-in function __segsize. The function returns the size of the segment specified in parameter. The function is expanded to Assembler instructions that use the SIZE operand.

Function can be used in both, local as well as in global scope.

The following example shows the usage of function in local scope

Example 8.31

INPUT

```
#pragma segnoinit "SEGRAM"
int sivar1;
void fn(void)
{
        unsigned int uisize = __segsize("SEGRAM") ;
}
```

CCU8 generates the following output when /near option is specified for above example:

OUTPUT

```
.................
.................
l er0,  $$S0                    ;; Size of segment is loaded in er0 register
.................
.................
rseg $$NTABsize1                ;; Table segment for near data access type
$$S0 :
DW  SIZE  SEGRAM                ;; Size of segment is stored
.................
.................
  rseg SEGRAM
_sivar1 :
  ds 02h
.................
.................
```

CCU8 generates the following output when /far option is specified for above example:

OUTPUT

```
.................
.................
l er0,  $$S0                    ;; Size of segment is loaded in er0 register
.................
.................
rseg $$FTABsize1                ;; Table segment for far data access type
$$S0 :
DW  SIZE  SEGRAM
.................
.................
  rseg SEGRAM
_sivar1 :
  ds 02h
.................
.................
```

The following example shows the usage of function in global scope

Example 8.32

INPUT

```
#pragma segdef "SEGRAM1" "DATA"
unsigned int size = __segsize( "SEGRAM1" ) ;
```

CCU8 generates the following output for above example:

OUTPUT

```
                .................
                .................

                SEGRAM1 segment data 2h any     ;; Code generated for Segdef pragma
                                                ;; Refer Segdef pragma specification for
                .................               ;; more details
                .................
                public _size
                .................
                .................

                rseg $$NINITTAB
                dw      SIZE    SEGRAM1

                rseg $$NINITVAR
_size :
                ds      02h

                end
```

8.7 STARTUP ROUTINE

The start up routine “$$start_up” is an assembly language routine containing stack and SFR initializations. Control is passed to the main function from the start up routine by means of a jump instruction

```
bl      _main
```

The routine is present in a separate start up assembly source file. This file may be modified by the user to include additional initializations. The start up object file may be directly specified while invoking RLU8.

9. EMULATION LIBRARIES

CCU8 supports the data type **long**, **float** and **double** although the U8 architectures do not support these data types. These data types are supported by using the floating point and long emulation routines. These routines are provided in three library files longu8.lib, floatu8.lib and doubleu8.lib. All arithmetic operations involving **long**, **float** and **double** data types are carried out with the help of these routines. CCU8 outputs a call instruction to the appropriate routine to perform the arithmetic operation. These routines are provided for small and large memory models.

Following files are stored in the emulation library:

Library File Name	Contents
LONGU8.LIB	Integer arithmetic
DOUBLEU8.LIB	Double-precision floating point arithmetic
FLOATU8.LIB	Single-precision floating point arithmetic

Following routines are stored in the emulation library:

LONGU8.LIB

Routines	Function	Stack cost	
		Small (*xx*=sw)	Large (*xx*=lw)
__cdivu8*xx*	Signed character division	4	6
__cmodu8*xx*	Signed character modulus	4	6
__cmulu8*xx*	Signed character multiplication	2	2
__idivu8*xx*	Signed integer division	14	18
__imodu8*xx*	Signed integer modulus	14	18
__uidivu8*xx*	Unsigned integer division	10	12
__uimodu8*xx*	Unsigned integer modulus	10	12
__imulu8*xx*	Signed/Unsigned integer multiplication	6	6
__ldivu8*xx*	Signed long integer division	20	24
__lmodu8*xx*	Signed long integer modulus	20	24
__uldivu8*xx*	Unsigned long integer division	18	20
__ulmodu8*xx*	Unsigned long integer modulus	18	20
__lmulu8*xx*	Signed/Unsigned long integer multiplication	22	24
__indru8*xx*	Indirection call routine (Large Model Only)	-	0
__chstu8*xx*	Check of stack over flow	4	4
__regpushu8*xx*	Register saving subroutine	12	12
__regpopu8*xx*	Register restoring subroutine	0	0

DOUBLEU8.LIB

Routines	Function	Stack cost	
		Small (*xx*=sw)	Large (*xx*=lw)
__faddu8*xx*	Single precision float addition	48	52
__fsubu8*xx*	Single precision float subtraction	48	52
__fmulu8*xx*	Single precision float multiplication	48	52
__fdivu8*xx*	Single precision float division	66	72
__fcmpu8*xx*	Single precision float comparison	46	50
__fildu8*xx*	Long to float conversion	36	40
__fuldu8*xx*	Unsigned long to float conversion	36	40
__ftodu8*xx*	Converts float to double	36	40
__ftolu8*xx*	Float to long conversion	36	40
__flnotu8*xx*	Single precision float logical not	6	6
__fnegu8*xx*	Single precision float negation	4	4
__daddu8*xx*	Double addition	48	52
__dsubu8*xx*	Double subtraction	48	52
__dmulu8*xx*	Double multiplication	48	52
__ddivu8*xx*	Double division	66	72
__dcmpu8*xx*	Double comparison	46	50
__dildu8*xx*	Converts long to double	36	40
__duldu8*xx*	Unsigned long to double conversion	36	40
__dtofu8*xx*	Double to float conversion	36	40
__dtolu8*xx*	Double to long conversion	36	40
__dlnotu8*xx*	Double logical not	6	6
__dnegu8*xx*	Double negation	4	4

FLOATU8.LIB

Routines	Function	Stack cost	
		Small (*xx*=sw)	Large (*xx*=lw)
__faddu8*xx*	Single precision float addition	40	44
__fsubu8*xx*	Single precision float subtraction	40	44
__fmulu8*xx*	Single precision float multiplication	40	44
__fdivu8*xx*	Single precision float division	48	52
__fcmpu8*xx*	Single precision float comparison	38	42
__fildu8*xx*	Long to float conversion	32	36
__fuldu8*xx*	Unsigned long to float conversion	32	36
__ftodu8*xx*	Converts float to double	32	36
__ftolu8*xx*	Float to long conversion	32	36
__flnotu8*xx*	Single precision float logical not	6	6
__fnegu8*xx*	Single precision float negation	4	4
__daddu8*xx*	Double addition	40	44
__dsubu8*xx*	Double subtraction	40	44
__dmulu8*xx*	Double multiplication	40	44
__ddivu8*xx*	Double division	48	52
__dcmpu8*xx*	Double comparison	38	42
__dildu8*xx*	Converts long to double	32	36
__duldu8*xx*	Unsigned long to double conversion	32	36
__dtofu8*xx*	Double to float conversion	32	36
__dtolu8*xx*	Double to long conversion	32	36
__dlnotu8*xx*	Double logical not	6	6
__dnegu8*xx*	Double negation	4	4

10. ASSEMBLING AND LINKING THE COMPILER OUTPUT

CCU8 creates as output an assembly file. In order to create an object file, the output from the compiler should be assembled using the Re-locatable Assembler RASU8. To invoke the assembler the following command line should be used.

C:> RASU8 FILE <CR>

Where 'FILE' specifies the name of the output file created by the Compiler, CCU8. If more than one file is compiled, then each of the output file should be assembled separately.

In 'C' language upper and lower case characters are different, so CCU8 generates code that is case sensitive. By default RASU8 does not differentiate upper and lower case characters, so in order to differentiate uppercase and lowercase characters '/CD' option should be specified in the command line of RASU8 as shown below:

C:> RASU8 FILE /CD <CR>

The assembler produces as output an object file. In order to debug the 'C' programs using 'C' source level debugger, the compiler output should be assembled using the '/SD' option as follows :

C:> RASU8 FILE /SD <CR>

This option informs the assembler to create the object file with necessary debugging information. This option must be specified to RASU8, only when the files are compiled with /SD option in CCU8.

The object files created by RASU8 can be linked using the Object Linker RLU8. The linker produces as output an absolute object file.

To link the object programs, the following command line should be used :

C:> RLU8 FILE1 FILE2,....,,/CC <CR>

where ‘FILE1 FILE2,...’ are the names of the input object files to RLU8. The /CC option informs RLU8, that the inputs are files compiled by CCU8 and assembled using RASU8. Thus RLU8 would take appropriate steps to reserve space for the stack and to initialize the stack pointer.

The output assembly file created by CCU8 makes use of routines, which are available in the libraries ‘doubleu8.lib”, ‘floatu8.lib’ and ‘longu8.lib’. RLU8 searches these three library files to resolve the externals. The RLU8 searches these library files in standard directories specified by the environment variable LIBU8 when /CC option is specified. The environment variable can be set by the following command at the DOS prompt.

C:> SET LIBU8=directory <CR>

Where ‘directory’ gives the name of the standard directory which will be used by RLU8 to search the library files.

In order to create absolute files with debugging information, the object files should be linked using the /SD option as follows :

C:> RLU8 FILE1 FILE2,....,,/CC /SD <CR>

This option informs the linker to create the absolute file with necessary debugging information.

11. EXIT CODES

CCU8, on termination passes the control to the operating system, while passing the control to the operating system, CCU8 returns a numeric value called exit code. The exit codes and the corresponding exit status are listed below.

TABLE 11.1	
Exit codes	**Status**
0	Normal end
1	Warnings Issued During Compilation
2	Errors Occurred During Compilation
3	Fatal Error Caused Termination

Exit code 0 (normal end) indicates that the compilation process was carried out till the end of the file without generating any warnings or errors.

Exit code 1 (warnings) indicates that the compilation process was carried out till the end of the file and warning messages were issued during compilation. There were no errors detected. The output file is created.

Exit code 2 (Errors) indicates that the compilation process was carried out possibly till the end of the file and error messages were generated during compilation. Warnings may or may not have occurred. The output file is not created in this case.

Exit code 3 (Fatal) indicates that a fatal error has led to an abnormal termination of compilation. In this case, output file is not created.

12. ERROR AND WARNING MESSAGES

The error messages given by the compiler fall into three categories :

1. Fatal error messages
2. Error messages
3. Warnings.

The messages for each category are listed below in numerical order, with a brief explanation of each error. All messages give the filename and the line number where the error occurred.

12.1 FATAL ERROR MESSAGES

A Fatal error message indicates a severe problem, one that prevents the compiler from processing the program any further. After displaying the fatal error message, execution is terminated immediately. The following fatal error messages are generated by CCU8 :

12.1.1 Command Line

F0000 Source file not given

The source file for compilation was not given in the command line.

F0001 Invalid filename, '.C' or '.H' extension expected

The filename of the source file had extension other than '.C' or '.H' or '.c' or '.h'.

F0002 Invalid command line option '*option*'

An invalid '*option*' was specified in the command line.

F0003 Directory not specified with /I option

The include directory name was not specified with /I option.

F0004 Filename not specified with /CT option

The calltree filename was not specified with /CT option.

F0005 Type is not specified with /T option

DCL filename was not specified with /T option.

F0006 Constant not specified with /SS option

Stack size constant was not specified with /SS option.

F0007 Constant not specified with /SL option

Maximum identifier length was not specified with /SL option.

F0008 Macro is not specified with /D option

Macro name was not specified with /D option.

F0009 Invalid constant for /SS option

An invalid constant was specified with /SS option.

F0010 Invalid stack size

The constant specified with /SS option must be in the range 0 - 65535, inclusive of both.

F0011 Stack size should be even

Stack size specified with /SS option should be even number.

F0012 Invalid constant for /SL option

An invalid constant or non-constant was specified with /SL option.

F0013 Invalid identifier length

The constant specified with /SL option was not in the range 31 - 254 inclusive of both.

F0014 Duplicate command line option '*option*'

The '*option*' was specified more than once in the command line.

F0015 Duplicate preprocessor option

Both the preprocessor options /LP and /PC were given in the command line.

F0016 Duplicate memory model option

Both the memory model options /MS and /ML were specified in the command line.

F0017 Duplicate data access specifier option

Both the data access specifier options /near and /far were specified in the command line.

F0018 /CT and preprocessor options are mutually exclusive

The option /CT option was specified along with /LP or /PC options.

F0019 /LE and preprocessor options are mutually exclusive

The option /LE was specified along with /LP or /PC options.

F0020 Illegal combination of optimization options

The optimization options were used incorrectly.

F0021 Type is not specified

One of the compulsory options /T was not specified.

F0022 Insufficient memory

The compiler ran out of memory.

F0023 Error in accessing the input file

The compiler was unable to access the input file while compiling.

F0024 Invalid identifier for /D option

An invalid identifier was specified with /D option.

F0025 Invalid identifier for /U option

An invalid identifier was specified with /U option.

F0026 Macro is not specified with /U option

Macro name was not specified with /U option.

F0027 Invalid constant for /W option

An invalid constant was specified with /W option.

F0028 Invalid warning level

The constant specified with /W option must be in the range 0 - 3 inclusive of both.

F0029 Warning level is not specified with /W option

Warning level was not specified with /W option.

F0030 Invalid warning number for /Wc

Warning number(s) specified with /Wc option was invalid warning number(s).

F0031 Warning number is not specified with /Wc option

Warning number(s) was not specified with /Wc option.

F0032 /Zg and preprocessor options are mutually exclusive

/Zg option was specified along with either /LP or /PC options.

F0033 '*identifier*' option cannot be specified after source file name

Command line option other than /I, /Fa, /D, /U, /W was specified after the source file name.

F0034 Unable to open response file 'filename'

The given 'filename' could not be opened.

F0035 Response file not found 'filename'

The given 'filename' either did not exist or was not found.

F0040 CC1U8 is not in the path

CC1U8 executable is not present in the path.

F0041 CC2U8 is not in the path

CC2U8 executable is not present in the path.

F0042 Unable to remove file 'filename'

The given 'filename' could not be removed.

F0043 Unable to read input file

The given 'filename' could not be read.

F0044 Unexpected end of file

End of file is not expected at that position.

F0045 /Fa and preprocessor options are mutually exclusive

The option /Fa was specified along with /LP or /PC options.

F0046 Incompatible version of ‘CC1U8.EXE or CC2U8.EXE’

The product version of CC1U8.EXE or CC2U8.EXE was different from CCU8.EXE.

F0047 Illegal combination of /nofar and /far

Both options /nofar and /far were specified in the command line.

F0050 Unable to open input file ‘*filename*’

The given ‘*filename*’ either did not exist or could not be opened or was not found.

F0051 Unable to open output file

The compiler could not open the output file. This may be due to one of the following reasons :

* The file cannot be opened for lack of space.
* A read-only file with the same name as ‘filename’ already exists.
* The output file path or directory specified with /Fa option does not exist.

F0052 Unable to open list file

The compiler could not open the list file. This may be due to one of the following reasons:

* The file cannot be opened for lack of space.
* A read-only file with the same name as ‘filename’ already exists.

F0053 Unable to open calltree file

The compiler could not open the calltree file. This may be due to one of the following reasons:

* The file cannot be opened for lack of space.
* A read-only file with the same name as ‘filename’ already exists.

12.1.2 General

F1000 File close error

The compiler was unable to close input/output file.

F1001 Internal stack overflow

The processing of the source program has resulted in a overflow of the internal stack in the compiler.

F1002 Internal compiler error

A fault in internal functioning of CCU8.

F1003 Insufficient memory

The compiler ran out of memory.

F1004 Too many errors

The number of errors in the source program have exceeded the compiler maximum limit.

F1005 Floating point overflow

Overflow in floating point arithmetic.

F1006 Divide overflow

Overflow in division/modulus operation for integral type constants.

F1007 Unable to read input file

The compiler was unable to read/access the input file during the compilation process.

12.1.3 Preprocessor

F2000 Bad preprocessor directive '*string*'

'*string*' specified after a '#' is not a valid preprocessor directive.

F2001 Incomplete assembly block

Either the #asm directive was not terminated with a matching #endasm or the #pragma asm directive was not terminated with a matching #pragma endasm.

F2002 Unexpected end of file

The end of the file was encountered unexpectedly.

F2003 Line number exceeds maximum value

Given source file is too big.

F2004 Too many nested ‘#ifxxxx’s

Maximum nesting levels for the directive #ifxxxx exceeded.

F2005 Unable to open include file ‘filename’

The given #include ‘filename’ either did not exist or could not be opened or was not found.

F2006 Integer constant expression expected

A constant expression must be specified with both ‘#if’ and ‘#elif’ directives.

F2007 Path exceeds maximum limit

File path specified in preprocessor directive ‘#include’ could have exceeded the maximum limit.

F2008 ‘#if[n]def’ expected an identifier

An identifier must be specified with the ‘#ifdef’ or ‘#ifndef’ directive.

F2009 ‘#endif’ expected

Before terminating an ‘#if’, ‘#ifdef’ or ‘#ifndef’ directive with a ‘#endif’ directive, end of file was found.

F2010 Parameter buffer overflow

Number of characters in the parameter in a macro could have exceeded the maximum limit.

F2011 Macro buffer overflow

Replacement token string in a macro definition could have exceeded the maximum limit.

F2012 Too many nested include files

Nested #include files exceeded the limit, possible recursion.

F2013 Internal buffer overflow

Macro expansion for a single identifier exceeded the compiler maximum limit.

12.1.4 Lexical

F3000 String too long

Memory not sufficient to hold the complete string literal.

12.1.5 Syntax And Semantic

F4000 Struct/Union nesting too deep

The number of nesting levels of struct/union exceeded the compiler maximum limit.

F4001 Parser stack overflow

The processing of the source program has resulted in a overflow of the parser stack in the compiler.

F4002 Too many nesting levels

The number of nesting levels of control statements (loops/switches/if) exceeded the compiler maximum limit.

F4003 Automatic allocation exceeds 32k

Size of local (stack) variable heap exceeded the maximum limit.

F4004 Unexpected ‘token’

Encountered ‘token’ was used incorrectly.

F4005 Operand stack overflow

The processing of the source program has resulted in a overflow of the operand stack in the compiler.

12.2 ERROR MESSAGES

12.2.1 Preprocessor

E2000 #error : ‘*string*’

The compiler has encountered #error directive and has displayed the given message ‘*string*’.

E2001 ‘##’ cannot occur at the beginning of a macro definition

A macro definition cannot begin with a token pasting operator (##), since a token pasting operator requires two tokens, one before it and one after it.

E2003 Formal parameter missing after '#'

The token following a stringizing operator (#) must be a formal parameter.

E2004 Reuse of formal parameter 'identifier'

The given identifier was used twice in the formal parameter list of a macro definition.

E2005 Invalid line number in '#line' directive

The #line directive encountered a invalid line-number.

E2006 Unexpected in formal list 'token'

The given 'token' was used incorrectly in the formal-parameter list of a macro definition.

E2007 Missing terminator '*character*'

Filename in '#include' directive should be terminated by '>'or '"'.

E2008 Unexpected end of line

The end of line was encountered unexpectedly, in a macro definition.

E2009 '## cannot occur at the end of a macro definition

A macro definition cannot end with a token pasting operator (##), since a token pasting operator requires two tokens, one before it and one after it.

E2010 '#define' syntax

The syntax of the '#define' directive was not correct.

E2011 'defined (*identifier*)' expected

Incorrect use of 'defined' operator.

E2012 '#include' expected a file name, found 'no token'

An #include directive did not specify the required filename specification.

E2013 Double quotes or angle brackets expected after '#include'

The #include directive expects a filename enclosed either in angle brackets (<>) or double quotation marks (“”).

E2014 ‘#line’ syntax

The syntax of the #line directive was not correct.

E2015 ‘#line’ expected a string as a file name

The #line directive did not specify the required filename specification.

E2016 Expected preprocessor command, found ‘*character*’

The given ‘*character*’ followed a number sign (#), but it was not the first letter of a preprocessor directive.

E2017 ‘#undef’ expects an identifier

Macro name was not specified in the #undef directive.

12.2.2 Lexical

E3000 Empty Character constant

The illegal character constant ‘’ was used.

E3001 Too many characters in constant

Character constant containing more than one character or escape sequence was used.

E3002 Constant too big

Integral constant exceeded range.

E3003 Hex constant must have atleast one hex digit

An hexadecimal value after the characters ‘0x’ was missing.

E3004 Unmatched close comment ‘*/’

The compiler might have encountered the closing comment characters ‘*/’ before encountering the opening comment characters ‘/*’.

E3005 Illegal escape sequence

The character(s) after ‘\’ did not form a valid escape sequence.

E3006 Bad octal number ‘token’

While enumerating an octal constant ‘8’/‘9’ could have been encountered.

E3007 Invalid character ‘character’

Encountered invalid character ‘character’.

E3008 Exponent value expected

Exponent was missing after specifying ‘e’/‘E’ in a floating-point number.

E3009 Newline in string

Unexpected end of line in string literal.

E3010 Newline in character literal

A newline character in a character literal.

E3011 Invalid KANJI character

An invalid KANJI character was encountered in strings.

E3012 Bad suffix on number

An invalid suffix was encountered in the number.

12.2.3 Syntactic And Semantic

E4000 More than one storage class specifier

More than one storage class specifier was used in a single declaration statement.

E4001 Unknown size struct/union

An attempt was made to get the size of undefined structure or union.

E4002 Illegal combination of type specifiers

An illegal combination of type specifiers was used in a single declaration statement.

E4003 Function cannot return array

Return value of a function evaluates to an array.

E4004 ‘void’ on variable

Void can be used only to declare pointer variables and functions. It can also come as a formal parameter to a function.

E4005 Redefinition of formal parameter '*identifier*'

The given *identifier* was used twice in the formal parameter list of a function.

E4006 Nonaddress expression

Expression used in initializing an item neither reduce to an lvalue nor a constant.

E4007 Redefinition of variable '*identifier*'

The given *identifier* was defined more than once.

E4008 '*identifier*' not in parameter list

A declaration was made for a formal parameter which was not in the formal parameter list.

E4009 Syntax error : '*token*'

The given '*token*' caused a syntax error.

E4010 Unexpected '*token*'

Encountered '*token*' unexpectedly.

E4011 Function cannot return function

Return value of a function evaluates to a function.

E4012 Array element type cannot be function

Array of functions are not allowed, but array of pointers to functions are allowed.

E4013 Redefinition of struct/union/enum tag '*identifier*'

The given '*identifier*' has already been used for some other structure or union or enum tag.

E4014 Missing subscript

In the definition of an array with multiple subscripts, a subscript value for a dimension other than the first dimension was missing.

E4015 Bit-field must be of type int or char

Bit-fields cannot have a type other than 'int' or 'char'.

E4016 Bit-field cannot have a modified type

Bit fields inside a structure cannot be declared as a pointer or an array or a function.

E4017 Named bit-field cannot have size '0'

A named bit-field inside a structure has size 0. Only unnamed bit-fields can have a size 0.

E4018 Bit-field size out of range

The number of bits specified in the bit field declaration is not in the range of 0-16 inclusive of both for integer bit fields or in the range 0-8 inclusive of both for character bit fields.

E4019 Struct/Union member redefinition '*identifier*'

The '*identifier*' was used for more than one member of the same structure or of the same union.

E4020 Unexpected constant

The given constant was used incorrectly.

E4021 Expected formal parameter list, not a type list

The function body has started after a function declaration statement. The function declaration statement has only type list not formal parameter list.

E4022 Struct/Union too large

The size of structure/union variable exceeded 64k, the compiler limit.

E4023 Value out of range for enum constant

An enumeration constant had a value outside the range of values allowed for type int.

E4024 Cannot use address of automatic variables as static initializer

An attempt was made to initialize a static variable with the address of an automatic variable. Only the address of global or static local or extern variables can be used to initialize static local and global variables.

E4025 Function cannot be a struct/union member

A structure or union member cannot be declared as a function.

E4026 ‘*identifier*’ uses unknown struct/union/enum

The *identifier* was declared as structure/union variable using an undefined structure/union.

E4027 Static function ‘*identifier*’ has no body

A function was declared as a static or inline function and also a call was made but the function was not defined.

E4028 Negative subscript

A value defining an array size was negative.

E4029 Integral constant expression expected

An integral constant expression is expected.

E4030 ‘*identifier*’ already has a body

An attempt was made to define a function body for the function ‘*identifier*’, whose body has been already defined.

E4031 Nonconstant initializer

An Initializer used a non-constant offset.

E4032 Undefined struct/union tag

The identifier was declared as structure/union variable using an undefined structure/union tag.

E4033 Left of ‘*identifier*’ has undefined struct/union

Left operand of .‘*identifier*’ or ->‘*identifier*’ is a struct/union name or a struct/union pointer whose body is not defined.

E4034 Illegal initialization

The initilization expression was illegal.

E4035 Function cannot be initialized

An attempt was made to initialize a function.

E4036 Too many initializers

The number of initializers exceeded the number of objects to be initialized.

E4037 Array initialization needs curly braces

To initialize an array aggregate type, curly braces ({}) are necessary.

E4038 Struct/Union initialization needs curly braces

To initialize an aggregate type, such as struct/union, the initializers must be enclosed within curly braces ({}).

E4039 Same type qualifier is used more than once

Same type qualifier could have appeared more than once in the same specifier list or qualifier list in a declaration, either directly or via one or more typedefs.

E4040 '*identifier*' typedef cannot be used for function definition

Typedef could have occurred in a function definition.

E4041 Invalid subscript

A value defining an array size was zero.

E4042 '*qualifier*' can qualify data only

An object that is not of type data, was qualified with either __near or __far.

E4043 '*qualifier*' can qualify functions only

An object that is not of type function, was qualified with __noreg.

E4044 Segment lost during conversion

An attempt was made to convert a far pointer to near pointer.

E4045 More than one '*qualifier*' qualifier specified

One of the qualifiers __near, __far or __noreg was specified more than once.

E4046 Illegal combination of __near and __far

A variable was qualified with both __near and __far.

E4047 Illegal combination of __near and __huge

A variable was qualified with both __near and __huge.

E4048 Illegal combination of __far and __huge

A variable was qualified with both __far and __huge.

E4049 Huge data can not be referred by far qualified pointer

Huge data can be referred with near qualified pointers only.

E4050 Huge data should be declared with pointer

Huge data should be declared with pointers only.

E4051 Array element type cannot be huge

Array of huge pointers are not allowed.

E4052 Bit-field must have type ‘int’, ‘signed int’ or ‘unsigned int’

A bit-field with type other than ‘int’, ‘signed int’ or ‘unsigned int’ was specified. This error is issued only when /Za option is specified.

E4053 Typename expected

A declaration was specified without any declaration qualifiers. This error is issued only when /Za option is specified.

E4054 ‘*identifier*’ : cannot initialize extern variable within block scope

An extern variable was initialized within a block. This error is issued only when /Za option is specified.

E4055 File must contain atleast one external linkage

A source file was specified without any external linkages. Every file must contain atleast one global object (either a data variable or a function). This error is issued only when /Za option is specified.

E4056 __EI not allowed in function ‘identifier’

The function ‘identifier’ was specified to prohibit multiple interrupt.

E4057 Interrupt function ‘identifier1’ is not allowed in category 1 function ‘identifier2’

The function ‘identifier2’ was specified as category 1 that is to prohibit multiple interrupt. So interrupt / SWI function ‘identifier1’ is not allowed.

E4058 SWI function ‘identifier1’ is not allowed in category 1 function ‘identifier2’

The function ‘identifier2’ was specified as category 1 that is to prohibit multiple interrupt. So interrupt / SWI function ‘identifier1’ is not allowed.

E4059 __far or __huge not allowed with /nofar

One of the qualifiers __far or __huge was specified in the source file. This error is issued only when /nofar option is specified.

E4060 Segment name 'segment name' specified in built-in function is invalid

The string passed as an argument to built-in function is not a valid segment name. Error is issued when string includes at least one character, which is not allowed in segment name.

E4061 Segment count exceeds 65535 limit

The number of segments generated by ccu8 in a compilation unit exceeds 65535.

E4062 Undefined segment name 'segment name'

The Segment name used in built-in functions is not defined within same compilation unit. The segment name used in built-in functions should either be defined by segment / segdef pragma or should be automatically generated by compiler .

E4063 segbase_f not allowed with /nofar

segbase_f function returns void __far * operand. This is not valid when /nofar option is specified.

12.2.4 Expression

E5000 Expression does not evaluate to a function

Operand could have been used like a function but is not a function.

E5001 '*identifier*' is not a function

An attempt was made to define a function body for an '*identifier*' which was not declared as a function.

E5002 '*identifier*' undefined

The given *identifier* was not defined before being used.

E5003 Subscript on non array

A subscript was used on a variable that was not an array.

E5004 '*operator*' : illegal for struct/union

Structure and union type values are not allowed with the given '*operator*'.

E5005 Left of .'*identifier*' must have struct/union type

Left operand of '.' operator should be a struct/union type.

E5006 ‘*identifier*’ is not struct/union member

Identifier to right of ‘.’ or ‘->’ operator is not a member of specified struct/union.

E5007 ‘*operator*’ needs lvalue

The given *operator* did not have lvalue operand.

E5008 Lval specifies ‘const’ object

Identifiers qualified by ‘const’ are non-modifiable. Hence attempt to assign or modify a const specified operand is illegal.

E5009 ‘&’ on register variable

The ‘&’ on a register variable was illegal.

E5010 Left of ->‘*identifier*’ must have struct/union pointer

Left operand of ‘->’ operator should be a struct/union pointer.

E5011 Illegal indirection

The indirection operator (*) was applied to a non-pointer value.

E5012 ‘~’ : bad operand

The operand for the operator ‘~’ was illegal.

E5013 ‘!’ : bad operand

The operand for the operator ‘!’ was illegal.

E5014 ‘unary plus’ : bad operand

The operand for the unary plus was illegal.

E5015 ‘unary minus’ : bad operand

The operand for the unary minus was illegal.

E5016 ‘*operator*’ : bad left operand

The left operand for the specified operator was illegal.

E5017 ‘*operator*’ : bad right operand

The right operand for the specified operator was illegal.

E5018 Pointer ‘+’ non integral value

An attempt was made to add a non-integral value to a pointer.

E5019 ‘+’ : 2 pointers

An attempt was made to add two pointers.

E5020 Pointer ‘-’ non integral value

An attempt was made to subtract a non-integral value from a pointer.

E5021 ‘=’ : left operand must be lvalue

Left operand of ‘=‘ should have lvalue

E5022 ‘&’ on bit-field

An attempt was made to take the address of a bit-field.

E5023 ‘*identifier*’ unknown size

Size of ‘*identifier*’ object was unknown.

E5024 Struct/Union comparison is illegal

Comparison of any two structure or union is not allowed. Individual members of structure or union can be compared.

E5025 Non-integral index

A non-integral expression was used in an array subscript.

E5026 ‘*operator*’: incompatible types

An expression with operands that are not compatible for the operation was encountered, For eg., expression with a pointer and a non-integral operand.

E5027 Illegal index, indirection not allowed

A subscript was applied to an expression that did not evaluate to a pointer.

E5028 Cast to function type is illegal

An object was cast to a function type.

E5029 Cast to array type is illegal

An object was cast to an array type.

E5030 Illegal cast

A type used in a cast operation was not a legal type.

E5031 Unknown size

Size of object was unknown.

E5032 Subscript too large

Subscript value exceeded 65535.

E5033 Size exceeds limit

The size of a object defined exceeds 65535.

E5034 '*identifier*' size exceeds limit

Size of '*identifier*' object exceeds 65535.

E5035 Too few actual parameters

Actual parameters passed to a function could be less than number of parameters formally specified.

E5036 Too many actual parameters

Actual parameters passed to a function could have exceeded number of parameters formally specified.

E5037 Void function returning value

The function was defined to return no value with the 'void' keyword but the function returns a value.

E5038 Illegal sizeof operand

A bit field could have been specified as an operand for sizeof operator.

E5039 '*identifier*' : has bad storage class

The specified storage class cannot be used in the context. For example, the auto storage class specifier cannot be used for variables declared at the external level.

E5040 Parameter has bad storage class

The specified storage class cannot be used in the context.

12.2.5 Control Statements

E6000 Illegal break

A break statement is legal only when it appears within a ‘do’, ‘for’, ‘while’ or ‘switch’ statement.

E6001 Illegal continue

A continue statement is legal only when it appears within a ‘do’, ‘for’, or ‘while’ statement.

E6002 Label ‘*identifier*’ defined more than once

A label ‘*identifier*’ was defined more than once in a function.

E6003 Case ‘*constant*’ already given

The given case value was already used inside the switch statement.

E6004 More than one ‘default’

A switch statement contained more than one ‘default’ keyword.

E6005 Label not defined ‘*identifier*’

A label ‘*identifier*’ used with a ‘goto’ statement was not defined within a function.

E6006 ‘case’ without switch

The ‘case’ keyword can appear only within a switch statement.

E6007 ‘default’ without switch

The ‘default’ keyword can appear only with a switch statement.

E6008 Switch expression is not integral

A switch expression was non-integral.

E6009 Controlling expression has type ‘void’

Conditional expression of a control statement evaluates to a ‘void’.

12.3 WARNING MESSAGES

Warning checks can be limited to a particular level by specifying a constant that represents the warning levels following /W command line option. Warning checks can be disabled with /W0 command line option. The constant that can be specified after /W and their meanings are shown below:

TABLE 12.1	
Level	**Description**
0	Displays no warning messages
1	Displays serious warning messages
2	Displays warning messages that leads to loss of data
3	Displays informative warning messages

If /W option is not specified, 1 is assumed is as default level.

Warning messages and their meanings are listed below. The number shown after the warning number within parentheses represents the warning level.

12.3.1 Preprocessor

W2000(1) '#undef' ignored for predefined macro 'identifier'

An attempt might have been made to undef the predefined macro 'identifier'.

W2001(1) Not enough arguments for macro 'identifier'

The number of actual arguments specified with the given identifier was less than the number of formal parameters given in macro definition of the identifier.

W2002(1) '#define' ignored for predefined macro 'macroname'

An attempt might have been made to install predefined 'macroname' as a macro.

W2003(1) Close bracket expected

Missing ')' in a macro definition or in macro call.

W2004(1) Unexpected characters following directive 'directive'

Extra characters found after processing a preprocessor directive.

W2005(1) Redefinition of macro '*identifier*'

The given identifier was redefined.

W2006(1) Comma separator missing

The formal parameters list in a macro definition must be separated by commas.

W2007(1) Argument expected before '*character*'

An argument was expected in macro call.

W2008(1) Extra arguments ignored for macro '*macroname*'

The number of actual arguments specified with the given *macroname* was greater than the number of formal parameters given in macro definition of the identifier.

12.3.2 Lexical

W3000(1) Identifier truncated to '*identifier*'

The maximum length of an identifier depends upon the value speciefied in /SL option. If /SL option is not specified, maximum of 31 characters are allowed for an identifier. The identifier is truncated to maximum length allowed and extra characters are ignored.

W3001(1) String too long, truncated

The length of the string exceeded 1023 characters.

W3002(3) Possible nested comment

A start comment '/*' is encountered within a comment. Nested comments are not allowed.

12.3.3 Syntactic And Semantic

W4000(1) Auto/Register ignored for global variables

An attempt was made to declare global variable with auto/register storage class.

W4001(1) Formal parameters ignored

The function was declared to take no arguments. But the function definition contains formal parameter declarations, or arguments were given in a call to the function.

W4002(1) ‘const’ ignored on argument

Since function formal parameters are allocated in stack, ‘const’ is ignored on formal parameter. Warning message is not issued when /Za option is specified.

W4003(1) Second parameter list is longer than first

A function was declared more than once with the argument type list in the second declaration longer than the argument type list in the first declaration.

W4004(1) First parameter list is longer than second

A function was declared more than once with the argument type list in the first declaration longer than the argument type list in the second declaration.

W4005(1) ‘const’ ignored for struct/union member ‘identifier’

‘const’ qualified variables are not allowed in struct/union.

W4006(1) Function was declared with formal parameter list

The function was declared to take arguments. But the function definition contains no formal parameter declarations, or no arguments were given in a call to the function.

W4007(1) ‘*identifier*’ : array bound overflows

Too many initializer were present for the array. The excess initializers are ignored.

W4008(1) Parameter *number* declaration different

Type of parameter declaration in prototype could be different from formal declaration.

W4009(1) Declared subscripts for arrays different

Two operands to an operation are arrays whose declared subscripts could be different.

W4010(1) Function was declared with variable arguments

There was a parameter(s) mismatch between prototype and actual definition of a function.

W4011(1) Function was not declared with variable arguments

There was a parameter(s) mismatch between prototype and actual definition of a function.

W4012(1) ‘const’ ignored on local variable ‘*identifier*’

All ‘const’ qualified variables are allocated in the data memory. But local variables are allocated in stack, hence, ‘const’ is ignored on local variables.

W4013(1) No declaration specifiers ; ‘int’ assumed

The variable was declared without any declaration specifiers. Type specifier ‘int’ is assumed for the variable.

W4014(1) Sign information ignored for bit field

A bit field member was decalared as signed.

W4015(1) memory attribute on cast ignored

The memory qualifier in the cast expression is qualiying a non-pointer object.

W4016(1) __near ignored on struct/union member ‘*identifier*’

‘__near’ qualified variables are not allowed in struct/union.

W4017(1) __far ignored on struct/union member ‘*identifier*’

‘__far’ qualified variables are not allowed in struct/union.

W4018(1) __far ignored on local variable ‘*identifier*’

Since local variables are allocated in stack, they cannot be qualified with __far.

W4019(1) __far ignored on argument variable ‘*identifier*’

Since arguments are allocated in stack, they cannot be qualified with __far.

W4020(1) Indirection to different types

Pointers used in the expression were pointing to different memory (Pointer size mismatch).

W4021(1) __noreg ignored for ‘main’

Function ‘main’ was qualified with __noreg.

W4022(1) Missing return value for function ‘*function name*’

The function was declared to return a value, but returns without one.

W4023(1) '*function name*' : no return value

The function 'name' was declared to return a value, but in one of the path, no return statement was found.

W4024(3) Tag '*tag name*' used prior to definition

An attempt was made to refer an undefined structure/union within a function prototype.

W4025(1) Segment lost during conversion

An attempt was made to convert a far pointer to near pointer.

W4026(1) Line splice character encountered in comment line

During line comment processing a line splice character was encountered.

W4027(2) Cast operation may lead to odd boundary access

An attempt was made to cast the pointer to charatcer type (signed or unsigned) to pointer to other data types that need word boundary access.

12.3.4 Expression

W5000(1) '*identifier*' function used as an argument

An attempt was made to pass function as an argument.

W5001(1) Function used as an argument

A formal parameter to a function was declared to be a function, which is not allowed. The formal parameter is converted to a function pointer.

W5002(1) '*operator*' : different levels of indirection

An expression had inconsistent levels of indirection.

W5003(1) Atleast one void operand

An expression with type 'void' was used as an operand.

W5005(1) Constant too large, converted to 'int'

The constant specified in the case statement, exceeded the maximum integer value.

W5006(1) Division by zero

The second operand in a division operation (/) evaluated to zero. Hence it was converted to one.

W5007(1) Mod by zero

The second operand in a remainder operation (%) evaluated to zero. Hence it was converted to one.

W5008(1) '*operator*' : indirection to different types

The indirection operator (*) was used in an expression to access values of different types.

W5009(1) Function Parameter lists differed

The type of the formal parameter did not agree with corresponding type in the function declaration (prototype).

W5010(1) Far pointer truncated to 'int'

A pointer was assigned to an integer variable in large data or large code memory model. The segment address is lost.

W5011(1) Far pointer converted to 'long'

A pointer was assigned to a long variable in large code or large data memory model.

W5012(1) Near pointer converted to 'long'

A pointer was assigned to a long variable in small code or small data memory model. The segment address is made zero.

W5013(1) Parameter mismatch, actual parameter converted

Type in actual parameter declaration was different from formal parameter declaration. Appropriate conversions are performed.

W5015(3) '*operator*' : signed / unsigned mismatch

A mismatch was encountered in sign information between the two operands of a relational or equality operator. Both operands are converted to their corresponding unsigned version.

W5016(3) Switch expression evaluates to 0/1

An expression which results in a boolean value was used as a switch expression.

W5017(1) Local variable '*identifier*' used before initialization

A reference was made to a local variable which was not assigned a value prior to reference.

W5018(2) Unary minus operator applied to unsigned type

Operand of an unary minus operator was of unsigned type. The result of this operation is still unsigned.

W5019(2) '*operator*' : integer constant overflow

An operation involving operators (+, -, *) resulted in an integer constant that overflowed or underflowed the size allocated for it.

W5020(2) Float constant overflow

An operation involving arithmetic operators and conversion to narrower floating type resulted in floating point type constant that overflowed or underflowed the size allocated for it.

W5021(2) '*operator*' : truncation of constant value

An attempt was made to assign a large integral constant to a location with lesser size.

W5022(2) 'type conversion' : possible loss of data

A floating type was converted to an integral type. There may be a possible loss of data.

W5023(2) Conversion between different integral types

Operands of an assignment expression had different integral types and the size of the left operand was less than the size of the right operand.

W5024(2) Conversion between different floating point types

Operands of an assignment expression had different float types and the size of the left operand was less than the size of the right operand.

W5025(3) '*identifier*' : unreferenced formal parameter

A declared formal parameter was never referenced in the body of the function.

W5026(3) Expression is useful only for its side effects

Expression had no effect in the program execution, but useful only for its side effects.

W5027(3) Meaningless use of an expression

Expression does nothing useful.

W5028(3) Assignment within conditional expression

The given control expression contains an assignment operator. It is most likely that assignment operator was used instead of equality operator by mistake.

W5029(3) '*identifier*' : unreferenced local variable

A local variable was not referenced inside a function.

W5030(3) '*function*' : unreferenced static function

A static function was not referenced within the given translation unit.

W5031(3) '*identifier*' : call to function with no prototype

A function call was encountered without any prototype.

W5032(3) '*identifier*' : unreferenced static variable

A local static variable defined within a function was not referenced inside the function or a global static variable defined in a file was not referenced any where in the file.

W5033(2) '*operator*' : value is out of range

An attempt was made to shift a value by a constant which exceeds the bit size of the type of the value.

W5034(3) Relational operation always results in *1/0*

An attempt was made to compare an unsigned value with zero. The result of the comparison is always TRUE or always FALSE.

W5035(3) Expression within 'sizeof' is not evaluated

The given expression within sizeof operator was not evaluated.

W5036(1) Sizeof returns 0

Expression within sizeof operator evaluated to 0.

W5037(1) '*identifier*' : attempt to return address of auto variable

An attempt was made to return the address of an auto variable.

W5038(1) Table data cannot be allocated to physical segment #0

Const qualified variables cannot be allocated in physical segment #0 as ROMWINDOW is not specified.

12.3.5 Controls

W6000(3) Switch statement contains no default label

A switch statement was encountered with no default block.

W6001(3) Unreachable code

Control flow can never reach the indicated line.

W6002(3) '*identifier*' : unreferenced label

A label was not referenced inside the function.

12.3.6 Pragmas

W8000(1) Unknown pragma '*token*'

An invalid keyword was specified with the preprocessor directive '#pragma'.

W8001(1) 'main' cannot be specified in '*pragma keyword*' pragma.

Function 'main' was specified in pragma '*pragma keyword*'.

W8002(1) '*pragma keyword*' pragma variables should be global or static local

The specified variable was neither a global variable nor a static local variable.

W8003(1) Vector address out of range for pragma '*pragma keyword*'

The vector address specified in either Interrupt or Swi pragma was out of range.

The valid range of vector addresses are as follows :

Interrupt- 0x4, 0x8 to 0x7e
Swi - 0x80 to 0xfe

W8004(1) Expected even vector address, for pragma '*pragma keyword*'

An odd vector address was specified in either Interrupt or Swi pragma.

W8005(1) More than one function for the same vector address

Two different functions were specified with same vector address in pragma Interrupt or Swi.

W8006(1) Pragma argument delimiter ‘,’ expected

The pragma argument delimiter ‘,’ (comma) was expected, as /PF option was specified in the command line.

W8007(1) Pragma must appear before function definition

Functions specified in a pragma should not have its body defined before the occurrence of the pragma. This warning message was issued for Interrupt or Swi when the specified function was already defined prior to the pragma directive.

W8008(1) Interrupt function has parameter/return value

Functions specified in pragma Interrupt/Swi either has parameters or returns a value or both.

W8009(1) ‘*pragma keyword*’ address exceeds range

The address specified in Absolute pragma was out of range.

The valid range of address is as follows :

Absolute (data) - 0x0 to 0xffff

W8010(1) Pragma must appear before variable initialization.

The variable specified in pragma was initialized prior to the pragma directive.

W8011(1) Duplicate pragma ‘*pragma keyword*’

Pragma ‘*pragma keyword*’ was specified more than once. This warning message is issued for Stacksize pragma when it is specified more than once in a source file.

W8012(1) Specified stack size out of range

The constant specified in pragma stacksize was out of range. The valid range of stack size is an even number between 0x0 and 0xfffe inclusive of both.

W8013(1) Expected even number as stack size

Size specified with pragma Stacksize was not an even number.

W8014(1) More than one pragma specified for ‘*symbol_name*’

‘*symbol_name*’ was specified in more than one pragma.

W8015(1) '*pragma keyword*' pragma expects function name

The specified symbol was not a function. Interrupt/Swi pragma expects a function name to be specified.

W8016(1) Pragma keyword expected, found no token

No token was found after '# pragma'.

W8017(1) Unexpected characters following pragma '*pragma keyword*'

Unexpected characters were found after a valid pragma '*pragma keyword*'.

W8018(1) Function cannot be specified in pragma '*pragma keyword*'

Variable declared as a function was specified in pragma '*pragma keyword*'.

W8019(1) Enum constants are not allowed in pragma.

An enum constant was specified in pragma directive.

W8020(1) 'Absolute' address leads to odd boundary access

This warning message is issued due to the following reasons:

* An odd address was specified for an initialized variable

W8021(1) Invalid 'Absolute' address for the variable 'token'

The Absolute address specified in the pragma Absolute for the variable 'token' excceded 0xffff.

W8022(1) Interrupt function '*function name*' used in expression

Function '*function name*' specified in Interrupt pragma was used in an expression. Functions specified in this pragma should not be called directly or indirectly in a 'C' program.

W8023(1) Constant expected, found no token

A constant was expected in the #pragma directive, but found no token.

W8024(1) Constant expected, found '*token*'

A constant was expected in the #pragma directive, but found '*token*'.

W8025(1) Pragma syntax error

The specified '#pragma' syntax was not recognized by CCU8.

W8026(1) Variable '*token*' specified in pragma not declared

Variable specified in a pragma was not declared in the file. All the variables specified in pragma should be declared in the file.

W8027(1) Identifier expected for pragma, found no token

An identifier was expected in the '# pragma' directive, but found no token.

W8028(1) Identifier expected for pragma, found '*token*'

An identifier was expected in the '# pragma' directive, but found '*token*'.

W8029(1) Unexpected 'Endasm' pragma ignored

'Endasm' pragma was specified without its correponding Asm pragma.

W8030(1) Identifier or constant expected for pragma, found ','

An Identifier or constant was expected in the '# pragma' directive, but found ','.

W8031(1) Segment number exceeds range

The specified segment number was not in the range 0 to 255 inclusive of both.

W8032(1) Segment should be 0 for 'near' variables

A non-zero segment was specified for near variables.

W8033(1) Identifier expected for pragma, but found ','

An identifier was expected in the '# pragma' directive, but found ',' (comma).

W8034(1) Constant expected for pragma, found ','

A constant was expected in the '# pragma' directive, but found ','.(comma).

W8035(1) '*function name*' specified in 'Inline' pragma is not expanded.

The function '*function name*' specified in inline pragma was not expanded, may be due to one of the following reasons:

* Jumps, labels or loops may be present
* Function was too big to expand
* Function contained variable number of arguments
* Function body contained ASM block
* Function definition preceeded pragma declaration
* If the inline expansion level exceeds inline depth

* If the inline function is called recursively when inline recursion flag is off.

W8036(1) Inline depth out of range

Inline depth specified in Inlinedepth pragma was out of range. The valid range of Inlinedepth is from 0 to 255 inclusive of both.

W8037(1) Invalid Romwin range

The address specified in Romwin pragma was out of range.

The valid range of address is described as follows :
The start_address is greater than or equal to zero
The end_address is greater than the start_address
The maximum value for end_address is 0xffff

W8038(1) Interrupt function '*function name*' declared without static modifier

Function '*function name*' specified in Interrupt pragma was declared without static modifier.

W8039(1) Illegal combination of pragma 'Romwin and Noromwin'

Romwin and Noromwin pragmas are mutually exclusive.

W8040(1) Absolute pragma variable 'identifier' in Romwin area is not const qualified

Absolute pragma variable resides inside Romwin area and it is not qualified with const

W8041(1) Absolute pragma variable 'identifier' outside Romwin area is const qualified

Absolute pragma variable resides outside Romwin area and it is qualified with const

W8042(1) 'const' variables cannot be specified in 'pragma_keyword' pragma

A 'const' qualified variable was specified in pragma 'pragma_keyword'. 'const' qualified variables may be specified only in Absolute pragma.

W8043(1) Mismatch between command line option /NOWIN and pragma ROMWIN

Pragma ROMWIN contradicts with command line option /NOWIN.

W8044(1) Invalid category for Interrupt pragma

The value of category specified with Interrupt pragma was not valid. The valid values for category are 1 and 2.

W8045(1) Invalid stack size

The stack size specified in '#pragma stacksize' was 0.

W8046(3) Segment name 'Name of the segment' is already specified in 'Name of the Pragma' pragma. Segment will be allocated to physical segment '0'

The segment name specified in the current code segment (SEGCODE Or SEGINTR) pragma has already been specified with the other code segment pragma.

This warning will be issued whenever the two code segment, having same names, have a different physical segment attribute. The warning will be issued only when the command line option /ML /W3 is set.

If 'segment name' specified with the __near specifier is the same as one specified with __far. The logical segment will be assigned to physical segment #0 after issuing a warning message.

W8047(1) Segment name 'Name specified in the segment' is already specified in 'Name of the pragma' pragma. Pragma Ignored

The segment name specified in the current segment pragma has already been specified with some other segment pragma.

Note: This warning will not be issued for the following cases:

1.Segcode and Segintr pragmas specify the same name.

2.Seginit and Segnoinit pragmas specify the same name.

W8048(3) Segment name 'Name of the segment' is already allocated to '__near/__far' 'Name of the pragma' segment. Segment will be allocated to physical segment '0'

The name specified in this pragma matches the name specified earlier with the pragma but with a different data access type.

Also Seginit and Segnoinit pragmas can have same names, if the data access specifiers are different then the above warning will be issued.

W8049(1) Segment 'Name of the segment' is already allocated to 'CODE/DATA/TABLE' segment by 'Name of the pragma ' pragma. Pragma Ignored

For a Seginit segment, the name of the table segment is generated by appending the word 'TAB' to the name specified with the pragma.

Example 12.1

```
# pragma Seginit __near “SegA”
```

The name of the table segment will be SegATAB

If there is a clash between the name of the table segment generated by the Seginit pragma and the name specified in a pragma, the above warning will be issued and the later pragma will be ignored.

Note: This warning will not be issued if the Seginit table segment name clashes with the name specified in the Segconst pragma.

W8050(1) Segment 'Segment name' is already allocated to 'Physical Segment [ANY /#0] segment in physical segment ‘[__near/__far]’ by ‘Name of the pragma’ pragma.Segment will be allocated to physical segment 0.

Segconst table segment name and seginit table segment name can be the same, however, if the physical segment attribuite of segconst is near, the above warning will be issued and the segment will be allocated to physical segment 0.

W8051(1) Address out of the range ‘Range’.

The address specified in the pragma is not within the range for the pragma.The range of the pragma depends upon the segment pragma and the data access specifier used.

W8052(1) Address should be outside ROMWIN range 'Rom Window'. Pragma Ignored

The address of the pragma should be outside the rom window , defined by the ROMWIN pragma.

This restriction applies to Seginit, Segnoinit and Segnvdata pragma, address variants.

W8053(3) ROM Window Address not defined

For Segconst pragma address variant, if the ROM window is not defined before the pragma definition , then the above warning message will be issued.

Note: The warning will be issued only when the warning level is set to 3

W8054(1) Segment name ‘Name Specified’ specified in the pragma is an assembler reserved word. Pragma Ignored

If the name specified in the segment pragma is an assembler reserved word then the pragma will be ignored and the above warning will be issued.

W8055(1) Invalid category for Swi pragma

The value of category specified with Swi pragma was not valid. The valid values for category are 1 and 2.

W8056(1) Swi function ‘function name’ already referenced

Function ‘function name’ should not be refered before it is specified in SWI pragma

W8057(1) Swi function ‘function name’ assigned to a pointer to function

Function ‘function name’ is a Swi function which is assigned to pointer to Function

W8058(1) Segment name expected, found comma

The default delimiter of the pragma arguments is ‘blank space’, this behaviour can be changed by specifying the command line option /PF. If /PF is specified then a comma is expected to delimit the arguments for the pragma.

When CCU8 finds a comma in place of ‘blank space’ as delimiter, in a name variant of a segment pragma, and /PF option is not specified, the above warning will be issued.

W8059(1) Segment address expected, found comma

The default delimiter of the pragma arguments is ‘blank space’, this behaviour can be changed by specifying the command line option /PF. If /PF is specified then a comma is expected to delimit the arguments for the pragma.

When CCU8 finds a comma in place of blank space as delimiter, in an address variant of a segment pragma, and /PF option is not specified, the above warning will be issued.

W8060(1) Seginit segment name exceeds the limit ‘ limit ’, the name will be truncated to ‘Number of characters‘ characters

For Seginit segment name the name length of the table segment is taken as the effective length. Since 3 more characters ‘TAB’ are appended to the seginit name for making the table segment name. Hence the maximum length of the name specified in the Seginit pragma can have a maximum length of ‘Maximum Length -3’ , where Maximum Length is defined by /SL command line option, or the default identifier length i:e 31.

W8061(1) Variable 'Variable Name' is already allocated to default segment

Variable 'Variable Name' is first allocated to default segment, and then is in the effect of user defined segment. Variable will remain in default segment. First occurrence of the variable is initialized and second occurrence of the variable is uninitialized.

W8062(1) Variable 'Variable Name' is already allocated to ‘Segment’ segment with '__near/__far' specifier

Variable 'Variable Name' is first allocated to a user defined segment 'Segment1', and then is in the effect of any other user defined segment 'Segment2'. Variable will remain in the first allocated user defined segment ‘Segment1’ . Either both the occurrence of the variable are uninitialized or only second occurrence is uninitialized.

W8063(1) Variable 'Variable Name' is reallocated to default segment

Variable 'Variable Name' is first alocated to user defined segment name, and then is in the effect of default segment. Variable will be reallocated to default segment. First occurrence of the variable is uninitialized and second occurrence of the variable is initialized.

W8064(1) Variable 'Variable Name' is reallocated to 'Segment1' segment with '__near/far' specifier

Variable 'Variable Name' is first allocated to a user defined segment 'Segment1', and then is in the effect of any other segment 'Segment2'. Variable will be reallocated to the later user defined segment ‘Segment2’. First occurrence of the variable is uninitialized and second occurrence of the variable is initialized.

W8065(1) Variable 'Variable Name' is already allocated to the segment starting at the address 'Segment Address'

Variable 'Variable Name' is first allocated to user defined segment starting at the address 'Segment Address', and then is in the effect of any other user defined segment 'Segment2'. Variable will remain in the first allocated segment ‘Segment1’. Either both the occurrence of the variable are uninitialized or only 2nd occurrence is uninitialized.

W8066(1) Variable 'Variable Name' is reallocated to the segment starting at the address 'Segment Address'

Variable 'Variable Name' is first allocated to the segment starting at the user defined address ‘Segment Address’ and then is in the effect of any other user defined segment 'Segment'. Variable will be reallocated to later user defined

segment ‘Segment’. First occurrence of the variable is uninitialized and second occurrence of the variable is initialized.

W8067(1) Address should be within the ROMWIN range 'romwin range’.Pragma ignored.

The address specified with the segconst pragma should be within the romwindow specified earlier in the source file using the romwin pragma, The warning will be issued for addresses in physical segment 0.

W8068(1) Absolute address for 'Variable Name' crosses physical segment boundary.

'Variable Name' allocated to address 'Segment number : 0x0000'

Address for assigning a particular variable crosses the physical segment boundary. The variable will be assigned to the 0th offset of the same physical segment.

W8069(1) Duplicate absolute 'Data/code' segment definition. Pragma ignored

Same address is specified either in more than one data pragmas or in more than one code pragmas.

W8070(1) Argument 'Argument' specified in optimization pragma is invalid. Pragma ignored

The argument specified with the optimization pragma should be one of the following ‘Od’ ,’Om’ ,’Ot’ ,’Og’ ,’Ol’ , ‘Oa’ or ‘default’. If an option other than the above seven are specified the above warning will be issued. Please note that the options are case sensitive.

W8071(1) Invalid segment type 'Segment Type' in SEGDEF pragma. Pragma ignored

Segment type specified on Segdef Pragma is not among ‘DATA’, ‘CODE’, ‘NVDATA’ and ‘TABLE’

W8072(1) Argument 'Od' cannot be specified with any other arguments. Pragma ignored.

This warning will be issued if the Od option is specified simultaneously with some other optimization option, in the Optimization pragma.

W8073(1) Argument 'default' cannot be specified with any other arguments. Pragma ignored

This warning will be issued if the ‘default’ option is specified simultaneously with some other optimization option

W8074(1) Optimization control pragma cannot be specified within a function body. Pragma ignored

This warning will be issued for the optimization pragmas that are specified within the function body. The pragma will be ignored.

W8075(1) Arguments 'Om' and 'Ot' cannot be specified concurrently. Pragma ignored.

This warning will be issued if both the arguments ‘Om’ and ‘Ot’ are specified simultaneously in the optimization pragma. The pragma definition will be ignored after issuing the warning.

W8076(1) Segment type redefinition for 'Segment Name' in 'Pragma Name' pragma. Pragma ignored.

Segment name specified with Segdef pragma matches with other segment name having different segment type. Other segment may be either default segment or user specified segment through any of the Segment pragmas.

W8077(1) Segment type expected, found comma

Comma is used as a delimiter instead of blank space in Segdef pragma definition, and /PF option is not specified