OKI

CC665S

User's Manual

Program Development Support Software

FIRST EDITION
ISSUE DATE:Mar. 1999

NOTICE

1. The information contained herein can change without notice owing to product and/or technical improvements. Before using the product, please make sure that the information being referred to is up-to-date.
2. The outline of action and examples for application circuits described herein have been chosen as an explanation for the standard action and performance of the product. When planning to use the product, please ensure that the external conditions are reflected in the actual circuit, assembly, and program designs.
3. When designing your product, please use our product below the specified maximum ratings and within the specified operating ranges including, but not limited to, operating voltage, power dissipation, and operating temperature.
4. OKI assumes no responsibility or liability whatsoever for any failure or unusual or unexpected operation resulting from misuse, neglect, improper installation, repair, alteration or accident, improper handling, or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified operating range.
5. Neither indemnity against nor license of a third party's industrial and intellectual property right, etc. is granted by us in connection with the use of the product and/or the information and drawings contained herein. No responsibility is assumed by us for any infringement of a third party's right which may result from the use thereof.
6. The products listed in this document are intended for use in general electronics equipment for commercial applications (e.g., office automation, communication equipment, measurement equipment, consumer electronics, etc.). These products are not authorized for use in any system or application that requires special or enhanced quality and reliability characteristics nor in any system or application where the failure of such system or application may result in the loss or damage of property, or death or injury to humans. Such applications include, but are not limited to, traffic and automotive equipment, safety devices, aerospace equipment, nuclear power control, medical equipment, and life-support systems.
7. Certain products in this document may need government approval before they can be exported to particular countries. The purchaser assumes the responsibility of determining the legality of export of these products and will take appropriate and necessary steps at their own expense for these.
8. No part of the contents contained herein may be reprinted or reproduced without our prior permission.
9. MS-DOS is a registered trademark of Microsoft Corporation.

CC665S User's Manual

Part 1. CC665S Ver.2.01 User Guide
Part 2. CC665S Ver.2.01 Language Reference

Part1.
CC665S Ver.2.01
User Guide

Table Of Contents

1. OVERVIEW

The C language is a powerful general purpose programming language that can generate efficient, compact and portable code. C is manageable because of its small size, flexible because of its ample supply of operators and powerful in its utilization of modern control flow and data structures.

CC665S is an optimizing C Compiler. It incorporates the features that are fundamental to the ‘C’ language and that exist in most C compilers.

Salient features of CC665S are listed below:

1. ‘C’ language supported by CC665S 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. Facilities to access any memory location in RAM and ROM are provided.
7. Mixed language programming can be done.

2. OPERATING ENVIRONMENT

2.1 HARDWARE AND MEMORY REQUIREMENT

MACHINES	:	NEC PC-9801 series, IBM-AT compatible and clones
OPERATING SYSTEM	:	MS-DOS Ver. 5.0 and above
PC MEMORY	:	640K with at least 384K extended RAM

2.2 SYSTEM CONFIGURATION

CC665S requires the following information to be included in the **CONFIG.SYS** file.

```
files=20
```

This information must be added to the CONFIG.SYS file before invoking CC665S. It allows the compiler to open atlas 20 files at a time.

2.3 ENVIRONMENT VARIABLES

CC665S uses two environment variables INCL66K and TMP.

INCL66K can be used to specify a directory to search the include files, specified with #include preprocessor directive.

The INCL66K environment variable can be defined using the DOS command SET. The SET command has the following format:

SET INCL66K=path

CC665S 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 **CC665S** to create temporary files during compilation in the given **path**.

SET TMP=PATH

Example :

SET TMP=C:\RAMDRIVE
CC665S 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.

3. INVOKING CC665S AND COMMAND LINE OPTIONS

3.1 INVOCATION OF CC665S

CC665S may be invoked by specifying the following command line:

```
C:\> CC665S [options....] file [file ....] <CR>
```

Each filename is an input 'C' source file and the name should have either ".C", ".c", ".H" or ".h" extension. If CC665S encounters any other extension a fatal error message is issued and the compilation process is terminated. The *file* may have pathname.

CC665S creates an assembly file as output for each of the files specified in the command line. The output file contains MSM66K "500" core assembly mnemonics or assembly mnemonics that are valid only in "500S" core, depending on the core option specified in the command line.

By default, the output file has the same name as the input file with an extension ".asm". The output file is always created in the current working directory even if the input file resides in some other directory.

Following are the command line options:

/T	specify the operand string for TYPE instruction
/MS	small C memory model
/MEM	effective medium C memory model
/MM	medium C memory model
/MC	compact C memory model
/MEL	effective large C memory model
/ML	large C memory model
/mixC	compact mixed memory model
/mixL	large mixed memory model
/mixM	medium mixed memory model
/Ot	optimize for speed
/Ol	enable loop optimizations
/Om	optimize for space
/Og	enable global optimizations
/Od	disable optimizations
/Oa	ignore aliasing
/nX500	generate code for nX-8/500
/nX500S	generate code for nX-8/500S
/LE	generate list file
/Fa	assembly listing file
/CT	list calltree in a file
/LP	preprocessed output in a file
/I	include file directory
/PC	preprocessed output with comment
/D	define macro
/ST	generate stack probe routine
/SS	change stack size
/SD	generate debug information with 'calls menu' option enabled
/OSD	generate debug information with 'calls menu' option disabled
/SL	change maximum identifier length
/J	default **char** type is **unsigned**
/PF	use comma as delimiter for pragma arguments
/REG	use register for argument/ return value
/WIN	assign window attribute to table
/AWIN	assign awin attribute to table
/SYS	change compiler segment naming strategy

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

On invocation, following copyright message is displayed.

```
CC665S C Compiler, Ver.2.01 Apr 1996
Copyright (C) 1992, Oki Electric Ind. Co., Ltd.
```

For the command line,

```
C:\> CC665S <CR>
```

the following usage is displayed.

```
CC665S C Compiler, Ver.2.01 Apr 1996
Copyright (C) 1992, Oki Electric Ind. Co., Ltd.

Usage: CC665S /T string [option ...] filename...
        /T string Specify the operand string for TYPE instruction

                              -C MEMORY MODEL-
/MS small model                        /MC compact model
/MEM effective medium model            /MEL effective large model
/MM medium model                       /ML  large model
                           -MIXED MEMORY MODEL-
/mixC compact model                    /mixL large model
/mixM medium model
                               -OPTIMIZATION-
/Ot optimize for speed                 /Ol enable loop optimizations
/Om optimize for space                 /Og enable global optimizations
/Od disable optimizations              /Oa ignore aliasing
                             -CODE GENERATION-
/nX500 generate code for nX-8/500      /nX500S generate code for nX-8/500S
                               -OUTPUT FILES-
/LE generate list file                 /Fa[filename] assembly listing file
/CT <filename> list calltree in a file
(Press <return> to continue)
                              -PREPROCESSOR-
/LP preprocessed output in a file      /I <directory> include file directory
/PC preprocessed output with comments  /D<identifier>[=[string]] define macro
                                  -STACK-
/ST generate stack probe routine       /SS <constant> change stack size
                                  -DEBUG-
/SD generate debug information with ‘calls menu’ option enabled
/OSD generate debug information with ‘calls menu’ option disabled
```

-MISCELLANEOUS-

/SL<constant> change maximum identifier length
/J default char type is unsigned
/PF use comma as delimiter for pragma arguments
/REG use register for argument/ return value
/WIN assign window attribute to table
/AWIN assign awin attribute to table
/SYS change compiler segment naming strategy

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 and /D may be specified more than once in the command line. If any option other than /I, /Fa or /D is specified more than once in the command line, CC665S issues fatal error message. Options /Fa and /D 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 CC665S. CC665S 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:\> CC665S /T m66589 test.c  <CR>
```

/T m66589 in the above example, instructs CC665S to output TYPE pseudo instruction as follows:

```
type (m66589)
```

3.2.2 'C' Memory Model Options

CC665S supports the following 'C' memory models:

1. Small C memory model
2. Effective medium C memory model
3. Medium C memory model
4. Compact C memory model
5. Effective large C memory model
6. Large C 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, CC665S issues a fatal error message. The C memory model options are described in detail in this section.

3.2.2.1 /MS OPTION

Syntax : /MS

/MS option instructs CC665S to compile programs in small C memory model. The small C memory model uses one physical data segment for data variables and one physical code segment for both functions and tables. This option is the default C memory model option. If no memory model option is specified in the command line, programs are compiled in this model.

Example 3.2

```
C:\> CC665S /T m66589 /MS test1.c <CR>
```

The command line option /MS in the above example, instructs CC665S to compile the source file "test1.c" in small memory model.

3.2.2.2. /MEM OPTION

Syntax : /MEM

/MEM option instructs CC665S to compile programs in effective medium C memory model. The effective medium C memory model uses one physical data segment for data variables, one physical code segment for functions and one separate physical code segment for tables. This memory model is an extension of small memory model.

Example 3.3

```
C:\> CC665S /T m66589 /MEM test2.c <CR>
```

The command line option /MEM in the above example, instructs CC665S to compile the source file “test2.c” in effective medium memory model.

3.2.2.3 /MM OPTION

Syntax : /MM

/MM option instructs CC665S to compile programs in medium C memory model. The medium C memory model uses one physical data segment for data variables and one or more physical code segments for functions. In this model, tables are allocated in one of the physical code segments used by functions.

Example 3.4

```
C:\> CC665S /T m66589 /MM test3.c <CR>
```

The command line option /MM in the above example, instructs CC665S to compile the source file “test3.c” in medium memory model.

3.2.2.4 /MC OPTION

Syntax : /MC

/MC option instructs CC665S to compile programs in compact C memory model. The compact C memory model uses one or more physical data segments for data variables and one physical code segment for both functions and tables.

Example 3.5

```
C:\> CC665S /T m66589 /MC test4.c <CR>
```

The command line option /MC in the above example, instructs CC665S to compile the source file “test4.c” in compact memory model.

3.2.2.5 /MEL OPTION

Syntax : /MEL

/MEL option instructs CC665S to compile programs in effective large C memory model. The effective large C memory model uses one or more physical data segments for data variables, one physical code segment for functions and one separate physical code segment for tables. This memory model is an extension of compact memory model.

Example 3.6

```
C:\> CC665S /T m66589 /MEL test5.c <CR>
```

The command line option /MEL in the above example, instructs CC665S to compile the source file "test5.c" in effective large memory model.

3.2.2.6 /ML OPTION

Syntax : /ML

/ML option instructs CC665S to compile programs in large C memory model. The large C memory model uses one or more physical data segments for data variables and one or more physical code segments for functions. In this model, tables are allocated in one of the physical code segments used by functions.

Example 3.7

```
C:\> CC665S /T m66589 /ML test6.c <CR>
```

The command line option /ML in the above example, instructs CC665S to compile the source file "test6.c" in large memory model.

Example 3.8

```
C:\> CC665S /T m66589 /MC /MM test.c <CR>
```

For the above command line option, CC665S issues a fatal error because more than one memory model option is specified.

3.2.3 Mixed Memory Model Options

Mixed memory model options specify the available memory in the hardware. Hardware supports the following memory models:

1. Small memory model
2. Medium memory model
3. Compact memory model
4. Large memory model

The mixed memory model options are described in this section.

3.2.3.1 /mixM OPTION

/mixM option instructs the compiler that the hardware supports medium hardware memory model. Medium hardware memory model contains one physical data segment and more than one physical code segment.

Example 3.9

```
C:\>CC665S /T m66589 /MEM /mixM test2.c <CR>
```

For the above command line, CC665S compiles “test2.c” in medium mixed memory model.

3.2.3.2 /mixC OPTION

/mixC option instructs the compiler that the hardware supports compact hardware memory model. Compact hardware memory model contains more than one physical data segment and one physical code segment.

Example 3.10

```
C:\>CC665S /T m66589 /MS /mixC test3.c <CR>
```

For the above command line, CC665S compiles “test3.c” in compact mixed memory model.

3.2.3.3 /mixL OPTION

/mixL option instructs the compiler that the hardware supports large hardware memory model. Large hardware memory model contains more than one physical data segment and more than one physical code segment.

Example 3.11

```
C:\>CC665S /T m66589 /MS /mixL test3.c <CR>
```

For the above command line, CC665S compiles “test3.c” in large mixed memory model.

3.2.4 ‘C’ And Mixed Memory Model Combination

A mixed memory model option may be specified with a C memory model option. If it is specified without a C memory model option, CC665S issues a fatal error message. Some combinations of C and mixed memory models are not allowed.

The valid and invalid combinations of C and mixed memory models are shown in the following table:

TABLE 3.1			
	/mixM	**/mixC**	**/mixL**
/MS	Valid	Valid	Valid
/MEM	Valid	Invalid	Valid
/MM	Valid	Invalid	Valid
/MC	Invalid	Invalid	Valid
/MEL	Invalid	Invalid	Valid
/ML	Invalid	Invalid	Valid

In the absence of a mixed memory model option in the command line, mixed memory model is set based on C memory model option. The following table specifies how C and mixed memory models are assumed based on C and mixed memory model options specified in the command line:

TABLE 3.2

MEMORY MODEL OPTIONS	C MEMORY MODEL	MIXED MEMORY MODEL
none	small	small
/MS	small	small
/MS /mixM	small	medium
/MS /mixC	small	compact
/MS /mixL	small	large
/MEM /mixM	effective medium	medium
/MEM	effective medium	medium
/MEM /mixL	effective medium	large
/MM	medium	medium
/MM /mixM	medium	medium
/MM /mixL	medium	large
/MC	compact	compact
/MC /mixL	compact	large
/MEL	effective large	large
/MEL /mixL	effective large	large
/ML	large	large
/ML /mixL	large	large

3.2.5 Optimization Options

The optimizing capabilities available with CC665S can reduce the target storage space and/or target execution time. This is achieved by eliminating unnecessary instructions and rearranging the code.

By default, CC665S performs all optimizations. One of the following optimization options may be used to suppress the optimization or to control the optimization.

The various optimization options are shown in the following table:

TABLE 3.3	
OPTION	**OPTIMIZING PROCEDURE**
/Od	Disables optimization
/Ol	Enables loop optimization
/Og	Enables global optimization
/Oa	Enables alias checks
/Om	Maximizes optimization
/Ot	Speed optimization

Following optimizations are performed by default.

1. Common subexpression elimination
2. Constant folding.
3. Peephole optimizations.

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

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

TABLE 3.4
1. Eliminating dead code
2. Eliminating dead blocks
3. Optimizing jumps
4. Optimization using algebraic identities
5. Global Register allocation and assignment

Other options have no control over these optimizations.

3.2.5.1 /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 3.12

```
C:\> CC665S /Od /T m66589 test.c <CR>
```

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

Example 3.13

```
C:\> CC665S /Od /Ol /T m66589 test.c <CR>
```

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

3.2.5.2 /Og OPTION

Syntax : /Og

When /Og option is specified, CC665S performs only the global optimizations. The global optimizations performed by CC665S are

1. Global common subexpression elimination
2. Global constant folding
3. Code sinking
4. Code hoisting.

/Og option enables CC665S to perform common subexpression elimination and constant folding by examining an entire function.

Example 3.14

```
C:\> CC665S /Og /T m66589 test.c <CR>
```

Loop optimizations and alias checks are not performed for the above command line. However, other optimizations shown in table 3.4 are performed.

3.2.5.3 /Ol OPTION

Syntax : /Ol

When /Ol option is specified, CC665S performs only those optimizations that involve loops.

Loops involve sections of code that are executed repeatedly. These sections of code are targets for optimization. These optimizations involve moving code or rearranging code so that it executes faster.

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 3.15

```
C:\> CC665S /Ol /T m66589 test.c <CR>
```

/Ol option enables CC665S to perform loop optimizations. Global optimizations and alias checks are not performed for the above command line. However, other optimizations shown in table 3.4 are performed.

Example 3.16

```
C:\> CC665S /Ol /Og /T m66589 test.c <CR>
```

/Ol option enables CC665S to perform loop optimizations and /Og enables global optimizations. Alias checking is not performed for the above command line. Other optimizations shown in table 3.4 are performed.

3.2.5.4 /Oa OPTION

Syntax : /Oa

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

Aliases are multiple names (that is, symbolic references) for the same memory location in a program. When /Oa option is specified, CC665S 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 3.17

```
C:\> CC665S /Oa /T m66589 test.c <CR>
```

/Oa option enables CC665S to perform alias checks. Loop optimizations and global optimizations are not performed for the above command line. Optimizations shown in table 3.4 are performed.

3.2.5.5 /Om OPTION

Syntax : /Om

/Om option enables CC665S to perform maximum possible optimizations. When /Om option is specified, CC665S 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 3.18

```
C:\> CC665S /Om  /T m66589 test.c <CR>
```

/Om option enables CC665S to perform all the optimizations iteratively. Alias checks are not performed.

Example 3.19

```
C:\> CC665S /Om /Og /T m66589 test.c <CR>
```

/Om option enables CC665S to perform all optimizations iteratively. /Og option in the above command line is redundant since global optimizations are also performed because of /Om option.

3.2.5.6 /Ot OPTION

Syntax : /Ot

/Ot option enables CC665S to perform optimization for speed. This also enables CC665S to perform global optimization, 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 3.20

```
C:\> CC665S  /Ot  /T m66589 test.c <CR>
```

/Ot option enables CC665S to perform optimization for speed.

3.2.5.7 SUMMARY OF OPTIMIZATION OPTIONS

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

TABLE 3.5

Optimization Options		**Loop Optimizations**	**Global Optimizations**
Default	no /Oa	performed - unsafe	performed - unsafe
	/Oa	performed - safe	performed - safe
/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.

3.2.5.8 COMBINATION OF OPTIMIZATION OPTIONS

The following table summarizes the combination of the optimization options.

TABLE 3.6

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

No.	Combinations	Validity	Optimizations Performed If The Combination Is Valid
14.	/Oa /Om	Valid	Loop Optimizations Global Optimizations Alias Checking Other Optimizations
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

3.2.6 Code Generation

3.2.6.1 /nX500 OPTION

Syntax : /nX500

The /nX500 option instructs CC665S to generate code for nX-8/500 core.

Example 3.21

```
C:\> CC665S /nX500 /T m66589 test.c <CR>
```

This option cannot be specified along with the other core option (/nX500S). If specified, CC665S issues a fatal error message “Duplicate core option”.

Example 3.22

```
C:\> CC665S /nX500S /nX500   test.c <CR>
```

A fatal error message is generated for the above command line, since the core options /nX500 and /nX500S are mutually exclusive.

3.2.6.2 /nX500S OPTION

Syntax : /nX500S

The option /nX500S instructs CC665S to generate code for nX-8/500S core. This is the default option. If no core is specified in the command line, CC665S generates code for nX-8/500S core.

Example 3.23

```
C:\> CC665S /nX500S /T m66589 test.c <CR>
```

This option cannot be specified along with the other core option (/nX500). If specified, CC665S issues a fatal error message : “Duplicate core option”.

Example 3.24

```
C:\> CC665S /nX500 /nX500S   test.c <CR>
```

A fatal error message is generated for the above command line, as the core options /nX500 and /nX500S are mutually exclusive.

3.2.7 Output Files

3.2.7.1 ERROR LISTING OPTION

Syntax : /LE

/LE option enables CC665S 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 ".LST".

This option cannot be specified along with one 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 3.25

```
C:\> CC665S /LE  /T m66589 test.c <CR>
```

CC665S generates a listing file "test.lst" for the above command line. The output listing file contains the source with line numbers and the generated errors, if any.

3.2.7.2 CALLTREE OPTION

Syntax : /CT filename

This option enables CC665S to generate a listing of function calls. Calltree 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 CC665S issues an error message. No default extension is assumed by CC665S.

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

Example 3.26

```
C:\> CC665S /CT test.cal /T m66589 test.c <CR>
```

/CT option in the above command line enables CC665S to generate a calltree listing file "test.cal". An indented listing of function names and the calls in each function is output in "test.cal".

Example 3.27

```
C:\> CC665S /CT test.cal /T m66589 t1.c t2.c <CR>
```

/CT option in the above command line enables CC665S 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 3.28

```
C:\> CC665S /CT test.cal /LP test.c <CR>
```

A fatal error is issued by CC665S for the above command line, since /CT and /LP options are mutually exclusive.

3.2.7.3 ASSEMBLY LISTING FILE

Syntax : /Fa[path]

/Fa option enables CC665S 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, CC665S 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 3.29

```
C:\> CC665S /Fa..\ /T m66589 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 3.30

```
C:\> CC665S /Fad:\asm\ /T m66589 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 3.31

```
C:\> CC665S /Faout1 /Faout2 /T m66589 test.c <CR>
```

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

3.2.8 Preprocessor Options

3.2.8.1 /LP OPTION

Syntax : /LP

This option enables CC665S to generate listings of preprocessed output of each of the input files. When this option is specified CC665S 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 “.P66”. When this option is specified, source files are not compiled.

List file option (/LE) and calltree option (/CT) cannot be specified with /LP option. The input filename may have any extension (including empty extension) when this option is specified.

Example 3.32

```
C:\> CC665S /LP test.c <CR>
```

/LP in the above example instructs the compiler to create a preprocessed output file test.p66. Comments will be stripped.

Example 3.33

```
C:\> CC665S /LP /LE test.c <CR>
```

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

3.2.8.2 /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. CC665S 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, CC665S issues a fatal error "Duplicate preprocessor option".

The name of the preprocessed file is the same as input file but with an extension ".P66". When this option is specified, source files are not compiled.

List file option (/LE) and calltree option (/CT) cannot be specified with /PC option. The input filename may have any extension (including empty extension) when this option is specified.

Example 3.34

```
C:\> CC665S /PC test10.inp <CR>
```

/PC in the above example instructs the compiler to create a preprocessed output file test10.p66. Comments are preserved in the output file.

Example 3.35

```
C:\> CC665S /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.

3.2.8.3 /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 INCL66K. CC665S searches the directory specified in this option first, before searching the standard places given in INCL66K environment variable.

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

Example 3.36

```
C:\> CC665S /I \include /T m66589 test.c <CR>
```

The above command line instructs CC665S to search the included files in the directory "\include" before searching in the directories specified using the environment variable INCL66K.

Example 3.37

```
C:\> CC665S /I include /I lib /LP test.c <CR>
```

For the above command line, CC665S searches the included file in the directory "include" first. And if not found, it searches in the directory "lib". If still not found, CC665S searches in the directories specified by the environment variable INCL66K.

3.2.8.4 /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 CC665S ignores /D option without giving warning message.

Example 3.38

```
C:\> CC665S  /Tm66589 /DVALUE(a) test14.c <CR>
```

For the above command line, CC665S ignores /D option 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 3.39

```
C:\> CC665S  /Tm66589  /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'.

3.2.9 Stack

3.2.9.1 STACK SIZE OPTION

Syntax : /SS constant

/SS option sets the size of the program stack. CC665S outputs the size specified in this option using the pseudo instruction STACKSEG. This enables the linker RL66K, at a later stage, to allocate memory for the program stack.

If this option is not specified, CC665S sets a default stack size of 1024 bytes.

The constant must be a decimal constant. The valid range of the constant is between 2 and 65534, inclusive of both. If an odd number is specified, a fatal error is issued. The space between /SS and the constant is optional.

Example 3.40

```
C:\> CC665S /SS 2048  /T m66589 test.c <CR>
```

STACKSEG pseudo instruction with size 2048 is output by CC665S for the above command line.

Example 3.41

```
C:\> CC665S /SS0x0800   /T m66589 test.c <CR>
```

A fatal error is issued by CC665S for the above command line, since only a decimal constant is expected as a parameter for /SS option.

Example 3.42

```
C:\> CC665S /SS 1023  /T m66589 test.c <CR>
```

A fatal error is issued for the above command line, since an even number is expected as a parameter for /SS option.

3.2.9.2 STACK CHECKING OPTION

Syntax : /ST

When /ST option is specified, stack probes are added in the assembly output by CC665S.

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 3.43

```
C:\> CC665S /ST /T m66589 test.c <CR>
```

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

3.2.10 Debugging Options

3.2.10.1 /SD OPTION

Syntax : /SD

If /SD option is specified, CC665S generates the necessary information for the 'C' source level debugger CDB665S. Files compiled without /SD or /OSD option cannot be debugged using the debugger CDB665S at source level.

Debugging information are stored in a separate file. The name of the debugging information file is the same as input file with an extension ".DBG".

This option cannot be specified along with the other debugging option /OSD. If specified, fatal error is issued.

Example 3.44

```
C:\> CC665S /SD /T m66589 test.c <CR>
```

For the above command line, a debug information file is created. The name of the debug information file is “test.dbg”. This file will not be created when CC665S issues error message or a fatal error message.

Example 3.45

```
C:\> CC665S /SD /OSD test.c <CR>
```

A fatal error is generated for the above command line, since /SD and /OSD options are mutually exclusive.

3.2.10.2 /OSD OPTION

Syntax : /OSD

This option is same as /SD option except that the source level debugger CDB665S does not support ‘calls menu’ option if /OSD option is used in compilation. /SD and /OSD options are mutually exclusive.

If both /SD and /OSD options are specified together, fatal error is issued.

Example 3.46

```
C:\> CC665S /OSD /T m66589 test.c <CR>
```

For the above command line, a debug information file is created. The name of the debug information file is “test.dbg”. This file will not be created when CC665S issues error message or a fatal error message.

Example 3.47

```
C:\> CC665S /OSD /SD test.c <CR>
```

A fatal error is generated for the above command line, since /SD and /OSD options are mutually exclusive.

3.2.11 Miscellaneous Options

3.2.11.1 /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, CC665S assumes the maximum length of an identifier as 31.

Example 3.48

```
C:\> CC665S /SL 40  /T m66589 test.c <CR>
```

In the above example, CC665S 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 3.49

```
C:\> CC665S /T m66589 test.c <CR>
```

In the above example, CC665S takes maximum length of an identifier as 31 characters (default maximum identifier length).

Example 3.50

```
C:\> CC665S /SL 1023  /T m66589 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 3.51

```
C:\> CC665S /SL /T m66589 test.c <CR>
```

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

3.2.11.2. /J OPTION

Syntax : /J

/J option instructs CC665S to treat default ‘**char**’ type as ‘**unsigned char’** type. If /J option is specified in the command line, CC665S treats all ‘**char**’ type without ‘**signed’** specifier as ‘**unsigned char’** type.

Example 3.52

```
char chr ;
```

By default, CC665S treats the variable ‘chr’ as ‘**signed char’** type. If /J option is specified in the command line, CC665S treats the variable ‘chr’ as ‘**unsigned char’** type.

3.2.11.3 /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 3.53

```
#pragma inpage  seg1  int_var1,  int_var2,  int_var3
```

The above pragma syntax is valid if /PF is specified in the command line. Otherwise, CC665S issues a warning message and ignores pragma.

3.2.11.4 /REG OPTION

Syntax : /REG

/REG option instructs the compiler to treat all the functions except **‘__noacc’** specified functions as ‘**__accpass**’ functions. The first argument and return value may be passed through the accumulator to and from **‘__accpass’** function.

Example 3.54

```
C:\> CC665S /REG /T m66589 test.c <CR>
```

In the above example, all the functions that are not qualified with ‘**__noacc**’ will be assumed to be qualified with ‘**__accpass**’ qualifier.

3.2.11.5 /WIN OPTION

/WIN option directs the compiler to assign all non-far tables to ROMWINDOW region. Data memory addressing is used to access non-far tables.

Example 3.55

```
C:\> CC665S /WIN /T m66589 test.c <CR>
```

In the above example, all the non-far tables in source file "test.c" are allocated in ROMWINDOW region.

3.2.11.6 /AWIN OPTION

/AWIN option instructs the compiler to output the pseudo instruction "awin" in the assembly listing. /AWIN option may be used while compiling library routines as they may be invoked from both programs compiled using /WIN option and programs compiled without /WIN option.

/WIN and /AWIN options are mutually exclusive. If both options are specified in command line, CC665S issues fatal error message.

Example 3.56

```
C:\> CC665S /AWIN /T m66589 test.c <CR>
```

In the above example, "awin" pseudo is output in the assembly listing file "test.asm".

Example 3.57

```
C:\> CC665S /AWIN /T m66589 /WIN test.c <CR>
```

In the above example, CC665S outputs fatal error message as /WIN and /AWIN options are mutually exclusive.

3.2.11.7 /SYS OPTION

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

Example 3.58

```
C:\> CC665S /SYS /T m66589 test.c <CR>
```

In the above example, CC665S uses a different segment naming strategy while compiling "test.c".

3.2.12 Invalid Combination Of Options

The following are invalid combinations of command line options.

1. Both preprocessor options (/LP and /PC)
2. Both debugging options (/SD and /OSD)
3. /WIN and /AWIN
4. /LE and preprocessor options (/LP or /PC).
5. /CT and preprocessor options (/LP or /PC).
6. /Fa and preprocessor options (/LP or /PC).
7. /Om and /Ot.
8. /Od and other optimization options (/Ol, /Og, /Oa, /Om and /Ot).
9. Both core options (/nX500 and /nX500S)
10. Invalid C and mixed memory model combinations as given in table 3.1.

4. MEMORY MODELS

This section describes about the various memory models supported by CC665S and the additional memory model qualifiers provided.

4.1 C MEMORY MODELS

CC665S supports the following C memory model options:

1. Small C memory model
2. Effective medium C memory model
3. Medium C memory model
4. Compact C memory model
5. Effective large C memory model
6. Large C memory model

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

1. /MS option for Small C memory model
2. /MEM option for Effective medium C memory model
3. /MM option for Medium C memory model
4. /MC option for Compact C memory model
5. /MEL option for Effective Large C memory model
6. /ML option for Large C memory model

4.2 HARDWARE MEMORY MODELS

MSM665xx chips can be classified into the following four types based on memory availability:

1. Small Memory Model
2. Medium Memory Model
3. Compact Memory Model
4. Large Memory Model

Small Memory Model architecture supports one physical data segment and one physical code segment. In Medium Memory Model, one physical data segment and more than one physical code segment are available. Compact Memory Model chips contain more than one physical data segment and one physical code segment. Large Memory Model architecture supports more than one physical data segment and more than one physical code segment.

Command line options to specify the hardware memory model options are as follows:

1. /mixM to specify Medium hardware memory model
2. /mixC to specify Compact hardware memory model
3. /mixL to specify Large hardware memory model

4.3 OBJECTS AFFECTED BY MEMORY MODELS

The objects of a C program that are affected by memory models may be divided into the following four major divisions:

1. Data Variables
2. Tables
3. Strings
4. Functions

Variables that are allocated in data memory are called **Data Variables**. Variables that are allocated in code memory are called **Tables**.

4.3.1 Memory Model Qualifiers

The following memory model qualifiers are supported by CC665S:

1. **__far**
2. **__nfar**

__far qualifier may be used with data variables, tables and functions. **__nfar** qualifier may be used only with functions.

4.3.2 Data Variables

4.3.2.1 NEAR DATA VARIABLES

If the C memory model option instructs the compiler to use at most one physical data segment, then CC665S allocates all the data variables which are not qualified by **__far** in physical segment #0. These data variables are called near data variables. CC665S will not consider Data Segment Register for accessing near data variables. The size of pointer to a near data variable is 2 bytes.

4.3.2.2 LARGE DATA VARIABLES

If the C memory model option instructs the compiler to use more than one physical data segment, then CC665S allocates data variables in any physical data segment. These data variables are called large data variables. CC665S will switch the Data Segment Register accordingly before each access of large data variables. The size of pointer to a large data variable is 4 bytes.

4.3.2.3 FAR DATA VARIABLES

If the C memory model option instructs the compiler to use at most one physical data segment, but the hardware memory model supports more than one physical data segments, then the **__far** qualified data variables will be allocated in any physical data segment. Such data variables are called far data variables. CC665S will switch the Data Segment Register accordingly before and after each access of far data variables. The size of pointer to a far data variable is 4 bytes.

4.3.3 Tables

4.3.3.1 NEAR TABLES

If the C memory model option instructs the compiler to use at most one physical code segment and one physical data segment, then the non **__far** qualified tables will always be allocated in physical code segment #0. Such tables are called near tables. CC665S will not consider Table Segment Register while accessing near tables. The size of pointer to a near table is 2 bytes.

4.3.3.2 EFFECTIVE NEAR TABLES

If the C memory model option instructs the compiler to use more than one physical code segment and at most one physical data segment, then the non **__far** qualified tables will be allocated in any physical segment. However, only one physical segment will be used for all the non far qualified tables in the program. Total size of the tables cannot exceed 64 Kilobytes. The startup routine initializes the Table Segment Register with the number of the segment allocated for tables. CC665S will not consider Table Segment Register while accessing effective near tables. The size of pointer to a effective near table is 2 bytes.

4.3.3.3 XNEAR TABLES

If the C memory model option instructs the compiler to use at most one physical code segment and more than one physical data segment, then the non **__far** qualified tables will be allocated in physical segment #0. Such tables are called xnear tables. CC665S will not consider Table Segment Register while accessing xnear tables. The size of pointer to a xnear table is 4 bytes. In these C memory models, the compiler assumes the data variables as large data variables. If the table is allocated in ROMWINDOW region data memory instructions are used to access these tables. As the size of pointers to data memory is 4 bytes, the size of pointers to a xnear tables are also 4 bytes.

4.3.3.4 EFFECTIVE XNEAR TABLES

If the C memory model option instructs the compiler to use more than one physical code segment and more than one physical data segment, then the non **__far** qualified tables will be allocated in any physical segment. However, all non **__far** qualified tables will be restricted to one physical code segment. Such tables are called effective xnear tables.

The startup routine initializes the Table Segment Register with the number of the segment allocated for tables. CC665S will not consider Table Segment Register while accessing effective xnear tables. The size of pointer to a effective xnear table is 4 bytes. In these C memory models, the compiler assumes the data variables as large data variables. If the table is allocated in ROMWINDOW region data memory instructions are used to access these tables. As the size of pointers to data memory is 4 bytes, the size of effective xnear tables are also 4 bytes.

4.3.3.5 FAR TABLES

If the hardware memory model supports more than one physical code segment, then **__far** qualified tables will be allocated in any physical code segment. These tables are called far tables. CC665S will switch Table Segment Register accordingly before and after each access to the far table. The size of pointer to a far table is 4 bytes.

4.3.4 Strings

Irrespective of memory models, a string type is same as default type of table. Strings cannot be qualified by **__far**. All strings are restricted to one physical segment only. Size of pointers to strings is 2 bytes if the C memory model options instructs the compiler to use at most one physical data segment. Otherwise, the size of pointers to strings is 4 bytes. Allocations of strings are similar to that of tables.

4.3.5 Functions

4.3.5.1 NEAR FUNCTIONS

If C memory model option instructs the compiler to use at most one physical code segment, then all non **__far** qualified functions will be allocated in physical code segment #0. These functions are called near functions. Calls to these functions is through “cal” instruction. These functions return using “rt” instruction. Size of pointer to a near function is 2 bytes.

4.3.5.2 LARGE FUNCTIONS

If C memory model option instructs the compiler to use more than one physical code segment, then CC665S allocates functions in any physical code segment. These functions are called large functions. Large functions are called using “fcal” instruction. These functions return using “frt” instruction. Size of pointer to a large function is 4 bytes.

4.3.5.3 FAR FUNCTIONS

If the hardware memory model supports more than one physical code segment, then CC665S allocates **__far** qualified functions in any physical segment. **__far** qualified functions are called far functions. Far functions are called using "fcal" instruction. Far functions return through "frt" instruction. Size of pointer to a far function is 4 bytes.

Far functions can be called by any other function. But far functions cannot call near functions.

4.3.5.4 NFAR FUNCTIONS

CC665S allocates all **__nfar** qualified functions in physical code segment #0. **__nfar** qualified functions are called nfar functions. Nfar functions are called using "fcal" instruction. Nfar functions return through "frt" instruction. Size of pointer to a nfar function is 4 bytes.

Nfar functions may be called by any other function. Nfar function may call any other function. Nfar functions act as a bridge between far functions and near functions. Calls to near functions from far functions must be through nfar functions.

4.4 COMBINATION OF C AND MIXED MEMORY MODEL OPTIONS

4.4.1 Small C Memory Model

CC665S generates output code for Small C memory model option when /MS option is specified in command line or when no C memory model option is specified in command line. Small C memory model option instructs the compiler to use one physical code segment and one physical data segment. Under this memory model, default data variables are near data variables, default tables are near tables and default functions are near functions.

4.4.1.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

Under Small C memory model option, if no mixed memory model option is specified in command line, then CC665S assumes the hardware memory model option to be Small memory model. As the hardware supports only one physical code segment and one physical data segment, far and nfar objects are not allowed in this memory model. CC665S issues warning message, if a data, a table or a function is qualified by **__far** or if a function is qualified by **__nfar.**

4.4.1.2 WITH /mixM OPTION

Mixed memory model option /mixM specifies that the hardware supports Medium memory model. Medium hardware memory model contains more than one physical code segment and one physical data segment. Under this option, CC665S allows far tables, far functions and nfar functions. CC665S issues warning message, if a data variable is qualified by **__far**.

4.4.1.3 WITH /mixC OPTION

Mixed memory model option /mixC specifies that the hardware supports Compact memory model. Compact hardware memory model contains one physical code segment and more than one physical data segment. Under this option, CC665S allows far data variables. CC665S issues warning message, if table or function is qualified by **__far** or if function is qualified by **__nfar**.

4.4.1.4 WITH /mixL OPTION

Mixed memory model option /mixL specifies that the hardware supports Large memory model. Large hardware memory model contains more than one physical code segment and more than one physical data segment. Under this option, CC665S allows far data variables, far tables, far functions and nfar functions.

4.4.2 Effective Medium C Memory Model

Command line option /MEM instructs CC665S to generate code for Effective medium C memory model. In effective medium C memory model, CC665S uses at most one physical data segment for data variables, one separate physical code segment for functions and a separate physical code segment for tables and strings. Under this option, default data variables are near data variables, default tables are effective near tables and default functions are near functions.

4.4.2.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

If no mixed memory model option is specified along with Effective medium C memory model option, CC665S assumes the hardware memory model to be Medium memory model. Under this combination, CC665S issues warning message, if data, table or function is qualified by **__far** or if function is qualified by **__nfar**.

4.4.2.2 WITH /mixM OPTION

Under this memory model, CC665S allows far functions, nfar functions and far tables. CC665S issues warning message, if data is qualified by **__far**

4.4.2.3 WITH /mixC OPTION

If /MEM and /mixC options are specified in command line, CC665S issues the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.2.4 WITH /mixL OPTION

Under this memory model, CC665S allows far data variables, far functions, nfar functions and far tables.

4.4.3 Medium C Memory Model

Command line option /MM instructs the compiler to generate code for Medium C memory model option. Under this option, CC665S uses one physical data segment for variables, more than one physical code segment for functions and one physical code segment for tables and strings. In this option, default data variables are near data variables, default functions are large functions, default tables are effective near tables.

As all functions are large functions, if a function is qualified by **__far**, irrespective of mixed memory model option, CC665S ignores the **__far** qualifier without giving any warning message.

4.4.3.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

If no mixed memory model option is specified along with Medium C memory model option, CC665S assumes the hardware memory model to be Medium memory model. Under this combination, CC665S issues warning message, if a data or a table is qualified by **__far**.

4.4.3.2 WITH /mixM OPTION

Under this memory model, CC665S allows far tables. CC665S issues warning message, if a data variable is qualified by **__far**.

4.4.3.3 WITH /mixC OPTION

If /MM and /mixC options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.3.4 WITH /mixL OPTION

Under this memory model, CC665S allows far data variables and far tables.

4.4.4 Compact C Memory Model

Command line option /MC instructs the compiler to generate code for Compact C memory model option. Under this option, CC665S uses more than one physical data segment for variables and one physical code segment for functions, tables and strings. In this option, default data variables are large data variables, default functions are near functions, default tables are xnear tables.

As all data variables are large data variables, if a data variable is qualified by **__far**, irrespective of mixed memory model option CC665S ignores the **__far** qualifier without giving any warning message.

4.4.4.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

If no mixed memory model option is specified along with Compact C memory model option, CC665S assumes the hardware memory model to be Compact memory model. Under this combination, CC665S outputs warning message if a table or a function is qualified by **__far** or if a function is qualified by **__nfar**.

4.4.4.2 WITH /mixM OPTION

If /MC and /mixM options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.4.3 WITH /mixC OPTION

If /MC and /mixC options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.4.4 WITH /mixL OPTION

Under this memory model, CC665S allows far functions, nfar functions and far tables.

4.4.5 Effective Large C Memory Model

Command line option /MEL instructs CC665S to generate code for Effective large C memory model. In effective large C memory model, CC665S uses more than one physical data segment for variables, one separate physical code segment for functions and a separate physical code segment for tables and strings. Under this option, default data variables are large data variables, default tables are effective xnear tables and default functions are near functions.

As all data variables are large data variables, if a data variable is qualified by **__far**, irrespective of mixed memory model option CC665S ignores the **__far** qualifier without giving any warning message.

4.4.5.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

If no mixed memory model option is specified along with Effective large C memory model option, CC665S assumes the hardware memory model to be Large memory model. Under this combination, CC665S outputs warning message if a table or a function is qualified by **__far** or if a function is qualified by **__nfar.**

4.4.5.2 WITH /mixM OPTION

If /MEL and /mixM options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.5.3 WITH /mixC OPTION

If /MEL and /mixC options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.5.4 WITH /mixL OPTION

Under this memory model, CC665S allows far functions, nfar functions and far tables.

4.4.6 Large C Memory Model

Command line option /ML instructs CC665S to generate code for Large C memory model. In Large C memory model, CC665S uses more than one physical data segment for variables, more than one physical code segment for functions and one physical code segment for tables and strings. Under this option, default data variables are large data variables, default tables are effective xnear tables and default functions are large functions.

As all data variables are large data variables, if a data variable is qualified by **__far**, irrespective of mixed memory model option CC665S ignores the **__far** qualifier without giving any warning message. Similarly, as all functions are large functions, if a function is qualified by **__far** or **__nfar**, irrespective of mixed memory model option, CC665S ignores the qualifier without giving any warning message.

4.4.6.1 WITHOUT ANY MIXED MEMORY MODEL OPTIONS

If no mixed memory model option is specified along with Large C memory model option, CC665S assumes the hardware memory model to be large memory model. Under this combination, CC665S outputs warning message if a table is qualified by **__far**.

4.4.6.2 WITH /mixM OPTION

If /ML and /mixM options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.6.3 WITH /mixC OPTION

If /ML and /mixC options are specified in command line, CC665S outputs the fatal error message indicating illegal combination of C and mixed memory model options.

4.4.6.4 WITH /mixL OPTION

Under this memory model, CC665S allows far tables.

5. PRAGMAS

Syntax :

```
#pragma pragma_keyword arguments
```

The directive **#pragma** directs CC665S 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 CC665S 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
```

b. /PF option not specified:

```
#pragma INTERRUPT function_name address
```

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. Extern functions may be specified in interrupt pragma.

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 0xfffe, inclusive of both. The interrupt vector address range 0x0 to 0x06, inclusive of both, is reserved.

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.

CC665S pushes all registers used in interrupt handling function at the entry to this function and it pops the corresponding registers at the exit. "**rti**" instruction is used to return from the interrupt handling routine.

CC665S 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 and usinginpage.
- If a __far or __nfar qualified function is specified in this pragma.
- 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 0x8 to 0xfffe, inclusive of both.
- If an odd address is specified.
- If the function is called in source file.

Example 5.1

INPUT

```
# pragma interrupt fn 0x10
void fn (void)
{
        output_fn () ;
}
```

OUTPUT

```
        $$INTERRUPTCODE segment code #0h

        rseg $$INTERRUPTCODE

CFUNCTION 0
_fn     :

;;{
CLINE 4
        pushs   pr
        pushs   er

;;      output_fn () ;
CLINE 5
        cal     _output_fn

;;}
CLINE 6
        pops    er
        pops    pr
        rti

        extrn code : _output_fn
        public _fn
        extrn code : _main

        cseg #0h at 010h
        dw      _fn
```

The following are erroneous cases :

Example 5.2

INPUT

```
int a ;

# pragma interrupt a 0x10
```

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

Example 5.3

INPUT

```
# pragma interrupt function 9
```

In the above example, an odd address is specified.

Example 5.4

INPUT

```
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, CC665S pushes the registers that may be used during the interrupt handling process in the entry code. The registers pushed, are dependent on the use of floating point emulation routines and function calls in the interrupt routine. The pushed registers are popped in the exit code.

5.1.1.1 INTERRUPT FUNCTION HAS NO FUNCTION CALL AND EMULATION ROUTINE CALL

When an interrupt function has no function calls and emulation routine calls, local registers and pointing registers used in the interrupt function are pushed and popped.

Example 5.5

INPUT

```
int a ;

# pragma interrupt fn 0x10

void fn ()
{
        a = a * a ;
}
```

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

OUTPUT

```
        $$INTERRUPTCODE segment code #0h

        rseg $$INTERRUPTCODE

CFUNCTION 0
_fn     :
```

```
;;{
CLINE 6
        pushs   er0
;;      a = a * a ;
CLINE 7
        l       a,      dir _a
        sqr     a
        mov     dir _a, er0
;;}
CLINE 8
        pops    er0
        rti

        public _fn
        _a comm data 02h #00h
        extrn code : _main

        cseg #0h at 010h
        dw      _fn
```

In the above assembly code for the interrupt function “fn” which has no function calls and emulation routine calls, er0 register is pushed and popped since it is used in the interrupt function.

5.1.1.2 INTERRUPT FUNCTION HAS FUNCTION CALL OR EMULATION ROUTINE CALL

When an interrupt function calls a function or an emulation routine, all local registers and pointing registers are pushed and popped.

Example 5.6

INPUT

```
int a ;

# pragma interrupt fn 0x10

void fn ()
{
        a = a * output_fn () ;
}
```

OUTPUT

```
                $$INTERRUPTCODE segment code #0h

                rseg $$INTERRUPTCODE

        CFUNCTION 0
        _fn     :

        ;;{
        CLINE 6
                pushs   pr
                pushs   er

        ;;      a = a * output_fn () ;
        CLINE 7
                mov     x2,     dir _a
                cal     _output_fn
                l       a,      x2
                mul     dp
                mov     dir _a, er0

        ;;}
        CLINE 8
                pops    er
                pops    pr
                rti

                extrn code : _output_fn
                public _fn
                _a comm data 02h #00h
                extrn code : _main

                cseg #0h at 010h
                dw      _fn
```

In the above assembly code, all local registers and pointing registers are pushed and popped since the interrupt function has a function call.

5.2 INTVECT PRAGMA

Syntax:

a. /PF option specified:

#pragma INTVECT function_name, address

a. /PF option not specified:

#pragma INTVECT function_name address

The pragma **intvect** is same as interrupt pragma except for the following :

1. Intvect pragma may also be specified after the function definition.
2. The function specified in intvect pragma should be qualified by the keyword **__interrupt**.

Example 5.7

INPUT

```
# pragma intvect func 0x40

void __interrupt func (void)
{
        output_fn () ;
}
```

In the above example, function "func" is specified in pragma **intvect** and is also qualified by the keyword '**__interrupt**'. The function "func" is treated as interrupt service routine.

The following examples show erroneous cases :

Example 5.8

INPUT

```
void func (void) ;
# pragma intvect func 0x10
```

In the above example, function "func" is not qualified by the keyword **'__interrupt'**.

5.3 VCAL PRAGMA

Syntax:

a. /PF option specified:

```
#pragma VCAL function_name, address
```

b. /PF option not specified:

```
#pragma VCAL function_name address
```

The pragma **vcal** is used to specify functions, coded in 'C', which can be invoked by VCAL instructions. The addresses of such functions are placed in the VCAL table area in code memory.

If a function is defined in 'C' source program with **function_name** same as that specified along with this pragma, then it is treated as a **VCAL** routine.

The **function_name** in this pragma specifies the name of the vcal function. The **function_name** must be followed by a vcal table address. The table **address** must be an even number between 0x4a and 0x68, inclusive of both. The appropriate **VCAL** table address in code memory is initialized with the address of the function. Extern functions may be specified in vcal pragma.

This pragma must appear before the definition of function specified in the pragma. CC665S issues a warning message if this pragma appears after the definition of the function and ignores it.

If this pragma is used more than once with the same VCAL 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 VCAL addresses.

CC665S 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 is already specified in a pragma other than **vcal** and **usinginpage**.
- If a **__far** or **__nfar** function is specified in this pragma.
- If the pragma is specified after the function definition.
- If the specified address is not in range 0x4a to 0x68, inclusive of both.
- If an odd address is specified.
- If an '**__interrupt**' qualified function is specified in this pragma.

Example 5.9

INPUT

```
# pragma vcal fn 0x64

void fn (void)
{
        output_fn () ;
}

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

OUTPUT

```
            $$NCODUE509 segment code #0h
            $$VCALSEG segment code #0h

            rseg $$VCALSEG

    CFUNCTION 0
    _fn     :

    ;;      output_fn () ;
    CLINE 5
            j       _output_fn

    ;;}
    CLINE 6

            rseg $$NCODUE509

    CFUNCTION 2
    _fn1    :

    ;;      fn () ;
    CLINE 10
            vcal    064h

    ;;}
    CLINE 11
            rt

            extrn code : _output_fn
            public _fn1
            public _fn
            extrn code : _main

            cseg #0h at 064h
            dw      _fn
```

Following examples illustrates erroneous cases:

Example 5.10

INPUT

```
# pragma interrupt function 0x10
# pragma vcal function 0x50
```

In the above example, function is specified in vcal pragma as well as in interrupt pragma.

Example 5.11

INPUT

```
# pragma vcal fn 0x24
```

In the above example, the specified address is not in vcal range.

5.4 ACAL PRAGMA

Syntax:

a. /PF option specified:

```
#pragma ACAL function_name [, function_name ...]
```

b. /PF option not specified:

```
#pragma ACAL function_name [ function_name ...]
```

The pragma **acal** is used to specify functions, coded in ‘C’, which can be invoked using ACAL instructions. The entry points to such functions would be placed in the ACAL area in code memory.

If a function is defined in ‘C’ source program with **function_name** same as that specified along with this pragma, then it is treated as a **ACAL** routine.

A near, **static** far or **static** large function can be specified in acal pragma. If any other function is specified in acal pragma, CC665S outputs warning message. If near and **static** far functions are specified in same acal pragma, CC665S outputs warning message. Extern functions may be specified in this pragma.

This pragma must appear before the definition of function specified in the pragma. CC665S issues a warning message if this pragma appears after the definition of the function and ignores it.

A list of function names may be specified in this pragma. CC665S issues warning message if symbols other than functions are specified in this pragma.

CC665S 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 is already specified in a pragma other than **usinginpage**.
- If the specified function is not a near, **static** far or **static** large function.
- If both near and **static** far functions are specified in same **acal** pragma.
- If the pragma is specified after the function definition.
- If an ‘**__interrupt**’ qualified function is specified in this pragma.

Example 5.12

INPUT

```
# pragma acal fn

void fn()
{
        output_fn () ;
}

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

In the above program, near function “fn” is called using ACAL instruction as shown in the following output :

OUTPUT

```
        $$NCODUE512 segment code #0h
        $$NACODUE512 segment code inacal #00h

        rseg $$NACODUE512

CFUNCTION 0
_fn     :

;;      output_fn () ;
CLINE 5
        j       _output_fn

;;}
CLINE 6

        rseg $$NCODUE512

CFUNCTION 2
_fn1    :

;;      fn () ;
CLINE 10
        acal    _fn

;;}
CLINE 11
        rt

        extrn code : _output_fn
        public _fn1
        public _fn
```

The following example shows an erroneous case:

Example 5.13

INPUT

```
# pragma ACAL fn1
int __nfar fn1 (void) ;
```

For the above example, CC665S issues a warning message because, nfar function cannot be specified in **acal** pragma.

5.5 CAL PRAGMA

Syntax:

a. /PF option specified:

```
#pragma CAL function_name [, function_name ...]
```

b. /PF option not specified:

```
#pragma CAL function_name [ function_name ...]
```

The pragma **cal** is used to specify functions, which can be invoked using CAL instructions. A static far or static large function can be specified in cal pragma. If any other function is specified in this pragma, CC665S outputs warning message.

CC665S issues a warning message if this pragma appears after the definition of the function and ignores it.

A list of function names may be specified in this pragma. CC665S issues warning message if symbols other than functions are specified in this pragma.

CC665S 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 is already specified in a pragma other than usinginpage.

- If the specified function is not a **static** far or **static** large.
- If an '**__interrupt**' qualified function is specified in this pragma.

Example 5.14

INPUT

```
# pragma cal fun1
static int __far fun1 (void) ;

int __far fun1 (void)
{
        return (1) ;
}

void main (void)
{
        fun1 () ;
}
```

In the above program, **static** far function "fun1" is called using CAL instruction as shown in the following output :

OUTPUT

```
        $$NCODUE514 segment code #0h
        $$FCODUE514 segment code
        STACKSEG 0400h

        rseg $$FCODUE514

CFUNCTION 0
_fun1   :

;;      return (1) ;
CLINE 6
        mov     dp,     #01h

;;}
CLINE 7
        rt

        rseg $$NCODUE514

CFUNCTION 2
_main   :

;;      fun1 () ;
CLINE 11
        cal     _fun1
```

```
;;}
CLINE 12
_$$end_of_main :
        sj      $

        public _fun1
        public _main
        extrn code : $$start_up

        cseg #0h at 0h
        dw      $$start_up
```

The following example shows an erroneous case:

Example 5.15

INPUT

```
# pragma cal fun1 fun2
void fun1 (void) ;
void __nfar fun2 (void) ;
```

In the above example, both "fun1" and "fun2" cannot be specified in cal pragma since "fun2" is a nfar function and "fun1" is a near function.

5.6 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, CC665S issues a warning message.

A list of function names may be specified in this pragma. CC665S issues warning message if symbols other than functions are specified in this pragma. The functions specified in this pragma are treated as **static** function. 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 is recursive.
- If the function has variable number of arguments.
- If the function is defined before the inline pragma specification.
- If the function contains loop, branch, label or **“goto**” statements.
- If there is any asm block in the function.
- If the function is too big.

If all the inline function calls are expanded then code for the function body will not be generated. CC665S outputs warning message if an inline function call is not expanded.

CC665S 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.
- If an ‘**__interrupt**’, ‘**__far**’ or **__nfar**’ qualified function is specified in this pragma.

Example 5.16

INPUT

```
int var ;
# pragma inline fn

int fn (int arg)
{
        return (arg*arg) ;
}

void fn1()
{
        var = fn (var) ;
}
```

OUTPUT

```
                $$NCODUE516 segment code #0h

                rseg $$NCODUE516

        CFUNCTION 1
        _fn1    :

        ;;      var = fn (var) ;
        CLINE 11
                l       a,      dir _var
                sqr     a
                mov     dir _var, er0

        ;;}
        CLINE 12
                rt

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

Example 5.17

INPUT

```
        # pragma inline fn

        void fn ()
        {
                fn () ;
        }

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

The inline function “fn” is not expanded since it is recursive. CC665S outputs warning message in this case.

5.7 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.

Variables declared in ‘C’ will be allocated in re-locatable segments. So pointers in ‘C’ are normally used to access specific addresses. But this requires a two byte pointer or four byte pointer depending on the memory model and it is inefficient. In MSM66K “500” core and “500S” core, the addresses of Special Function Registers (SFR) are fixed. To access a SFR it is preferable to use direct specification of addresses. This pragma is used to specify absolute addresses for global variables and **static** local variables.

If physical segment address is not specified, it is considered as zero. CC665S 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, CC665S flags a warning and assigns the address specified with the first pragma. Extern variables may be specified in this pragma.

Physical segment address must be specified in the absolute pragma directive for effective near and effective xnear variables.

An odd address cannot be specified for initialized variables. However, odd address within SFR region can be specified for any type of uninitialized variables. Both, odd and even addresses can be specified for variables of type **char** and array of **char**.

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

CC665S 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 an odd address outside SFR region is specified for uninitialized variables other than **char** and array of **char**.
- If segment address is not specified for 'effective near' and 'effective xnear' variables.
- If specified address is not in absolute range.
- If the pragma is specified after variable initialization.

Example 5.18

INPUT

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

OUTPUT

```
_acc data 040h
```

Example 5.19

INPUT

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

OUTPUT

```
        public _la

        dseg #02h at 01000h
_la :
        ds      04h
```

Following example illustrates an erroneous case:

Example 5.20

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.

5.8 SFR PRAGMA

Syntax:

a. /PF option specified:

#pragma SFR name, [segment:]offset

b. /PF option not specified:

#pragma SFR name [segment:]offset

This pragma is similar to absolute pragma except for the following :

- Only data variables can be specified in this pragma.
- The address specified in this pragma should be in **sfr** region (0x0 to 0xff) or in **xsfr** region (0x100 to 0x1ff). The physical segment address specified in this pragma is always ignored since sfr and xsfr region is in COMMON area.
- CC665S will not output debug information for sfr variables.

CC665S issues warnings for the following cases :

- If the specified symbol is not a global variable or static local variable.
- If the specified variable is qualified by ‘**const**’.
- If the variable is already specified in any pragma.

- If the variable specified in this pragma is not declared within the same file.
- If specified address is not in sfr region or in xsfr region.
- If an odd address is specified for initialized variables.
- If the pragma is specified after variable initialization.

Example 5.21

INPUT

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

OUTPUT

```
        _acc data 040h
```

Following example illustrates an erroneous case:

Example 5.22

INPUT

```
# pragma sfr a 0x400
```

In the above example, the address specified in the pragma is not in sfr area.

5.9 INPAGE PRAGMA

Syntax:

a. /PF option specified:

```
#pragma INPAGE [(no)] segment_name, name [, name ...]
```

b. /PF option not specified:

```
#pragma INPAGE [(no)] segment_name name [ name ...]
```

This pragma instructs the compiler to allocate one or more global variables or static local variables, given by the list of names in an inpage segment.

If the ‘no’ which indicates the page number is specified, then the segment would be allocated in the indicated page. ‘no’ being optional, if omitted, the segment would be allocated in any one of the 256 pages. ‘no’ is an integer constant, takes value between 0 and 255 (inclusive of both). Segment name specified in this pragma, must not be specified in a sbainpage pragma earlier, however, same page number ‘no’ can be specified in both inpage and sbainpage pragmas.

If more than one inpage pragma appears with same segment name , then all variables specified in the list of names in each of these pragmas, are allocated in the same segment.

CC665S issues warning for the following cases:

- If the symbol specified in this pragma is not a global or static local variable.
- If the variable specified in this pragma is qualified by ‘**const**’.
- If the variable is already specified in any pragma.
- If the segment name specified in this pragma is already specified in **sbainpage** pragma.
- If two different page numbers are specified for same segment.
- If both, near and far variables are specified with same segment name
- If the pragma is specified after variable initialization.

Example 5.23

INPUT

```
int a ;
# pragma inpage page1 a
```

OUTPUT

```
            page1 segment data 2h inpage #00h
            public _a

            rseg page1
_a :
            ds          02h
```

In the above example, variable ‘a’ is allocated in an inpage segment. Page number is not specified in the pragma, so, it will be allocated in any one of the 256 pages.

Example 5.24

INPUT

```
        int a ;
        # pragma inpage (5) page1 a
```

OUTPUT

```
                page1 segment data 2h inpage(5) #00h
                public _a

                rseg page1
        _a :
                ds          02h
```

In the above example, page number is specified. Therefore, the variable ‘a’ is output in a segment which will be allocated in page 5.

Following example illustrates an erroneous case:

Example 5.25

INPUT

```
        int a ;
        int __far b ;
        # pragma inpage (5) page1 a b
```

In the above code, both near and far variables are specified with inpage segment ‘page1’.

5.10 SBAINPAGE PRAGMA

Syntax:

a. /PF option specified:

```
        #pragma SBAINPAGE [(no)] segment_name, name [, name ...]
```

b. /PF option not specified:

```
        #pragma SBAINPAGE [(no)] segment_name name [ name ...]
```

The pragma **sbainpage** specifies the variables to be allocated in a segment with SBA attribute. SBA is a Special Bit Addressable Area.

This pragma instructs CC665S to allocate one or more global variables or static local variables given by the list of names in a segment with SBA attribute. All the variables specified with the same segment name are allocated in the same SBA segment. If optionally, a 'no' which indicates the page number is given, then the segment would be allocated in the indicated page. The 'no' can take any value between 0 and 255. Segment name specified in this pragma, must not be specified in a inpage pragma earlier, however, same page number 'no' can be specified in both inpage and sbainpage pragmas.

If more than one sbainpage pragma appears with same segment name , then all variables specified in the list of names in each of these pragmas, are allocated in the same SBA segment.

Variables qualified by **const** cannot be specified in this pragma.

CC665S issues warning for the following cases:

- If the symbol specified in this pragma is not a global or static local variable.
- If the variable specified in this pragma is qualified by '**const**'.
- If the variable is already specified in any pragma.
- If the segment name specified in this pragma is already specified in **inpage** pragma.
- If two different page numbers are specified for same segment.
- If both, near and far variables are specified with same segment name.
- If the pragma is specified after variable initialization.

Example 5.26

INPUT

```
# pragma SBAINPAGE SEG1 bit_var
struct tag
{
        unsigned int bit1 : 1 ;
        unsigned int bit2 : 1 ;
        unsigned int bit3 : 1 ;
} bit_var ;
```

OUTPUT

```
        SEG1 segment data 2h sba #00h
        public _bit_var

        rseg SEG1
_bit_var :
        ds      02h
```

Example 5.27

INPUT

```
# pragma SBAINPAGE (5) seg2 bit_var

struct tag
{
        unsigned int a : 1 ;
        unsigned int b : 1 ;
} bit_var ;
```

OUTPUT

```
        seg2 segment data 2h sba(5) #00h
        public _bit_var

        rseg seg2
_bit_var :
        ds      02h
```

Following example illustrates an erroneous case:

Example 5.28

INPUT

```
int a, b ;
int __far c ;
# pragma sbainpage  sba_page a b
# pragma sbainpage  sba_page c
```

In the above code, both near and far variables are specified with sbainpage segment 'sba_page'. CC665S ignores second pragma with a warning message.

5.11 USINGINPAGE PRAGMA

Syntax:

a. /PF option specified:

```
#pragma USINGINPAGE [-lrb] function_name, segment_name
#pragma USINGINPAGE [-lrb] function_name, pageno
```

b. /PF option not specified:

```
#pragma USINGINPAGE [-lrb] function_name segment_name
#pragma USINGINPAGE [-lrb] function_name pageno
```

This pragma specifies an **inpage** or **sbainpage** segment "segment_name" or page number "pageno" to be used in a function specified by "function_name". In such cases the pageno is used to initialize the Local Register Base (LRB). "pageno" is an integer constant that takes value between 0 and 255, inclusive of both.

CC665S uses current page addressing for the variables allocated in the same page, which is used in this function.

If '-lrb' option is not specified in the pragma, then LRB register is saved in the entry code and restored in the exit code of the usinginpage function. If '-lrb' option is specified, CC665S will not save and restore the LRB register. The '-lrb' option is intended to save unnecessary manipulation of the LRB register when the function and its caller use the same page.

Either 'segment_name' or 'page_number' specified in this pragma must be specified in **inpage** or **sbainpage** pragma prior to this directive. Extern functions and function "main" may be specified in this pragma.

This pragma must appear before the definition of function specified in the pragma. CC665S issues a warning message if this pragma appears after the definition of the function and ignores it.

CC665S issues warning for the following cases:

- If the specified symbol is not a function.
- If the function specified in this pragma is not declared in the file being compiled.
- If the segment name or page number specified in the pragma is not defined in **inpage** or **sbainpage** pragma prior to this directive.
- If the function specified in this pragma is already specified in a pragma other than **interrupt**, **intvect**, **vcal**, **acal** and **cal**.
- If the pragma is specified after the function definition.

Example 5.29

INPUT

```
# pragma inpage seg1 var1 var2
# pragma usinginpage fun1 seg1
# pragma usinginpage -lrb fun2 seg1
int var1, var2 ;
```

```
void fun1()
{
        var1 = 10 ;
        fun2 () ;
}

fun2 ()
{
        var2 = var1 * var1 ;
}
```

The following is the code generated for functions “fun1” and “fun2”.

OUTPUT

```
        $$NCODue529 segment code #0h
        seg1 segment data 2h inpage #00h

        rseg $$NCODue529

CFUNCTION 0
_fun1   :

;;{
CLINE 7
        pushs   lrb
        movb    ALRBH, #page seg1
        using page      seg1

;;      var1 = 10 ;
CLINE 8
        mov     off _var1,      #0ah

;;      fun2 () ;
CLINE 9
        cal     _fun2

;;}
CLINE 10
        pops    lrb
        using page      any
        rt

CFUNCTION 2
_fun2   :

;;{
CLINE 13
        using page      seg1
;;          var2 = var1 * var1 ;
CLINE 14
            l           a,          off _var1
            sqr         a
            mov         off _var2,              er0

;;}
CLINE 15
            using page              any
            rt

            public _fun2
            public _fun1
            public _var2
            public _var1
            extrn code : _main

            rseg seg1
_var2 :
            ds          02h
_var1 :
            ds          02h
```

5.12 GROUP PRAGMA

Syntax:

a. /PF option specified:

#pragma GROUP segment_name [, segment_name ..]

b. /PF option not specified:

#pragma GROUP segment_name [segment_name ..]

The pragma **group** instructs the compiler to allocate the specified segments in the same physical segment by using the pseudo instruction group. Segment names specified in the group pseudo instruction must have been specified earlier in pragma inpage or sbainpage.

All segments specified in a group pragma should be either near segments or far segments. If mix of near and far segments are specified in a group pragma, then near segments will be ignored with warning message.

The following defines the three types of segments based on the variables specified with that segment in inpage/sbainpage pragma :

- A segment specified in inpage/sbainpage pragma is said to be a **near segment** if, only near variables are specified with that segment name.

- A segment is said to be a **far segment** if, only far variables are specified with that segment name in inpage/sbainpage pragma.
- A segment is said to be **undefined segment** if, all the variables specified with that segment name is not declared in the source file.

CC665S issues warning for the following cases :

- If both near and far segments are specified in the same group pragma.
- If the segment is not defined in inpage or sbainpage pragma prior to the group pragma.
- If the segment is "undefined".

Example 5.30

INPUT

```
# pragma INPAGE seg1 var1 var2
# pragma SBAINPAGE (2) seg2 var3 var4
# pragma GROUP seg1 seg2
int var1, var2, var3, var4 ;

fn ()
{
        var1 = var2 + var3 + var4 ;
        var2 = var1 + var3 + var4 ;
        var3 = var1 + var2 + var4 ;
        var4 = var1 + var2 + var3 ;
}
```

This example shows how group pragma can be used with inpage or sbainpage pragmas. If the above program is compiled in large data memory model, DSR switching will be done only once to compute the values of var1, var2, var3 and var4, because, the segments "seg1" and "seg2" are in same physical segment, so these four variables are in the same physical segment.

The following code is output for the function “fn”:

OUTPUT

```
CFUNCTION 0
_fn     :

;;      var1 = var2 + var3 + var4 ;
CLINE 8
        movb    DSR,    #SEG _var2
        l       a,      OFFSET _var2
        add     a,      OFFSET _var3
        add     a,      OFFSET _var4
        st      a,      OFFSET _var1

;;      var2 = var1 + var3 + var4 ;
CLINE 9
        add     a,      OFFSET _var3
        add     a,      OFFSET _var4
        st      a,      OFFSET _var2

;;      var3 = var1 + var2 + var4 ;
CLINE 10
        l       a,      OFFSET _var1
        add     a,      OFFSET _var2
        st      a,      er0
        add     a,      OFFSET _var4
        st      a,      OFFSET _var3

;;      var4 = var1 + var2 + var3 ;
CLINE 11
        add     a,      er0
        st      a,      OFFSET _var4

;;}
CLINE 12
        frt

        group   seg1 seg2
```

The following examples show erroneous cases:

Example 5.31

INPUT

```
# pragma INPAGE inpage_seg var1 var2
# pragma SBAINPAGE sba_seg var3 var4
# pragma GROUP inpage_seg sba_seg
int __far var3, __far var4 ;
int var1, var2 ;
```

For the above example, CC665S outputs warning message and ignores the near segment “inpage_seg” specification in the group pragma with far segment “sba_seg”.

Example 5.32

INPUT

```
# pragma INPAGE SEG5 var1 var2
# pragma GROUP SEG5 SEG6
int var1, var2 ;
```

For the above example, CC665S issues a warning message as segment SEG6 was not specified either in INPAGE or SBAINPAGE pragmas.

5.13 WINDOW PRAGMA

window pragma is not supported by CC665S from version 1.70. If window pragma is specified, it is ignored with a warning message.

5.14 ROMWINDOW PRAGMA

Syntax:

a. /PF option specified:

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

b. /PF option not specified:

```
#pragma ROMWINDOW variable [ variable ..]
```

The pragma **romwindow** instructs the compiler to allocate one or more global variables given by the list of variables within the ROMWINDOW area, but excluding the EEPROM, DUAL PORT and internal RAM ranges.

Local variables cannot be allocated in ROMWINDOW, because they are allocated in stack. CC665S issues a warning message if a variable specified in this pragma is not qualified by ‘**const**’, because ROMWINDOW area is in ROM.

This pragma is ignored when /WIN or /AWIN option is specified in the command line.

Romwindow pragma can be specified to 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.

CC665S accesses variables allocated in ROMWINDOW area using RAM addressing modes and not through ROM addressing modes.

CC665S 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 variable is not qualified by '**const**'.
- If the variable is already specified in any pragma.
- If the variable is initialized before specifying in this pragma directive.

Example 5.33

INPUT

```
const int romvar ;
# pragma romwindow romvar
```

OUTPUT

```
        $$NWINUE533 segment code window #0h
        public _romvar

        rseg $$NWINUE533
_romvar :
        dw      00h
```

The following case is erroneous because, romwindow variables must be qualified by '**const**'.

Example 5.34

INPUT

```
int var ;
# pragma romwindow var
```

5.15 FIXED PAGE PRAGMA

Syntax:

a. /PF option specified:

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

b. /PF option not specified:

```
#pragma FIX variable [ variable ..]
```

The pragma **fix** instructs the compiler to allocate one or more global variables given by the list of variables within the FIXED PAGE area in RAM.

Local variables cannot be allocated in FIXED PAGE area, because they are allocated in stack. CC665S issues a warning message if a variable specified in this pragma is qualified by ‘**const**’, because FIXED PAGE area is in RAM.

CC665S accesses the variables allocated in FIXED PAGE area using fixed page addressing modes.

CC665S 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 variable is qualified by ‘**const**’.
- If the variable is already specified in any pragma.
- If a far variable is specified.
- If the variable is initialized before this pragma directive.

Example 5.35

INPUT

```
int fix_var ;
# pragma fix fix_var
```

OUTPUT

```
        _fix_var comm data 02h fix #00h
```

5.16 DUAL PORT PRAGMA

Syntax:

a. /PF option specified:

#pragma DUAL variable [, variable ..]

b. /PF option not specified:

#pragma DUAL variable [variable ..]

The pragma **dual** instructs the compiler to allocate one or more global variables or **static** local variables given by the list of variables within the DUAL PORT area in RAM.

Local variables cannot be allocated in DUAL PORT area, because they are allocated in stack.

Since DUAL PORT area is RAM, CC665S issues a warning message if a variable specified in this pragma is qualified by '**const**'.

CC665S 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 variable is qualified by '**const**'.
- If the variable is already specified in any pragma.
- If a far variable is specified.
- If the variable is initialized before this pragma directive.

Example 5.36

INPUT

```
int dual_var ;
# pragma dual dual_var
```

OUTPUT

```
        _dual_var comm data 02h dual #00h
```

5.17 EDATA PRAGMA

Syntax:

a. /PF option specified:

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

b. /PF option not specified:

```
#pragma EDATA variable [ variable ..]
```

The pragma **edata** instructs the compiler to allocate one or more global variables or **static** local variables given by the list of variables within the EEPROM area in RAM.

Local variables cannot be allocated in EEPROM area, because they are allocated in stack. CC665S issues a warning message if a variable specified in this pragma is qualified by '**const**', because EEPROM area is in RAM.

CC665S 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 variable is qualified by '**const**'.
- If the variable is already specified in any pragma.
- If a far variable is specified.
- If the variable is initialized before this pragma directive.

Example 5.37

INPUT

```
int edata_var ;
# pragma edata edata_var
```

OUTPUT

```
        $$NEDATAUE537 segment edata 02h
        public _edata_var

        rseg $$NEDATAUE537
_edata_var :
        dw        00h
```

5.18 SBAFIX PRAGMA

Syntax:

a. /PF option specified:

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

b. /PF option not specified:

```
#pragma SBAFIX variable [ variable ..]
```

The pragma **sbafix** instructs the compiler to allocate one or more global variables or **static** local variables given by the list of variables within the SBA AREA in the fixed page.

Local variables cannot be allocated in SBA area in fixed page because, they are allocated in stack. CC665S issues a warning message if a variable specified in this pragma is qualified by '**const**', because SBA area is in RAM.

CC665S 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 variable is qualified by '**const**'.
- If the variable is already specified in any pragma.
- If a far variable is specified.
- If the variable is initialized before this pragma directive.

Example 5.38

INPUT

```
int sbafix_var ;
# pragma sbafix sbafix_var
```

OUTPUT

```
_sbafix_var comm data 02h sba fix #00h
```

5.19 COMMONVAR PRAGMA

Syntax:

a. /PF option specified:

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

b. /PF option not specified:

```
#pragma COMMONVAR variable [ variable ..]
```

The pragma commonvar instructs the compiler to allocate one or more global variables or **static** local variables given by the list of variables within the COMMON AREA in RAM.

This pragma is valid only in large data C memory model programs (compact, effective large and large models). CC665S issues a warning if it is specified in other memory model programs.

Local variables cannot be allocated in COMMON area because, they are allocated in stack. CC665S issues a warning message if a variable specified in this pragma is qualified by ‘**const**’, because COMMON area is in RAM.

CC665S issues warning for the following cases :

- If the data memory model is not large.
- 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 variable is qualified by **‘const’**.
- If the variable is already specified in any pragma.
- If a far variable is specified.
- If the variable is initialized before this pragma directive.

Example 5.39

INPUT

```
int com_var ;
# pragma commonvar com_var
```

CC665S generates the following when the above code is compiled in large data memory model:

OUTPUT

```
_com_var comm data 02h #00h
```

5.20 COMMON PRAGMA

common pragma is not supported by CC665S from version 1.70. If common pragma is specified, it is ignored with a warning message.

5.21 STACKSIZE PRAGMA

Syntax :

```
#pragma STACKSIZE constant
```

The pragma **stacksize** sets stacksize. The constant specifies the size of the stack in bytes. Any even value between 0x1 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. This pragma is valid only if the source file has “main” function definition.

CC665S issues warning for the following case :

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

The following example shows erroneous case:

Example 5.40

INPUT

```
# pragma STACKSIZE 3001
```

For the above pragma, CC665S issues warning message since the stacksize pragma specifies an odd number as stacksize.

5.22 STACK CHECK PRAGMAS

Syntax :

```
#pragma CHECKSTACKON
#pragma CHECKSTACKOFF
```

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

The pragma **checkstackoff** 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 5.41

INPUT

```
# pragma CHECKSTACKON
void fn (void)
{
        fn1 (0) ;
}
```

CC665S generates the following code for function “fn” in /MM option :

OUTPUT

```
CFUNCTION 0
_fn     :

;;{

CLINE 3
        mov     dp,     #06h
        fcal    __chsts50m

;;      fn1 (0) ;
CLINE 4
        clr     a
        pushs   a
        fcal    _fn1
        pops    a

;;}
CLINE 5
        frt
```

5.23 LOOP OPTIMIZATIONS PRAGMAS

Syntax :

```
#pragma LOOPOPTON
#pragma LOOPOPTOFF
```

The pragma **loopopton** instructs the compiler to perform loop optimizations in functions that are defined after this pragma. This pragma is ignored if the command line option /Od is specified.

The pragma **loopoptoff** instructs the compiler not to perform loop optimizations in functions defined after this pragma. This pragma is ignored if the command line option /Od is specified.

5.24 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”. CC665S does not process this block of text. This block will be output in the assembly listing file as given in the source file.

CC665S issues warning for the following case:

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

CC665S 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 5.42

INPUT

```
fn ()
{
# pragma asm
        clrb    TSR             ;;      clear table segment register
        clrb    DSR             ;;      clear data segment register
# pragma endasm
}
```

CC665S generates the following function body for function “fn”:

OUTPUT

```
CFUNCTION 0
_fn     :

;;# pragma asm
CLINE 3
        clrb    TSR             ;;      clear table segment register
        clrb    DSR             ;;      clear data segment register

;;}
CLINE 7
        rt
```

The following are erroneous cases:

Example 5.43

INPUT

```
fn ()
{
# pragma endasm
# pragma asm
        clrb    TSR             ;;      clear table segment register
        clrb    DSR             ;;      clear data segment register
# pragma endasm
}
```

CC665S issues a warning message for the first **endasm** pragma because, it is specified without its corresponding **asm** pragma.

Example 5.44

INPUT

```
fn ()
{
# pragma asm
        clrb    TSR             ;;      clear table segment register
        clrb    DSR             ;;      clear data segment register
}
```

CC665S issues a fatal error message for the **asm** pragma because, its corresponding **endasm** pragma is not specified.

6. OUTPUT FILES

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

TABLE 6.1	
Output File	**Extension**
*Assembly Output	.ASM
Source/Error Listing	.LST
**Calltree Listing	-
Debug Information File	.DBG
Preprocessed Output	.P66

* 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
Debug Information File	/SD or /OSD
Preprocessed Output	/LP or /PC

* indicates that CC665S 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 CC665S is an assembly file which contains MSM66K “500” core or “500S” core 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:\>CC665S /Tm66589 /MS /mixC /SS 10000 test.c
```

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

OUTPUT

```
;; Compile Options : /Tm66589 /MS /mixC /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.2.01 Apr 1996
```

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:\>CC665S /Tm66589 /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 CC665S, which are required by RAS66K.

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:\>CC665S /Tm66589 test.c <CR>
```

For the above command line, the following pseudo instruction is output in “test.asm”:

OUTPUT

```
type (m66589)
```

6.1.2.2 CMODEL PSEUDO INSTRUCTION

The CMODEL pseudo instruction is used to specify the C memory model in the assembly listing file. One of the following is output based on the C memory model:

small	for small C memory model
emedium	for effective medium C memory model
medium	for medium C memory model
compact	for compact C memory model
elarge	for effective large C memory model
large	for large C memory model

Example 6.5

COMMAND LINE

```
C:\>CC665S  /MC  /Tm66589 test.c <CR>
```

For the above command line, the following pseudo instruction is output in “test.asm”:

OUTPUT

```
cmodel compact
```

6.1.2.3 MODEL PSEUDO INSTRUCTION

The MODEL pseudo instruction is used to specify the mixed memory model in the assembly listing file. One of the following is output based on the mixed memory model:

small	for small mixed memory model
medium	for medium mixed memory model
compact	for compact mixed memory model
large	for large mixed memory model

Example 6.6

COMMAND LINE

```
C:\>CC665S  /MM /mixL /Tm66589 test.c <CR>
```

For the following command line, the following pseudo instruction is output in “test.asm”:

OUTPUT

```
model large
```

6.1.2.4 WIN/AWIN PSEUDO INSTRUCTION

The WIN pseudo instruction is output by CC665S when /WIN option is specified in the command line.

Example 6.7

COMMAND LINE

```
C:\>CC665S  /Tm66589  /WIN test.c
```

For the above command line, the following pseudo instruction is output in “test.asm”:

```
win
```

The AWIN pseudo instruction is output by CC665S when /AWIN option is specified in the command line.

Example 6.8

COMMAND LINE

```
C:\>CC665S  /Tm66589  /AWIN test.c
```

For the above command line, the following pseudo instruction is output in “test.asm”:

```
awin
```

6.1.2.5 SEGMENT DEFINITION PSEUDO INSTRUCTION

This section contains the definitions of all the relocatable segments, that have been used in the assembly output file. Each segment definition contains the name of the segment and the properties associated with that segment.

Example 6.9

$$NCOD*file* segment code #0h

The above segment definition indicates that, the segment ‘$$NCOD*file*’ is allocated in 0th physical code segment.

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
    - CFILE directive
    - CFUNCTION directive
    - CBLOCK directive
    - C source line
    - CLINE directive
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 6.10

INPUT

```
/* ue610.c*/
void fn ()
{
}
```

OUTPUT

```
rseg $$NCODue610
```

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 6.11

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 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. CFILE directive is output only when /SD or /OSD option is specified in the command line.

Example 6.12

INPUT

```
/* content of ue612.h */

int a, b, c ;

int mul_arg ( int a, int b )
{
        return ( a * b ) ;
}

/* content of ue612.c */

#include "ue612.h"

int func1 ()
{
        a = b + c ;
}
```

OUTPUT

```
CFILE 0

... code for the function ‘mul_arg’ in file ‘ue612.h’

CFILE 1

...code for the function ‘func1’ file ‘ue612.c’
```

In the above example ‘CFILE’ directive is output for file ‘ue612.h’ with file number 0 and ‘CFILE’ is output for ‘ue612.c’ with file number 1.

6.1.3.3.2 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.

Example 6.13

INPUT

```
int func_id ()
{
        fun1 () ;
}
```

OUTPUT

```
CFUNCTION 0
_func_id :
```

In the above example, 0 is assigned as the function number for the function ‘func_id’.

6.1.3.3.3 CBLOCK/CBLOCKEND directives

CBLOCK and CBLOCKEND directives are output only when /SD or /OSD option is specified in the command line. For each ‘{’ in the source file, a CBLOCK directive is output. Along with CBLOCK the function id and the block number is also output. Similarly, for each ‘}’ in the source file, a CBLOCKEND directives is output along with the function id and the corresponding block number (specified in CBLOCK directive).

Example 6.14

INPUT

```
int a, b, c ;

void fn ()
{
        {
                a = b +c ;
        }
}
```

OUTPUT

```
....
CBLOCK 0 2

;;              a = b +c ;
CLINE 7
        l       a,      dir _b
        add     a,      dir _c
        st      a,      dir _a
CBLOCKEND 0 2

....
```

6.1.3.3.4 C source line

For each executable line for which assembly instructions are output, the corresponding C statement is output as comments.

Example 6.15

INPUT

```
a = fn ();
```

For the above C statement, the C source line is output in the assembly file as follows:

OUTPUT

```
;; a = fn () ;
```

6.1.3.3.5 CLINE directive

CLINE directive is output for each executable statement, for which assembly instructions have been generated. The CLINE directive is followed by the line number of the C statement in the source file.

Example 6.16

INPUT

```
int a, b, c ;

void
fn ()
{
        a = b * c ;     /* line number 06 */
}
```

OUTPUT

```
......

;;          a = b * c ;           /* line number 06 */
CLINE 6
            l           a,          dir _b
            mul         dir _c
            mov         dir _a,     er0

.....
```

In the above example, for the C statement ‘a = b * c’ at line number 6, ‘CLINE 6’ is output.

6.1.3.4 ASSEMBLY INSTRUCTIONS

One or more assembly instructions are generated for a C statement. They are grouped together and output after the CLINE directive.

Example 6.17

INPUT

```
int b, c ;

void
fn ()
{
            b = fun1 () ;
            c += b ;
}
```

OUTPUT

```
.....

;;          b = fun1 () ;
CLINE 6
            cal         _fun1
            mov         dir _b,     dp

;;          c += b ;
CLINE 7
            l           a,          dp
            add         dir _c,     a
.....
```

In the above example, the assembly instructions that follow CLINE 6 are generated for ‘b = fun1 () ;’ expression and those following CLINE 7 are for the expression ‘c += b ;’.

6.1.4 Symbol Declarations 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 6.18

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 6.19

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 6.20

INPUT

```
        extern int a ;

        main ()
        {
                a = 1 ;

                fn () ;
        }
```

OUTPUT

```
        .....
        extrn code : _fn
        .....
        extrn data : _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. CC665S follows similar methods to output static, non-static and aggregate (array, structure/union) initialized global data variables. Memory initialization instructions DW and DB are used to allocate and initialize **const** variables in code memory.

Example 6.21

INPUT

```
        long var = 10 ;
        const int cint = 20 ;
```

OUTPUT

```
                rseg $$NINITTAB
                dw      0ah
                dw      00h

                rseg $$NTABue621
        _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 in code 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 6.22

INPUT

```
int     a, b ;
int     c = 10 ;

void fn ( void )
{
        b = fn1 () ;

        a = b * c ;

        return ;
}
```

OUTPUT

```
;; Compile Options : /Tm66589 /MS /mixC /SS 10000
;; Version Number  : Ver.2.01 Apr 1996
;; File Name       : ue622.c

        type (m66589)
        cmodel small
        model compact
        $$NCODue622 segment code #0h
        $$NINITTAB segment code
        $$NINITVAR segment data 02h #0h
        rseg $$NCODue622

CFUNCTION 0
_fn     :

;;      b = fn1 () ;
CLINE 6
        cal     _fn1
        mov     dir _b,  dp

;;      a = b * c ;
CLINE 7
        l       a,       dp
        mul     dir _c
        mov     dir _a,  er0

;;}
CLINE 9
        rt

        extrn code : _fn1
        public _c
        public _fn
        _a comm data 02h #00h
        _b comm data 02h #00h
        extrn code : _main

        rseg $$NINITTAB
        dw      0ah

        rseg $$NINITVAR
_c :
        ds      02h

        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 6.23

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 "ue623.c" is compiled in /LE /Tm66589 options:

OUTPUT

```
                                   Page :          1
                                 Date : 04-23-1996
                                 Time :     14:02:34
CC665S C Compiler Ver.2.01 Apr 1996, Source List
Source File : ue623.c


 Line #  Source Line

   1 int  a ;
   2 int b ;
   3
   4 void fn ()
   5 {
   6      output_fn () ;
   7      if ( a == b [1] )
***** ue623.c(7) : Error : E5003 : Subscript on non array
   8              return a ;
   9 }
***** ue623.c(8) : Error : E5039 : Void function returning value
  10


Error(s)     :  2
Warning(s)   :  0
```

If the source file compiles without an error or fatal error, then a list of stack information used in different functions is issued. The following example shows a complete source listing with stack information.

Example 6.24

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 : 04-23-1996
                                                Time :   14:12:12
CC665S C Compiler Ver.2.01 Apr 1996, Source List
Source File : ue624.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 : 04-23-1996
                                   Time :     14:12:12
CC665S C Compiler Ver.2.01 Apr 1996, Source List
Source File : ue624.c



                STACK INFORMATION
                -----------------

 FUNCTION                LOCALS   ARGUMENTS   OTHERS   TOTAL
 --------                  ------    ---------      ------    -----
 _begin                   0         4              8         12
 _end                     0         4              6         10
```

OTHERS include the size of stack used for storing the return address of the function and the size of stack used for pushing the base registers at the entry of the function.

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 6.25

INPUT

```
void
fn ()
{
}

void
fn1 ()
{
        fn () ;
        fn1 () ;
        fn2 () ;
}
```

For the above source file “ue625.c”, the calltree listing generated by CC665S is shown below.

```
CC665S C Compiler, Ver.2.01 Apr 1996, Calltree Listing

Source File : ue625.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 6.26

INPUT

```
/* ue626a.c */
void fn ()
{
        fn () ;
}

/* ue626b.c */
void fn1 ()
{
        fn () ;
}
```

In the above code, the function “fn” is defined in source file “ue626a.c” and function “fn1” in “ue626b.c” calls function “fn”.

OUTPUT

```
CC665S C Compiler, Ver.2.01 Apr 1996, Calltree Listing

Source File : ue626a.c

fn
|   fn*

Source File : ue626b.c

fn1
|   fn?
```

In the calltree listing of function “fn1”, a question mark follows “fn” since function “fn” was not defined in source file “ue605b.c”.

6.4 DEBUGGING INFORMATION FILE

CC665S creates debug information file with an extension “.dbg” and the base name derived from the source file when /SD or /OSD option is specified in the command line. Compiler stores symbol information, line number and block information of source file in debugging information file for the source level debugger CDB665S.

Debugging information file is created as a binary file in a predefined format. This file is opened and processed by RAS66K when /CC option is specified in the command line. Assembler creates the object file which includes the debugging information. The absolute addresses and values are fixed by the linker RL66K when the /SD option is specified in the command line of RL66K and transferred to the absolute file. C source level Debugger CDB665S reads the absolute file to obtain the debugging information.

CC665S also outputs information to support **calls menu** option in source level debugger CDB665S if /SD option is specified in the command line. A debugging information file created by specifying /OSD option in the command line does not contain information to support **calls menu** option in the debugger CDB665S.

Example 6.27

INPUT

```
int a ;

void
fn ( )
{
        a = 1 ;
}
```

The following output is generated when the above program “ue627.c” is compiled in /SD option:

```
CFUNCTION 0
_fn     :
CBLOCK 0 1
;;{
CLINE 5
        l       a,      _$baseptr
        pushs   a
        mov     _$baseptr,      ssp

;;      a = 1 ;
CLINE 6
        mov     dir _a,  #01h

;;}
CLINE 7
        pops    a
        mov     _$baseptr,      a
        rt
CBLOCKEND 0 1
```

The following output is generated when the above program “ue627.c” is compiled in /OSD option:

```
CFUNCTION 0
_fn         :
CBLOCK 0 1

;;          a = 1 ;
CLINE 6
            mov       dir _a,    #01h

;;}
CLINE 7
            rt
CBLOCKEND 0 1
```

7. OPTIMIZATIONS

CC665S 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.

CC665S 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.

CC665S 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 :

Constant propagation

1. Common sub-expression elimination
2. Code sinking
3. 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. The resultant constant expressions are computed at compile time and the computed result is used in the expression.

Example 7.1

```
int a, b, x, y, m, n ;

const_propagate ()
{
        a = 45 ;

        if ( b < x)
        {
            m = a + 20 ;        /* changed to m = 65 */
        }

        y = a + m ;             /* changed to y = 45 + m */
}
```

Assembly code generated by CC665S for the above function ‘const_propagate’ is shown below

```
CFUNCTION 0
_const_propagate            :

;;          a = 45 ;
CLINE 5
            mov     dir _a,     #02dh

;;          if ( b < x)
CLINE 7
            l       a,          dir _b
            cmp     a,          dir _x
            jges    _$L1

;;                  m = a + 20 ;        /* changed to m = 65 */
CLINE 9
            mov     dir _m,     #041h

;;          }
CLINE 10
_$L1 :

;;          y = a + m ;         /* changed to y = 45 + m */
CLINE 12
            l       a,          dir _m
            add     a,          #02dh
            st      a,          dir _y

;;}
CLINE 13
            rt
```

7.1.2 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 7.2

```
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 CC665S for the above function ‘common_sub_exp’ is shown below:

```
CFUNCTION 0
_common_sub_exp         :

;;          x = a + b ;                         /* a + b is also assigned to a temporary */
CLINE 5
            l           a,          dir _a
            add         a,          dir _b
            st          a,          dir _x
            st          a,          er0

;;          if (a < b)
CLINE 7
            l           a,          dir _a
            cmp         a,          dir _b
            jges        _$L1

;;                      m = a + b + y ;         /* a + b is replaced by the temporary */
CLINE 8
            l           a,          er0
            add         a,          dir _y
            st          a,          dir _m

;;          else
CLINE 9
            rt
```

```
_$L1 :

;;          n = (a + b) >> 4 ;   /* a + b is replaced by the temporary */
CLINE 10
        l       a,      er0
        sra     a,      04h
        st      a,      dir _n

;;}
CLINE 11
        rt
```

7.1.3 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 7.3

```
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 CC665S for the above function ‘sink’ is shown below :

```
CFUNCTION 0
_sink   :

;;      if ( a == b )
CLINE 5
        l       a,      dir _a
        cmp     a,      dir _b
        jne     _$L1

;;              func () ;
CLINE 7
        cal     _func
```

```
;;}
CLINE 16
_$L0 :
        l       a,      dir _e
        add     a,      #019h
        st      a,      dir _m
        mov     dp,     dir _e
        rt

;;      }
CLINE 10
_$L1 :

;;      x = y + z ;
CLINE 12
        l       a,      dir _y
        add     a,      dir _z
        st      a,      dir _x

;;      return (e) ;
CLINE 15
        sj      _$L0
```

7.1.4 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 7.4

```
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 CC665S for the above function 'hoist' is shown below:

```
CFUNCTION 0
_hoist  :
        l       a,      dir _x
        add     a,      dir _y
        st      a,      dir _m

;;      if ( a == b )
CLINE 5
        l       a,      dir _a
        cmp     a,      dir _b
        jne     _$L1

;;              x = z ;
CLINE 8
        mov     dir _x, dir _z

;;      else
CLINE 10
        rt
_$L1 :

;;              fn1 () ;
CLINE 13
        j       _fn1

;;}
CLINE 15
```

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 7.5

```
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 CC665S for the above function ‘loop_invar’ is shown below:

```
CFUNCTION 0
_loop_invar       :
        mov     er0,    dir _n
        clr     a
        divq    dir _o
        st      a,      dir _p
        l       a,      dir _m
        mul     dir _r
        mov     er1,    er0

;;      do
CLINE 5
_$L3 :

;;              x = m * r + i ;     /* subexpression m * r is moved outside the loop */
CLINE 8
        l       a,      er1
        add     a,      dir _i
        st      a,      er0
        st      a,      dir _x
```

```
;;                  y [i] += x ;
CLINE 9
          l         a,        dir _i
          sll       a,        01h
          st        a,        x1
          l         a,        er0
          add       _y[x1],   a

;;                  i ++ ;
CLINE 10
          inc       dir _i

;;        } while ( x < i ) ;
CLINE 11
          l         a,        dir _x
          cmp       a,        dir _i
          jlt       _$L3

;;}
CLINE 12
          rt
```

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 7.6

```
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 CC665S for the above function loop_variant_code_motion is shown below:

```
CFUNCTION 0
_loop_variant_code_motion          :
            add       dir _a,    #037h
            mov       dir _i,    #0bh

;;}
CLINE 7
            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 7.7

```
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 CC665S for the above function 'induction_var_elim' is shown below:

```
CFUNCTION 0
_induction_var_elim        :

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
CLINE 6
            clr         dir _i
            mov         dir _j,     #0ah
_$L3 :

;;                      a [ i ] = j + 3 ;
CLINE 8
            lb          a,          dir _i
            addb        a,          #03h
            mov         x1,         dir _i
            stb         a,          _a[x1]

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
CLINE 6
            inc         dir _i
            cmp         dir _i,     #0ah
            jlts        _$L3

;;}
CLINE 10
            rt
```

7.2.4 Strength Reduction

Expressions, in loops, that use costly operations are modified to use cheaper operations.

Example 7.8

```
int a [10] ;
int i ;

strength_reduction ()
{
    for ( i = 0 ; i < 10 ; i ++ )
    {
        a [i] = 0 ;             /* for accessing ith element of the 'a', multiplication by 2 */
                                /* is necessary, because 'a' is an array of 'int' */
    }
}
```

The above loop is transformed to

```
for (temp = 0, i = 0; temp < 20 ; temp += 2, i ++)
{
        * ( a + temp ) = 0 ;            /* multiplication inside the loop is */
                                        /* removed by varying the loop */
                                        /* control (incremented by 2 instead */
                                        /* of 1 ) and the exit condition ( less */
                                        /* than 20 instead of less than 10), */
                                        /* by using a temporary loop */
                                        /* control variable (temp) */
}
```

As CC665 Ver.1.52 and later give priority to space optimization over speed optimization, strength reduction is not performed. Assembly code generated by CC665S for the above function ‘strength_reduction’ is shown below:

```
CFUNCTION 0
_strength_reduction          :

;;          for ( i = 0 ; i < 10 ; i ++ )
CLINE 6
            clr         dir _i
_$L3 :

;;                      a [i] = 0 ;     /* for accessing ith element of the ‘a’, multiplication by 2*/
CLINE 8
            l           a,          dir _i
            sll         a,          01h
            st          a,          x1
            clr         _a[x1]

;;          for ( i = 0 ; i < 10 ; i ++ )
CLINE 6
            inc         dir _i
            cmp         dir _i,     #0ah
            jlts        _$L3

;;}
CLINE 11
            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 7.9

```
loop_unroll ()
{
    int i ;

    for ( i = 0 ; i < 2 ; i ++)
        function () ;
}
```

The above loop is transformed to

```
function () ;
function () ;
```

Assembly code generated by CC665S for the above function ‘loop_unroll’ is shown below:

```
CFUNCTION 0
_loop_unroll        :
        cal         _function
        j           _function

;;}
CLINE 7
```

7.3 OTHER OPTIMIZATIONS

The other optimizations performed include :

1. Dead code elimination
2. Dead variable elimination
3. Algebraic transformation
4. Optimizing jumps

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 7.10

```
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 CC665S for the above function ‘dead_code’ is shown below:

```
CFUNCTION 0
_dead_code      :

;;      a = 10 ;
CLINE 5
        mov     dir _a,   #0ah

;;      r = p + q ;
CLINE 6
        l       a,      dir _p
        add     a,      dir _q
        st      a,      dir _r

;;}
CLINE 10
        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 7.11

```
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 CC665S for the above function ‘dead_var’ is shown below:

```
CFUNCTION 0
_dead_var       :
```

```
;;          r = p * q ;
CLINE 8
            l           a,          dir _p
            mul         dir _q
            mov         dir _r,     er0

;;          x = r >> 2 ;
CLINE 9
            l           a,          er0
            sra         a,          02h
            st          a,          dir _x

;;}
CLINE 11
            rt
```

7.3.3 Algebraic Transformation

Expressions are modified, using commutative and associative laws, for optimal use of registers.

Example 7.12

```
int a, x, b, c ;

alg_transfer ()
{
      a = x + ( b - c ) ;         /* requires 2 registers */
}
```

The above statement is transformed to

```
a = ( b - c ) + x ;               /* requires 1 register */
```

Assembly code generated by CC665S for the above function ‘alg_transfer’ is shown below

```
CFUNCTION 0
_alg_transfer       :

;;          a = x + ( b - c ) ;   /* requires 2 registers */
CLINE 5
            l           a,          dir _b
            sub         a,          dir _c
            add         a,          dir _x
            st          a,          dir _a

;;}
CLINE 6
            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 7.13

```
LABEL2 :
    goto LABEL1          /* LABEL1 is replaced by LABEL2 */
LABEL1 :
    goto LABEL2
```

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

7.4.1 Removal Of Redundant Transfer Instructions

The generated assembly instructions are scanned for unnecessary transfers to and from registers.

Example 7.14

```
l       a,      #20h
st      a,      _one
l       a,      #20h         ; this instruction is removed
st      a,      _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 7.15

```
        jz   L10
        j    L20
    $L10 :
```

The above instructions are replaced by

```
        jnz  L20
    $L10 :
```

7.5 LOCAL OPTIMIZATIONS

These are optimizations that are performed within a basic block :

1. Constant propagation
2. Common sub-expression elimination
3. Use of algebraic identities

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 7.16

```
int c, d ;

local_constant_prop ()
{
      c = 30 ;
      d = c ;          /* instead of c, 30 is assigned to d */
}
```

Assembly code generated by CC665S for the above function ‘local_constant_prop’ is shown below

```
CFUNCTION 0
_local_constant_prop        :

;;          c = 30 ;
CLINE 5
            l           a,          #01eh
            st          a,          dir _c
```

```
;;          d = c ;     /* instead of c, 30 is assigned to d */
CLINE 6
            st          a,          dir _d

;;}
CLINE 7
            rt
```

7.5.2 Common Sub-Expression Elimination

Code containing repeated sub-expressions are modified, so that the sub-expressions are evaluated only once.

Example 7.17

```
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 CC665S for the above function 'local_cse' is shown below:

```
CFUNCTION 0
_local_cse              :

;;          a = b + c * d ;     /* c *d is evaluated and assigned to a temporary */
CLINE 5
            l           a,          dir _c
            mul         dir _d
            l           a,          er0
            add         a,          dir _b
            st          a,          dir _a

;;          x = c * d / y ;     /* value of c * d stored */
CLINE 6
            clr         a
            divq        dir _y
            st          a,          dir _x

;;}
CLINE 8
            rt
```

7.5.3 Use Of Algebraic Identities

Expressions that conform to algebraic laws are modified, so that unnecessary operations are eliminated.

Example 7.18

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

alg_identities ()
{
    a = b + 0 ;     /* addition is eliminated */
    c = d * 1 ;     /* multiplication is eliminated */
}
```

Assembly code generated by CC665S for the above function ‘alg_identities’ is shown below:

```
CFUNCTION 0
_alg_identities     :

;;          a = b + 0 ;         /* addition is eliminated */
CLINE 6
            mov     dir _a,     dir _b

;;          c = d * 1 ;         /* multiplication is eliminated */
CLINE 7
            mov     dir _c,     dir _d

;;}
CLINE 8
            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 CC665S does not check for aliases. The default optimizations performed by CC665S 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 CC665S to check for aliases while performing optimizations. Though this results in correct code, it reduces the extent to which optimizations are performed.

Example 7.19

```
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 CC665S for the above function ‘alias_check’ in default command line option (all optimizations are performed) is shown below (No /Oa option)

```
CFUNCTION 0
_alias_check        :

;;          a = b + c ;
CLINE 5
            l       a,      dir _b
            add     a,      dir _c
            st      a,      er0
            st      a,      dir _a

;;          if (x < a)
CLINE 7
            l       a,      dir _x
            cmp     a,      er0
            jges    _$L1
```

```
;;                      * ptr = 56 ;
CLINE 9
            mov     dp,     dir _ptr
            mov     [dp],   #038h

;;                      y = b + c ;         /* By default, alias are ignored, so */
CLINE 10
            mov     dir _y, er0

;;          }
CLINE 13
_$L1 :

;;}
CLINE 14
            rt
```

Assembly code generated by CC665S for the above function 'alias_check', when '/Oa' option (perform alias check), is specified in the command line, is shown below:

```
CFUNCTION 0
_alias_check        :

;;          a = b + c ;
CLINE 5
            l       a,      dir _b
            add     a,      dir _c
            st      a,      er0
            st      a,      dir _a

;;          if (x < a)
CLINE 7
            l       a,      dir _x
            cmp     a,      er0
            jges    _$L1

;;                      * ptr = 56 ;
CLINE 9
            mov     dp,     dir _ptr
            mov     [dp],   #038h

;;                      y = b + c ;         /* By default, alias are ignored, so */
CLINE 10
            mov     dir _y, er0

;;          }
CLINE 13
_$L1 :

;;}
CLINE 14
            rt
```

8. IMPROVING COMPILER OUTPUT

8.1 CONTROLLING OPTIMIZATIONS

CC665S provides a number of optimization options that can improve program speed. In addition, CC665S include pragmas to control loop optimizations on a local basis within a source program.

Default Optimization

By default, CC665S performs all optimizations. If no optimization must be performed, the user must specify /Od option.

Relaxing Alias Checking

By default, CC665S 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 which on execution may be slower.

/Oa option may be omitted safely by the user, if multiple aliases to refer to the same location, either directly or indirectly, is 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.

Controlling Loop Optimization On A Local Basis

Loop optimizations may be controlled on local basis by using the pragmas LOOPOPTON and LOOPOPTOFF. Loop optimizations are turned off for any function following #pragma LOOPOPTOFF and is turned on for any function following #pragma LOOPOPTON in a source program.

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 CC665S 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.

Speed Optimization

The command line option /Ot enables the compiler to perform speed optimization. This optimization is same /Om option with the only difference that, in /Ot option, stack allocation and deallocation instructions are optimized.

/Ot option with /Oa option enables the user to obtain an output on which speed and safe optimizations are performed.

8.2 USING REGISTER VARIABLES

By default, the compiler allocates registers to local variables. If register is not available, stack locations are used. The order of allocation of these registers is based on the frequency of use of local variables.

Therefore, there is a possibility of not allocating register to a variable that is not used many times, but used in a portion of code that will be executed many times repeatedly. Inorder to allocate registers for such variables, the storage class register may be specified. Variables specified with register keyword has more priority than other variables. In the process of deallocation when register is not available, CC665S deallocates registers assigned to ordinary variables first and then to the variables declared as register variables.

However, CC665S does not guarantee that a register specified variable will always be allocated in registers. The **register** storage class may be specified to any variable, but register specifications are ignored for variables whose type is not **int** or **short** or for pointer types that are not of the same size as type **int**.

8.3 REMOVING STACK PROBES

Program execution may be speed 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-overflows 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 CC665S to call stack probe routines at the beginning of each function.

Stack checking may be controlled on local basis also by using either #pragma CHECKSTACKON or #pragma CHECKSTACKOFF. Stack checking is turned off for any function following #pragma CHECKSTACKOFF and turned on for any function following #pragma CHECKSTACKON.

8.4 CONTROLLING ALLOCATION OF VARIABLES

Pragmas of CC665S may be used in controlling the allocation of variables. This enables CC665S to use variety of addressing modes in order to improve the assembly output.

#pragma INPAGE instructs CC665S to allocate the variables in the inpage area. And if a function uses the same inpage area, then CC665S uses current page addressing modes to access the variables specified in INPAGE pragma, as far as possible.

Using #pragma ABSOLUTE, a variable may be allocated anywhere in code or data memory. This enables the user to access SFR area also.

Using #pragma SFR, a variable may be allocated anywhere in a data memory. This is similar to ABSOLUTE pragma, except that only data memory variables can be specified.

Other pragmas like EDATA, SBAINPAGE, SBAFIX, FIX, DUAL, ROMWINDOW etc., enables the user to allocate the given variable in any part of the memory.

8.5 MIXED LANGUAGE PROGRAMMING

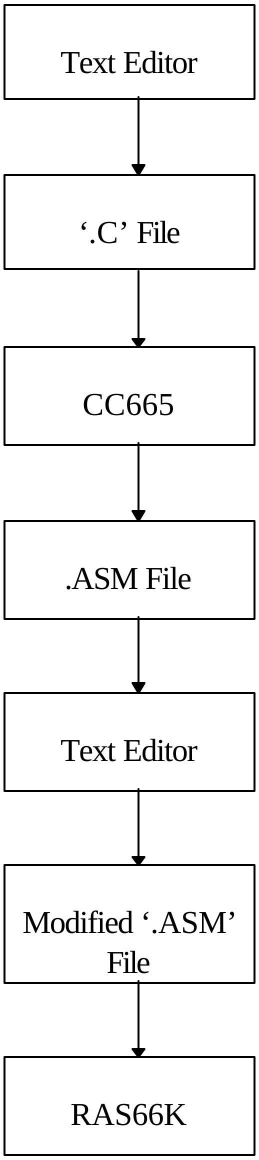
This section explains how to use MSM66K “500” core or “500S” core assembly language routines with C programs and functions compiled using CC665S. In particular it explains how to call assembly language routines from ‘C’ programs and how to call ‘C’ language functions from an assembly language routine.

8.5.1 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:

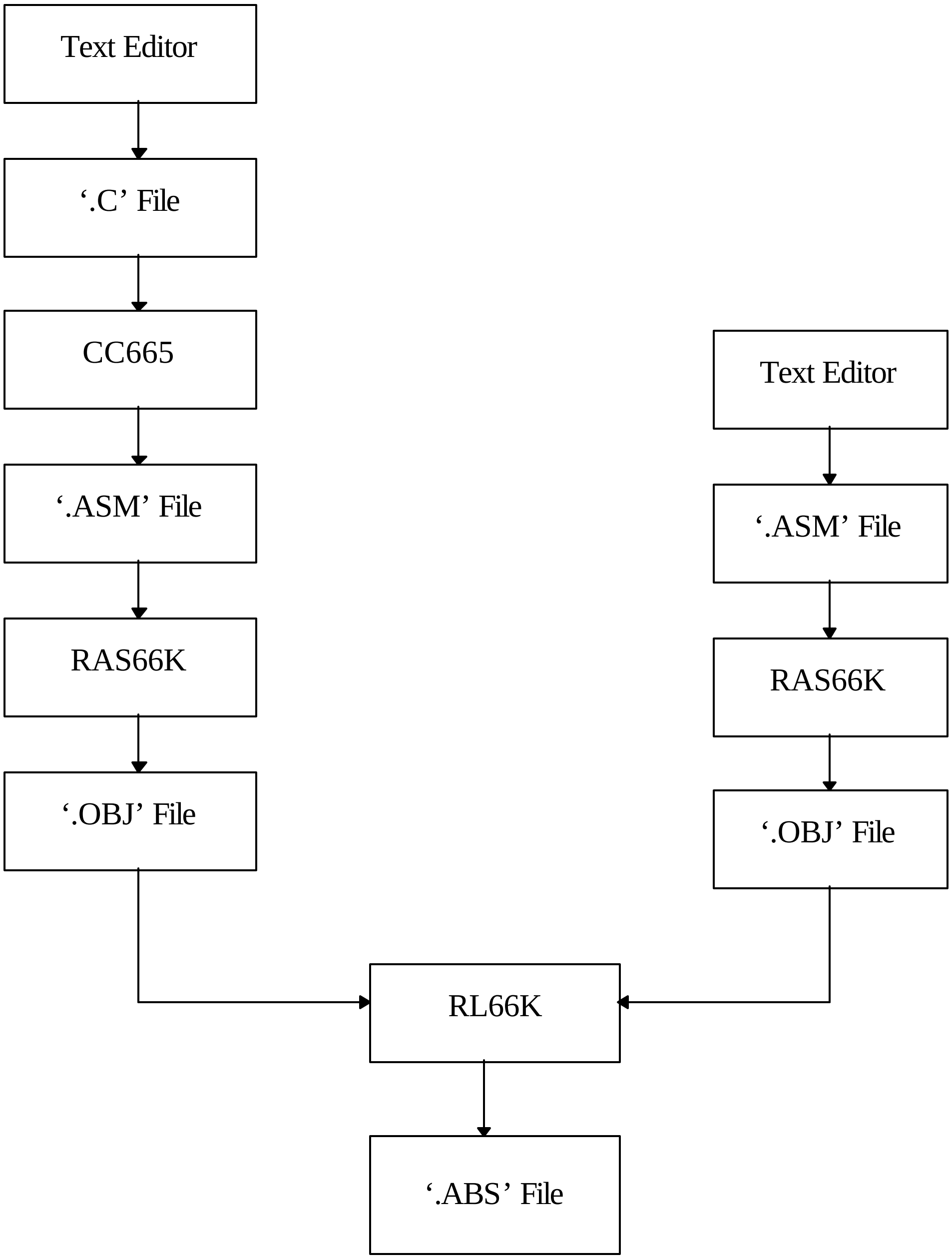
Method 1

In this method, programmer writes a ‘C’ program and then compiles the ‘C’ program using CC665S. The output produced by CC665S is an assembly language file containing MSM66K “500” core or “500S” core 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 RAS66K and RL66K respectively to produce the absolute file.



Method 2

In this method, programmer writes a C program and compiles it using CC665S. 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 RAS66K. The result will be two '.OBJ' files. These two '.OBJ' files can be linked using the linker RL66K.



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**. CC665S 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. CC665S outputs whatever is the argument to this keyword as it is in the output file.

The return value of **__asm** keyword cannot be used.

CC665S 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 8.1

INPUT

```
void fn ()
{
        __asm (“DI\n”, “EI\n” ) ;
}
```

CC665S outputs error message for the above program as more than one argument is specified for **__asm** keyword.

Example 8.2

INPUT

```
void fn ()
{
        return __asm (“DI\n”, “EI\n” ) ;
}
```

CC665S outputs error message for the above program as return value of **__asm**. keyword is used. The following example shows how mixed mode language programming can be used efficiently:

By using mixed mode language programming, programmer can write very efficient and flexible code. For example, if an error recovery library function which takes error number as argument is to be executed without interruption from the maskable hardware interrupts, programmer can disable maskable interrupts before calling that function as shown below:

Example 8.3

INPUT

```
# define  BAD_STATUS    1

int err_no ;

void error_check_fn ()
{
        if ( err_no == BAD_STATUS )
        {
                __asm ( “\t\tDI\n” ) ;          /* maskable hardware interrupts disabled */
                output_error_with_beep ( ) ;
# pragma asm
                mov     a,      dir _err_no
                pushs   a
                cal _error_recovery_fn
                pops    a
# pragma endasm
                __asm ( “\t\tEI\n” ) ;      /* maskable hardware interrupts enabled */
        }
        else
                output_error_with_beep () ;
}
```

The following code is generated for the above function definition:

OUTPUT

```
CFUNCTION 0
_error_check_fn :

;;      if ( err_no == BAD_STATUS )
CLINE 5
        cmp     dir _err_no,    #01h
        jeq     _$L4
        j       _$L1
_$L4 :

;;              __asm ( "\t\tDI\n" ) ;      /* maskable hardware interrupts disabled */
CLINE 7
                DI

;;              output_error_with_beep ( ) ;
CLINE 8
        cal     _output_error_with_beep

;; # pragma asm
CLINE 9
                mov     a,      dir _err_no
                pushs   a
                cal _error_recovery_fn
                pops    a

;;              __asm ( "\t\tEI\n" ) ;      /* maskable hardware interrupts enabled */
CLINE 15
                EI

;;      else
CLINE 17
        rt
_$L1 :
```

```
;;                  output_error_with_beep () ;
CLINE 18
         j          _output_error_with_beep

;;}
CLINE 19
```

8.5.2 Calling Conventions Of CC665S

CC665S 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. CC665S passes arguments to any given function by pushing the value of each of the arguments into a stack form right to left. The function call pushes the value of the last argument first and the first argument last. If an argument is an expression, CC665S computes the expression’s value before pushing it onto the stack. The expression evaluation is carried out from left to right, that is the first argument is evaluated first and the last argument is evaluated last, but the arguments are pushed into the stack in the reverse order.

Arguments, which have **char** or **int** as their type, occupy one word in the stack. Whereas, arguments which are of type **long** or **float** occupy two words in the stack. The **char** type arguments are sign-extended to **int** type before being pushed into stack. If the argument is pointer, the number of words pushed depends on the memory model. For large memory two words are pushed and for small memory one word is pushed into the stack. After a function returns control to a routine 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 SSP.

8.5.3 Return Values

Assembly language routines that wish to return values to a ‘C’ program or receive return values from ‘C’ functions must follow CC665S return value conventions. If the function has a return value of size less than or equal to 2 bytes, CC665S places return value of functions in dp register. If the function has a return value of size greater than 2 bytes, CC665S places the return value in the register pair dp and x1. The higher word of the return value is placed in register x1. If the return value type is structure or union or double, CC665S passes the address of the variable to which the return value is assigned, as the first argument. Therefore, the return value is updated in the called function.

Example 8.4

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

```
CFUNCTION 0
_add_int :

;;{
CLINE 2
        pushs   usp
        mov     usp,    ssp

;;      return ( a + b ) ;
CLINE 3
        l       a,      6[usp]
        add     a,      8[usp]
        st      a,      dp

;;}
CLINE 4
        pops    usp
        rt

CFUNCTION 1
_add_long :

;;{
CLINE 7
        pushs   usp
        mov     usp,    ssp

;;      return ( a + b ) ;
CLINE 8
        l       a,      10[usp]
        add     a,      6[usp]
        st      a,      dp
        l       a,      12[usp]
        adc     a,      8[usp]
        st      a,      x1

;;}
CLINE 9
        pops    usp
        rt

CFUNCTION 2
_add_double     :

;;{
CLINE 13
        pushs   usp
        mov     usp,    ssp

;;      return ( a + b ) ;
CLINE 14
        l       a,      22[usp]
        pushs   a
        l       a,      20[usp]
        pushs   a
        l       a,      18[usp]
        pushs   a
        l       a,      16[usp]
        pushs   a
        l       a,      14[usp]
        pushs   a
        l       a,      12[usp]
        pushs   a
        l       a,      10[usp]
        pushs   a
        l       a,      8[usp]
        pushs   a
        cal     __dadds50s
        mov     dp,     6[usp]
        mov     x1,     SSP
        mov     [dp+],  0ah[x1]
        mov     [dp-],  0ch[x1]
        mov     4[dp],  0eh[x1]
        mov     6[dp],  010h[x1]
        add     SSP,    #010h

;;}
CLINE 15
        pops    usp
        rt
```

8.5.4 Interrupt Handling Routines In Assembly

CC665S 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.

8.5.5 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”.

8.6 QUALIFYING FUNCTIONS WITH ‘__accpass’ AND ‘__noacc’

A function may be qualified with **__accpass** to inform the compiler to use Accumulator for it’s first argument and for the return value.

If a function is qualified with **__accpass** and the size of the first argument is less than or equal to 2 bytes, then the value of the first argument is stored in the Accumulator. The function accesses the first argument using Accumulator. However, if the size of the first argument is greater than 2 bytes, first argument processing will be done as for other arguments. However, usage of Accumulator for first argument is more efficient than using stack.

Similarly, when the function is qualified with **__accpass**, the compiler places the return value in the Accumulator, therefore reducing the register movement for the return values. However, if the size of return value is greater than 2 bytes, Accumulator is not used to store the return value.

A function may also be qualified with **__noacc**. This qualifier instructs the compiler not to use Accumulator for it’s first argument and for the return value.

If **/REG** option is specified in the command line, all functions except those qualified with **__noacc**, are treated as **__accpass** qualified functions. Arguments for library functions must always be passed through stack as they may be invoked from a function which might or might not have been compiled using **/REG** option. Therefore, all library routines must be qualified with **__noacc**.

Example 8.5

INPUT

```
int __accpass acc_add ( int a, int b );
int var1, var2 ;

int __accpass acc_add ( int a, int b )
{
        int     l_ret   ;
        l_ret = a + b ;
        return ( l_ret ) ;
}

fn ()
{
        var1 = acc_add ( var1, var2 ) ;
}
```

OUPUT

```
CFUNCTION 0
_acc_add        :

;;{
CLINE 6
        pushs   usp
        mov     usp,    ssp

;;      l_ret = a + b ;
CLINE 8
        add     a,      6[usp]

;;}
CLINE 10
        pops    usp
        rt

CFUNCTION 2
_fn     :

;;      var1 = acc_add ( var1, var2 ) ;
CLINE 14
        l       a,      dir _var2
        pushs   a
        l       a,      dir _var1
```

```
                cal     _acc_add
                add     SSP,    #02h
                st      a,      dir _var1

        ;;}
        CLINE 15
                rt
```

If /REG option is not specified in the command line, the proto type of all functions qualified with **__accpass** must be declared before the function call, to obtain intended results.

8.7 BUILT-IN FUNCTIONS

CC665S supports built-in functions for high-precision multiplication, high-precision division and high-precision remainder (mod) operations. 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. CC665S issues error message if a built-in function is defined in the source file.

CC665S issues warning message, if an incompatible parameter is passed to a built-in function. However, the compiler converts the actual parameter to the formal parameter type.

CC665S issues error message, if number of actual parameters does not agree with the prototype.

The following sections explain the built-in functions in detail:

8.7.1. Higher Precision Multiplication

Prototypes:

unsigned long __mulu(unsigned int, unsigned int) ;
unsigned int __mulbu(unsigned char, unsigned char) ;

The function "**__mulu**" uses MUL instruction to multiply two 2-byte operands and returns a 4-byte value. The function "**__mulbu**" uses MULB instruction to multiply two 1-byte operands and returns a 2-byte value.

Example 8.6

INPUT:

```
        unsigned long long_var ;

        unsigned int var1 ;
        unsigned int var2 ;

        void fn1 ()
        {
                long_var =  __mulu ( var1, var2 ) ;
        }
```

The following is the code generated for the function “fn1” defined in the above program:

OUTPUT

```
        CFUNCTION 0
        _fn1        :

        ;;          long_var =  __mulu ( var1, var2 ) ;
        CLINE 6
                    l           a,          dir _var1
                    mul         dir _var2
                    mov         dir _long_var,      er0
                    st          a,          dir _long_var+02h

        ;;}
        CLINE 7
                    rt
```

Example 8.7

INPUT

```
        long long_var ;

        unsigned int var1 ;
        unsigned char var2 ;

        void fn2 ()
        {
                long_var =  __mulbu ( var1, var2 ) ;
        }
```

The following is the code generated for the function “fn2” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn2     :

;;       long_var =  __mulbu ( var1, var2 ) ;
CLINE 8
         lb       a,       dir _var1
         mulb     dir _var2
         sdd
         st       a,       dir _long_var
         clr      dir _long_var+02h

;;}
CLINE 9
         rt
```

Example 8.8

INPUT

```
long __mulbu (char arg1, char arg2) ;
```

In the above example, CC665S issues an error as the prototype of built-in function “**__mulbu**” is redefined.

8.7.2. Higher Precision Division

Prototypes:

unsigned long __divu(unsigned long, unsigned int) ;
unsigned int __divqu(unsigned long, unsigned int) ;
unsigned int __divbu(unsigned int, unsigned char) ;

The function “**__divu**” uses DIV instruction to divide a 4-byte value by a 2-byte value and returns a 4-byte quotient. The function “**__divqu**” uses DIVQ instruction to divide a 4-byte value by a 2-byte value and returns a 2-byte quotient. The function “**divbu**” uses DIVB instruction to divide a 2-byte value by a 1-byte value and returns a 2-byte quotient.

Example 8.9

INPUT

```
        unsigned long var1 ;
        unsigned int var2 ;
        unsigned long var3 ;

        void fn1 ()
        {
                var3 =  __divu ( var1, var2 ) ;
        }
```

The following is the code generated for the function “fn1” defined in the above program:

OUTPUT

```
        CFUNCTION 0
        _fn1    :

        ;;      var3 =  __divu ( var1, var2 ) ;
        CLINE 7
                mov     er0,    dir _var1
                l       a,      dir _var1+02h
                div     dir _var2
                mov     dir _var3,      er0
                st      a,      dir _var3+02h

        ;;}
        CLINE 8
                rt
```

Example 8.10

INPUT

```
        unsigned long var1 ;
        unsigned int var3 ;

        void fn2 ()
        {
                var3 =  __divqu ( var1, (int) var1 ) ;
        }
```

The following is the code generated for the function “fn2” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn2     :

;;       var3 =  __divqu ( var1, (int) var1 ) ;
CLINE 6
         mov     er0,     dir _var1
         l       a,       dir _var1+02h
         divq    dir _var1
         st      a,       dir _var3

;;}
CLINE 7
         rt
```

Example 8.11

INPUT

```
unsigned char var1 ;
unsigned long var2 ;

void fn3 ()
{
         var2 =  __divbu ( var1 , 0xff ) ;
}
```

The following is the code generated for the function “fn3” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn3     :

;;       var2 =  __divbu ( var1 , 0xff ) ;
CLINE 6
         lb      a,       dir _var1
         extnd
         fillb   r1
         divb    r1
         st      a,       dir _var2
         clr     dir _var2+02h

;;}
CLINE 7
         rt
```

8.7.3. Higher Precision Remainder

Prototypes:

unsigned int __modu(unsigned long, unsigned int) ;
unsigned int __modqu(unsigned long, unsigned int) ;
unsigned char __modbu(unsigned int, unsigned char) ;

The function “**__modu**” uses DIV instruction to divide a 4-byte value by a 2-byte value and returns a 2-byte remainder. The function “**__modu**” uses DIVQ instruction to divide a 4-byte value by a 2-byte value and returns a 2-byte remainder. The function “**__modbu**” uses DIVB instruction to divide a 2-byte value by a 1-byte value and returns a 1-byte remainder.

Example 8.12

INPUT

```
unsigned long var1 ;
unsigned int var2 ;
unsigned int var3 ;

void fn1 ()
{
        var3 =  __modu ( var1, var2 ) ;
}
```

The following is the code generated for the function “fn1” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn1        :

;;          var3 =  __modu ( var1, var2 ) ;
CLINE 7
            mov     er0,        dir _var1
            l       a,          dir _var1+02h
            div     dir _var2
            mov     dir _var3,          er1

;;}
CLINE 8
            rt
```

Example 8.13

INPUT

```
unsigned int var ;

void fn2 ()
{
        var =  __modqu ( 0x100000l , var ) ;
}
```

The following is the code generated for the function “fn2” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn2    :

;;      var =  __modqu ( 0x100000l , var ) ;
CLINE 5
        clr     er0
        l       a,      #010h
        divq    dir _var
        mov     dir _var, er1

;;}
CLINE 6
        rt
```

Example 8.14

INPUT

```
unsigned char var1 ;
unsigned int var2 ;

void fn3 ()
{
        var1 =  __modbu ( __mulu( var2, var1) , var1 ) ;
}
```

The following is the code generated for the function “fn3” defined in the above program:

OUTPUT

```
CFUNCTION 0
_fn3    :

;;      var1 =  __modbu ( __mulu( var2, var1) , var1 ) ;
CLINE 6
        lb      a,      dir _var1
        extnd
        mul     dir _var2
        l       a,      er0
        divb    dir _var1
        lb      a,      r1
        stb     a,      dir _var1

;;}
CLINE 7
        rt
```

8.8 RUNTIME STACK PREPARATION

The runtime stack preparation is carried out at the beginning of each function. CC665S uses the register usp or x2 as the base pointer. Since usp can be accessed only within 64 bytes range, usp will be used for accessing local/arguments, if the size of locals and size of arguments do not exceed 64 bytes each, otherwise x2 is used for the same purpose.

Memory required for local variables used in a function is allocated in stack in the entry code of the function. The allocated memory is freed in the exit code of the function since the scope of the local variables are limited to this function.

Example 8.15

INPUT

```
int fn (int arg)
{
        int a [10];
        return a[arg] ;
}
```

The following is the code generated for the above defined function:

OUTPUT

```
CFUNCTION 0
_fn     :

;;{
CLINE 2
        pushs   usp
        mov     usp,    ssp
        sub     ssp,    #014H
;;          return a[arg] ;
CLINE 4
            l       a,      6[usp]
            sll     a,      01h
            add     a,      usp
            st      a,      x1
            mov     dp,     0ffeeh[x1]

;;}
CLINE 5
            mov     ssp,    usp
            pops    usp
            rt
```

In the above example, base pointer **usp** is pushed into the stack at the beginning of the function. Stack pointer is moved into the base pointer (usp). The memory space required for the local variables used in the function is allocated by subtracting stack pointer (ssp) by constant 0x14. The allocated memory space is freed in the exit code by restoring the old value of stack pointer (ssp) from the base pointer (usp).

8.9 REGISTER USAGE

In MSM66K “500” core and “500S” core architectures, the local registers and pointing registers must be allocated in the startup code.

The PRBANK pseudo instruction aids in allocating the pointing register set used by CC665S. This enables the linker RL66K to allocate the specified pointing register set.

The LRBANK pseudo instruction aids in allocating the local register set used by CC665S.

For example,

```
prbank 0
lrbank 8
```

The above pseudo instructions in the startup code enables the linker RL66K to allocate pointing register set PR0 and local register set LR8.

Emulation library routines use the same set of pointing registers and local registers as used by other routines compiled using CC665S.

Register usp or x2, depending on locals/arguments size, is reserved by CC665S for use as base pointer, hence it is not used for other purposes, however, the other register is used freely for storage and indexing purposes.

Accumulator, the pointing registers x2/usp, x1 and dp and the local registers er0, er1, er2 and er3 are freely used by CC665S in code generation.

Registers dp and x1 are used to carry the return values of ‘C’ functions. If a function returns a value of size less than or equal to 2 bytes, then it is returned in the register dp. In case, a function returns a value of size greater than 2 bytes, then the return value is in the register pair dp and x1. The lower word of the value is in the register dp and the higher word in x1. If the value returned is that of a structure/union, then dp and x1 are not used for returning from the function.

Accumulator is used for passing the first argument to **__accpass** qualified functions. Similarly, accumulator is used to store the return value of **__accpass** qualified function.

8.10 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

```
        j          _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 added to long66.lib or float66.lib, or directly specified while invoking RL66K.

9. EMULATION LIBRARIES

CC665S supports the data type **long**, **float** and **double** although the MSM66K "500" core and "500S" core 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 two library files float66.lib and long66.lib. All arithmetic operations involving **long**, **float** and **double** data types are carried out with the help of these routines. CC665S outputs a call instruction to the appropriate routine to perform the arithmetic operation. Separate routines are provided for nX-8/500 and nX-8/500S. CC665S invokes the appropriate nX-8/500 or nX-8/500S routine based on the core option specified in the command line. These routines are provided for all the memory models, namely - Small, Effective medium, Medium, Compact, Effective large and Large. Separate emulation routines are provided to be called from near, far and large functions.

Following routines are stored in the emulation library:

1. Long multiplication
2. Signed long division
3. Unsigned long division
4. Signed long modulus
5. Unsigned long modulus
6. Signed integer division
7. Signed integer modulus
8. Float addition
9. Float subtraction
10. Float multiplication

11. Float division
12. Float comparison
13. Float negation
14. Double addition
15. Double subtraction
16. Double multiplication
17. Double division
18. Double comparison
19. Double negation
20. Long to float conversion
21. Unsigned long to float conversion
22. Long to double conversion
23. Unsigned long to double conversion
24. Float to long conversion
25. Double to long conversion
26. Float to double conversion
27. Double to float conversion
28. Indirect far call
29. Checkstack

10. ASSEMBLING AND LINKING THE COMPILER OUTPUT

CC665S 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 RAS66K. To invoke the assembler the following command line should be used.

```
C:> RAS66K FILE <CR>
```

Where 'FILE' specifies the name of the output file created by the Compiler, CC665S. 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 CC665S generates code that is case sensitive. By default RAS66K 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 RAS66K as shown below:

```
C:> RAS66K FILE /CD <CR>
```

The assembler produces as output an object file. In order to debug the C programs using CDB665S, the compiler output should be assembled using the '/CC' option as follows :

```
C:> RAS66K FILE /CC <CR>
```

This option informs the assembler to create the object file with necessary debugging information. This option must be specified to RAS66K, only when the files are compiled with /SD option in CC665S.

The object files created by RAS66K can be linked using the Object Linker RL66K. The linker produces as output an absolute object file.

To link the object programs, the following command line should be used :

```
C:> RL66K FILE1 FILE2,....,,/CC <CR>
```

where ‘FILE1 FILE2,...’ are the names of the input object files to RL66K. The /CC option informs RL66K, that the inputs are files compiled by CC665S and assembled using RAS66K. Thus RL66K would take appropriate steps to reserve space for the stack and to initialize the stack pointer.

The output assembly file created by CC665S makes use of routines, which are available in the libraries ‘float66.lib’ and ‘long66.lib’. RL66K searches these two library files to resolve the externals. The RL66K searches these library files in standard directories specified by the environment variable LIB66K when /CC option is specified. The environment variable can be set by the following command at the DOS prompt.

```
C:> SET LIB66K=directory <CR>
```

Where ‘directory’ gives the name of the standard directory which will be used by RL66K to search the library files.

In order to create absolute files with debugging information for CDB665S, the object files should be linked using the /SD option as follows :

```
C:> RL66K FILE1 FILE2,....,,/CC /SD <CR>
```

This option informs the linker to create the absolute file with necessary debugging information.

11. EXIT CODES

CC665S, on termination passes the control to the operating system, while passing the control to the operating system, CC665S returns a numeric value called exit code. The exit codes and the corresponding exit status are listed below.

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 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 indicate 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 CC665S :

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 or a nonconstant was specified with /SS option.

F0010 Invalid stack size.

The constant specified with /SS option must be an even number, in the range 2 - 65534, inclusive of both.

F0011 Stack size should be even

Stack size specified with /SS option should be an 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

More than one C memory model option was specified in the command line or more than one mixed memory model option was specified.

F0017 Duplicate core option

Both the core options /nX500 and /nX500S were specified together.

F0018 Duplicate debugger option.

Both the debugger options /SD and /OSD were specified in the command line.

F0019 /CT and preprocessor options are mutually exclusive

The option /CT option was specified along with /LP or /PC options.

F0020 /LE and preprocessor options are mutually exclusive

The option /LE was specified along with /LP or /PC options.

F0021 /Fa and preprocessor options are mutually exclusive.

/Fa option was specified along with either /LP or /PC options.

F0022 /WIN and /AWIN options are mutually exclusive.

Both /WIN and /AWIN options were specified in the command line.

F0023 Illegal combination of optimization options.

The optimization options were used incorrectly.

F0024 Illegal combination of C and mixed memory model options.

Invalid combination of C and mixed memory model options was specified.

F0025 Type is not specified

One of the compulsory options /T was not specified.

F0026 Mixed memory model should be specified with C memory model.

A mixed memory model option was specified without specifying a C memory model option.

F0027 Insufficient memory

The compiler ran out of memory.

F0028 Unable to open input file '*filename*'

The given '*filename*' either did not exist or could not be opened or was not found.

F0029 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.

F0030 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.

F0031 Unable to open calltree file

The compiler could not open the calltree file, due to similar reasons mentioned in error F0030.

F0032 Error in accessing the input file

The compiler was unable to access the input file while compiling.

12.1.2 General

F1000 File close error

The compiler was unable to close input/output file. This error results due to insufficient disk space.

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 CC665S.

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

Possible overflow in floating point arithmetic.

F1006 Unable to read input file

The compiler was unable to read/access the input file during the compilation process.

F1007 Error in creating debug information file

The compiler could not create the debug information file, due to similar reasons mentioned in error F0030.

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.

E2002 Parameter expected after ‘#’

The token following a stringizing operator (#) must be a formal parameter.

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

A 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.

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 functions only

An object that is not of type function, was qualified with either __nfar, __accpass, __noacc or __interrupt.

E4043 Segment lost during conversion

An attempt was made to convert a far pointer to near pointer.

E4044 A far function cannot call near function

An attempt was made to call a near function from a far function.

E4045 Function specified in Cal pragma cannot be called from near/nfar functions

An attempt was made to call a function specified in Cal pragma, from a near or nfar function.

E4046 More than one ‘*qualifier*’ qualifier specified

On of the function qualifiers __accpass, __noacc or __interrupt was specified more that once.

E4047 Illegal combination of __accpass and __noacc

A function was qualified with both __accpass and __noacc.

E4048 Illegal combination of __far and __nfar

A function was qualified with both __far and __nfar.

E4049 Illegal combination of __far/__nfar and __interrupt

A function was qualified with __far or __nfar is also qualified with __interrupt.

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 as they reside in code memory (ROM). 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 Cast to different memory

A code memory object was cast to a data memory object or vice-versa.

E5036 Indirection to different memory

Indirection was used in an expression to access values from different memory address spaces.

E5037 Too few actual parameters

Actual parameters passed to a function could be less than number of parameters formally specified.

E5038 Too many actual parameters

Actual parameters passed to a function could have exceeded number of parameters formally specified.

E5039 Void function returning value

The function was defined to return no value with the ‘void’ keyword but the function returns a value.

E5040 Illegal sizeof operand

A bit field could have been specified as an operand for sizeof operator.

E5041 ‘*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.

E5042 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

12.3.1 Preprocessor

W2000 ‘#undef’ ignored for predefined macro ‘identifier’

An attempt might have been made to undef the predefined macro ‘identifier’.

W2001 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 ‘#define’ ignored for predefined macro ‘macroname’

An attempt might have been made to install predefined ‘macroname’ as a macro.

W2003 Close bracket expected

Missing ‘)’ in a macro definition or in macro call.

W2004 Unexpected characters following directive ‘directive’

Extra characters found after processing a preprocessor directive.

W2005 Redefinition of macro ‘*identifier*’

The given identifier was redefined.

W2006 Comma separator missing

The formal parameters list in a macro definition must be separated by commas.

W2007 Argument expected before ‘*character*’

An argument was expected in macro call

W2008 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.

W2009 Expected an identifier, found no token

Expecting a valid identifier.

12.3.2 Lexical

W3000 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 String too long, truncated

The length of the string exceeded 1023 characters.

12.3.3 Syntactic And Semantic

W4000 Auto/Register ignored for global variables

An attempt was made to declare global variable with auto/register storage class.

W4001 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 ‘const’ ignored on argument

Since function formal parameters are allocated in stack, ‘const’ is ignored on formal parameter.

W4003 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 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 ‘const’ ignored for struct/union member ‘*identifier*’

‘const’ qualified variables are not allowed in struct/union.

W4006 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 ‘*identifier*’ : array bound overflows

Too many initializer were present for the array. The excess initializers are ignored.

W4008 Parameter *number* declaration different

Type of parameter declaration in prototype could be different from formal declaration.

W4009 Declared subscripts for arrays different

Two operands to an operation are arrays whose declared subscripts could be different.

W4010 Function was declared with variable arguments

There was a parameter(s) mismatch between prototype and actual definition of a function.

W4011 Function was not declared with variable arguments

There was a parameter(s) mismatch between prototype and actual definition of a function.

W4012 ‘const’ ignored on local variable ‘*identifier*’

All ‘const’ qualified variables are allocated in the code memory. But local variables are allocated in stack, hence, ‘const’ is ignored on local variables.

W4013 No declaration specifiers ; ‘int’ assumed

The variable was declared without any declaration specifiers. Type specifier ‘int’ is assumed for the variable.

W4014 Sign information ignored for bit field

A bit field member was decalared as signed.

W4015 memory attribute on cast ignored

The memory qualifier in the cast expression is qualiying a non-pointer object.

W4016 const object modified

An object qualified with const has been modified under /WIN option.

W4017 __far ignored on struct/union member ‘*identifier*’

‘__far’ qualified variables are not allowed in struct/union.

W4018 __far ignored on local variable ‘*identifier*’

Since local variables are allocated in stack, they cannot be qualified with __far.

W4019 __far ignored on argument variable ‘*identifier*’

Since arguments are allocated in stack, they cannot be qualified with __far.

W4020 __far not allowed for *memory type* memory

The mixed memory model option specified in the command line or assumed by the compiler does not support far code memory or far data memory.

W4021 __far/__nfar functions not allowed

The mixed memory model option specified or assumed does not support __far/__nfar functions.

W4022 Indirection to different types

Pointers used in the expression were pointing to different memory (Pointer size mismatch).

W4023 __far/__nfar ignored for ‘main’

Function ‘main’ was qualified with __far/__nfar.

W4024 __accpass/__noacc ignored for ‘main’

Function ‘main’ was qualified with __accpass/__noacc.

W4025 __interrupt ignored for ‘main’

Function ‘main’ was qualified with __interrupt.

W4026 Missing return value for function ‘*function name*’

The function was declared to return a value, but returns without one.

W4027 ‘*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.

12.3.4 Expression

W5000 ‘*identifier*’ function used as an argument

An attempt was made to pass function as an argument.

W5001 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 ‘*operator*’ : different levels of indirection

An expression had inconsistent levels of indirection.

W5003 Atleast one void operand

An expression with type ‘void’ was used as an operand.

W5004 ‘&’ on array ignored

An attempt was made to apply the address of operator (&) to an array.

W5005 Constant too large, converted to ‘int’

The constant specified in the case statement, exceeded the maximum integer value.

W5006 Division by zero

The second operand in a division operation (/) evaluated to zero. Hence it was converted to one.

W5007 Mod by zero

The second operand in a remainder operation (%) evaluated to zero. Hence it was converted to one.

W5008 ‘*operator*’ : indirection to different types

The indirection operator (*) was used in an expression to access values of different types.

W5009 Function Parameter lists differed

The type of the formal parameter did not agree with corresponding type in the function declaration (prototype).

W5010 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 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.

W5012 Parameter mismatch, actual parameter converted

Type in actual parameter declaration was different from formal parameter declaration. Appropriate conversions are performed.

12.3.5 Pragmas

W8000 Expected a pragma keyword, found no token

A valid pragma keyword was expected after the preprocessor directive ‘#pragma’, found no token.

W8001 Unknown pragma ‘*token*’

An invalid keyword was specified with the preprocessor directive ‘#pragma’.

W8002 ‘main’ cannot be specified in ‘*pragma keyword*’ pragma.

Function ‘main’ was specified in pragma ‘*pragma keyword*’. It may be specified only in pragma Usinginpage.

W8003 ‘*pragma keyword*’ pragma variables should be global or static local

The specified variable was neither a global variable nor a static local variable.

W8004 Vector address out of range for pragma ‘*pragma keyword*’

The vector address specified in either Interrupt, Intvect or Vcal pragma was out of range.

The valid range of vector addresses are as follows :

Interrupt	-	0x8	to	0xfffe
Intvect	-	0x8	to	0xfffe
Vcal	-	0x4a	to	0x68

W8005 Expected even vector address, for pragma ‘*pragma keyword*’

An odd vector address was specified in either Interrupt, Intvect or Vcal pragma.

W8006 More than one function for the same vector address

Two different functions were specified with same vector address in pragma Interrupt, Intvect or Vcal.

W8007 Pragma argument delimiter ‘,’ expected

The pragma argument delimiter ‘,’ (comma) was expected, as /PF option was specified in the command line.

W8008 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, Intvect, Vcal or Usinginpage when the specified function was already defined prior to the pragma directive.

W8009 Interrupt function has parameter/return value

Functions specified in pragma Interrupt/Intvect either has parameters or returns a value or both.

W8010 ‘*pragma keyword*’ address exceeds range

The address specified either in absolute or sfr pragma was out of range. The valid range of addresses are as follows

The valid range of vector addresses are as follows :

Absolute (code)	-	0x0	to	0xffff
Absolute (data)	-	0x0	to	0xffff
Sfr	-	0x0	to	0x1ff

W8011 Pragma must appear before variable initialization.

The variable specified in pragma was initialized prior to the pragma directive.

W8012 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.

W8013 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 0x2and 0xfffe inclusive of both.

W8014 Expected even number as stack size

Size specified with pragma Stacksize was not an even number.

W8015 More than one pragma specified for variable ‘*variable*’

‘*variable*’ was specified in more than one pragma.

W8016 Different page numbers for the same segment ‘*segment name*’

Two different page numbers were specified for a segment ‘*segment name*’ in two different instances of pragma Inpage or Sbainpage.

W8017 ‘*pragma keyword*’ pragma expects function name

The specified symbol was not a function. Interrupt, Intvect, Vcal, Acal, Cal, and Usinginpage pragma expects a function name to be specified.

W8018 Page number out of range

Page number specified in pragma Inpage or Sbainpage was out of range. The valid range of the page number is from 0 to 255 inclusive of both.

W8019 ‘Window’ pragma ignored

The window pragma which is not supported was ignored.

W8020 Pragma keyword expected, found no token

No token was found after ‘# pragma’.

W8021 Unexpected characters following pragma ‘*pragma keyword*’

Unexpected characters was found after a valid pragma ‘*pragma keyword*’

W8022 Function cannot be specified in pragma ‘*pragma keyword*’

Variable declared as a function was specified in pragma ‘*pragma keyword*’.

W8023 Enum constants are not allowed in pragma.

An enum constant was specified in pragma directive.

W8024 ‘Absolute/Sfr’ address leads to odd boundary access

This warning message is issued due to one of the following reasons:

* An odd address was specified for an initialized variable
* An odd address outside SFR region was specified for uninitialized variables of type other than **char** and array of **char**

W8025 Invalid ‘Absolute’ address for the variable ‘token’

The Absolute address specified in the pragma Absolute for the variable ‘token’ excceded 0xffff.

W8026 ‘__interrupt’ qualified function cannot be specified in pragma ‘*pragma keyword*’

An ‘__interrupt’ qualified function was specified in pragma ‘*pragma keyword*’. A function qualified by ‘__interrupt’ may be specified only in Interrupt, Intvect and Usinginpage pragmas.

W8027 Interrupt function ‘*function name*’ used in expression

Function ‘*function name*’ specified in Interrupt/Intvect pragma was used in an expression. Functions specified in these pragmas should not be called directly or indirectly in a ‘C’ program.

W8028 Constant expected, found no token

A constant was expected in the #pragma directive, but found no token

W8029 Constant expected, found ‘*token*’

A constant was expected in the #pragma directive, but found ‘*token*’.

W8030 ‘Common’ pragma ignored

The common pragma which is not supported, was ignored.

W8031 Pragma syntax error

The specified ‘#pragma’ syntax was not recognized by CC665S.

W8032 ‘*segment name*’ cannot be specified along with far segments in ‘Group’ pragma

An attempt was made to mix near segments and far segments in pragma Group.

W8033 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.

W8034 Identifier or constant expected for pragma, found no token

An identifier or constant was expected in the #pragma directive, but found no token.

W8035 Identifier or constant expected for pragma, found ‘*token*’

An identifier or constant was expected in the #pragma directive, but found ‘*token*’.

W8036 Group segment ‘*segment name*’ not in pragma ‘Inpage/sbainpage’

The segment ‘*segment name*’ specified in pragma group was not specified in pragma Inpage/Sbainpage prior to this Group directive.

W8037 Close bracket expected, found no token

A close bracket was expected in the ‘# pragma’ directive, but found no token.

W8038 Close bracket expected, found ‘*token*’

A close bracket was expected in the ‘# pragma’ directive, but found ‘*token*’.

W8039 Identifier expected for pragma, found no token

An identifier was expected in the ‘# pragma’ directive, but found no token.

W8040 Identifier expected for pragma, found ‘*token*’

An identifier was expected in the ‘# pragma’ directive, but found ‘*token*’.

W8041 ‘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 pragma Romwindow and Absolute.

W8042 Unexpected ‘Endasm’ pragma ignored

‘Endasm’ pragma was specified without its correponding Asm pragma.

W8043 Expected an identifier or ‘-lrb’ option, found no token

An identifier or ‘-lrb’ option was expected after pragma keyword Usinginpage, but found no token.

W8044 Expected an identifier or ‘-lrb’ option, found ‘*token*’

An identifier or ‘-lrb’ option was expected after pragma keyword Usinginpage, but found ‘*token*’.

W8045 Identifier or constant expected for pragma, found ‘,’

An Identifier or constant was expected in the ‘# pragma’ directive, but found ‘,’.

W8046 Segment number exceeds range

The specified segment number was not in the range 0 to 255 inclusive of both.

W8047 ‘Romwindow’ variables should be qualified with ‘const’

Variable specified in pragma Romwindow was not qualified with ‘const’.

W8048 ‘Commonvar’ pragma can be specified only for large data memory models

The source file was not compiled in memory model options that support large data.

W8049 Expected ‘__interrupt’ qualified function for ‘Intvect’ pragma

The function specified in Intvect pragma was not qualified with ‘__interrupt’.

W8050 Invalid ‘Sfr’ address for the variable ‘*token*’

The address specified in Sfr pragma was not in sfr area. The address should be in the range 0x0 to 0x1ff.

W8051 Segment ‘*segment name*’ specified in ‘Group’ pragma is not defined

The variables specified in the segment ‘*segment name*’ was not declared in the source file. A segment is defined only when a variable specified in that segment is declared in the source file.

W8052 Segment should be 0 for ‘near’ variables

A non-zero segment was specified for near variables.

W8053 Segment ‘*segment name*’ not defined in ‘Inpage/sbainpage’ pragma

The segment ‘*sement name*’ specified in Usinginpage pragma was not defined in Inpage or Sbainpage pragma prior to this directive.

W8054 Page ‘*pageno*’ not specified in ‘Inpage/sbainpage’ pragma

The page number ‘*pageno*’ specified in Usinginpage pragma was not defined in Inpage or Sbainpage pragma prior to this directive.

W8055 Segment ‘*segment name*’ already specified in ‘Inpage/sbainpage’ pragma

The segment ‘*segment name*’ specified in Inpage pragma was already specified in Sbainpage pragma or the segment ‘segment_name’ specified in Sbainpage pragma was already specified in Inpage pragma.

W8056 ‘Absolute’ pragma expects segment address for ef-near/ef-xnear variables

Segment was not specified for effective-near/effective-xnear variable in absolute pragma.

W8057 Far/nfar functions cannot be specified in pragma ‘*pragma keyword*’

A far/nfar function was specified in pragma ‘*pragma keyword*’. Functions qualified with __far/__nfar cannot be specified in Interrupt, Intvect and Vcal pragmas.

W8058 Far variable cannot be specified in pragma ‘*pragma keywords*’

A far variable was specified in pragma ‘*pragma keyword*’. Variables qualified with __far cannot be specified in Fix, Sbafix, Dual, Edata and Commonvar pragmas.

W8059 ‘*function name*’ specified in ‘Acal’ pragma is not near, static far or static large function

The function specified in Acal pragma was not near, static far or static large function.

W8060 ‘*function name*’ specified in ‘Cal’ pragma is not static far or static large function

The function specified in Cal pragma should be static far or static large function.

W8061 Illegal combination of near and static far functions in ‘Acal’ pragma

Near and static far functions cannot be specified in the same Acal pragma.

W8062 Illegal combination of near and far variables in pragma ‘*pragma keyword*’

Near and far variables cannot be specified in same Inpage or Sbainpage pragma.

W8063 Identifier expected for pragma, but found ‘,’

An identifier was expected in the ‘# pragma’ directive, but found ‘,’.

W8064 Constant expected for pragma, found ‘,’

A constant was expected in the ‘# pragma’ directive, but found ‘,’.

W8065 ‘*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:

∗ The inline function was recursive.
∗ 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

Part2.
CC665S Ver.2.01
Language Reference

Table Of Contents

1. PREPROCESSOR

1.1 INTRODUCTION

CC665S may be invoked with /LP or /PC option so as to process text without compiling. CC665S behaves like a text processor that manipulates the text of a source file, when invoked with /LP or /PC option.

Preprocessor performs the following functions

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

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

1.2 TRANSLATION PHASES

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

1. Trigraph sequences are replaced by the corresponding single-character internal representations.

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

1.2.1 Trigraph sequences

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

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

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

Example 1.1

INPUT:

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

OUTPUT:

```
main ()
{
}
```

1.2.2 Line Splicing

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

1.3 MACROS

1.3.1 Introduction

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

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

a) #define
b) #undef

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

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

a) stringizing (#)

b) token pasting (##).

1.3.2 Macro Definition

1.3.2.1 Defining Macros Without Parameters

Syntax :

*# **define** identifier token_sequence*

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

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

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

Example 1.2

# define ABC 1 + 2

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

1.3.2.2 Defining Macros With Parameters

Syntax :

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

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

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

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

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

Example 1.3

# define ABC(x,y) x + y

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

1.3.2.3 Operators In Macro Processing

1.3.2.3.1 Stringizer

Operator symbol : #

Syntax :

# parameter

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

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

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

Example 1.4

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

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

1.3.2.3.2 Token Paster

Operator symbol : ##

Syntax :

token ## token

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

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

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

Example 1.5

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

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

1.4 MACRO EXPANSION

1.4.1 Expansion Of Macros Without Parameters

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

Example 1.6

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

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

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

Example 1.7

```
# define ONE              THREE
# define TWO              2
# define THREE            3
```

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

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

Example 1.8

```
# define ONE              TWO
# define TWO              THREE
# define THREE            TWO
```

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

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

Example 1.9

```
# define ABCD          a + b + c+d
# define XYZ           a/* abcde */+ 2
```

MACRO CALL	REPLACEMENT TEXT
ABCD	a + b + c+d
XYZ	a + 2

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

Example 1.10

```
# define ONE           1
```

MACRO CALL	REPLACEMENT TEXT
“ONE”	“ONE”

1.4.2 Expansion Of Macros With Parameters

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

identifier[white space]([actual_argument_list])

Example 1.11

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

MACRO CALL	REPLACEMENT TEXT
x=MUL(23,43) - ADD(12,400)	x=(23 * 43) - 12 + 400

ARGUMENTS OF MACRO CALL

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

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

Leading and trailing spaces in each argument are discarded.

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

The arguments may run through more than one line.

Example 1.12

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

MACRO CALL	REPLACEMENT TEXT
ADD(1,2)	1 + 2
MUL(12,2)	(12 * 2)
ADD(xxx(1,2),3)	xxx(1,2) + 3
ADD(1 + 2,3)	1 + 2 + 3
ADD (11,3)	11 + 3
ADD (12345 + 678, 9)	12345 + 678 + 9

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

Example 1.13

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

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

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

Example 1.14

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

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

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

Example 1.15

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

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

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

Example 1.16

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

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

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

Example 1.17

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

MACRO CALL	REPLACEMENT TEXT
XCAT(XCAT(1,2),3)	123

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

1.5 MACRO REMOVAL

Syntax :

*# **undef** identifier*

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

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

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

Example 1.18

```
# define ONE              1

x = ONE ;

# undef                   ONE

y = ONE

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

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

1.6 REDEFINITION OF MACROS

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

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

The following redefinitions are erroneous.

Example 1.19

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

Example 1.20

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

Example 1.21

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

Example 1.22

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

Example 1.23

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

The following redefinitions are nonerroneous.

Example 1.24

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

Example 1.25

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

Example 1.26

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

1.7 FILE INCLUSION

1.7.1 Introduction

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

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

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

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

1.7.2 Include File Specification Using Double Quotation Marks

Syntax :

# include *"filename"*

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

a) directory of parent files is searched (parent file is the file containing the # include directive)
b) the directories of any grand parent files

c) the directories specified using /I command line option
d) the standard directory set by the environment variable INCL66K.

Example 1.27

```
# include “\dir1\dir2\abc.c”
```

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

1.7.3 Include File Specification Using Angle Brackets

Syntax :

*# **include** <filename>*

The filename may contain a path specification.

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

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

1.7.4 Macros In Include Directive

Syntax :

*# **include** token_sequence*

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

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

a) filename specified within double quotation marks

b) filename specified within angle brackets.

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

Example 1.28

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

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

1.8 CONDITIONAL COMPILATION

1.8.1 Introduction

The following preprocessor directives are used for conditional compilation.

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

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

1.8.2 Conditional Compilation Directives

Syntax :

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

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

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

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

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

Example 1.29

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

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

Example 1.30

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

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

Example 1.31

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

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

Example 1.32

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

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

Example 1.33

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

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

1.8.3 Restricted Constant Expression

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

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

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

Example 1.34

```
# if 1 +2
```

Example 1.35

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

Example 1.36

```
# if A
```

Example 1.37

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

Example 1.38

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

Example 1.39

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

Example 1.40

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

The following are erroneous:

Example 1.41

```
# if A = 2
```

Example 1.42

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

Example 1.43

```
# if X ++
```

Example 1.44

```
# if &X
```

Example 1.45

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

Example 1.46

```
# if A, C
```

Example 1.47

```
# if 1.2
```

1.8.4 defined Operator

Syntax :

defined identifier
defined (identifier)

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

Example 1.48

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

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

The defined operator may also appear with other operators.

Example 1.49

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

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

1.8.5 Nesting

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

Example 1.50

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

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

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

Nesting level is restricted to 32.

1.8.6 Testing Symbol Definition With #ifdef and #ifndef

Syntax :

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

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

Example 1.51

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

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

# define B 2

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

# undef A

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

1.9 LINE

Syntax :

***# line** constant [“filename”]*

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

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

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

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

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

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

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

Example 1.52

```
# line 124
```

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

Example 1.53

```
# line 1234 “file.c”
```

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

Example 1.54

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

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

1.10 ERROR

Syntax :

***# error** [token_sequence]*

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

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

Macros in the token sequence are not expanded.

Example 1.55

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

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

Example 1.56

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

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

1.11 MIXED LANGUAGE PROGRAMMING

Syntax :

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

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

The assembly text is not restricted to a single line.

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

Example 1.57

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

1.12 PREDEFINED MACROS

The following macros are predefined.

1. __LINE__
2. __FILE__
3. __DATE__
4. __TIME__
5. __STDC__
6. __CC665S__
7. __VERSION__
8. __ARCHITECTURE__
9. __NX_8_500__
10. __NX_8_500S__

11. __BASEPTR__
12. __NO_BASEPTR__
13. __MS__
14. __ME__
15. __MM__
16. __MC__
17. __MK__
18. __ML__
19. __MIXC__
20. __MIXM__
21. __MIXL__
22. __UNSIGNEDCHAR__

The above predefined macros cannot be redefined or undefined.

1. __LINE__

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

2. __FILE__

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

3. __DATE__

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

   “Mmm dd yyyy”

4. __TIME__

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

   “hh:mm:ss”

5. __STDC__

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

6. __CC665S__

   __CC665S__ expands to a decimal constant 1.

7. __VERSION__

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

   “Ver.X.YY”

   where X.YY is the current version number.

8. __ARCHITECTURE__

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

   “core”

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

9. __NX_8_500__

   __NX_8_500__ expands to a decimal constant 1, if the C source program is compiled for /nX500 CPU core. Otherwise this macro is not defined.

10. __NX_8_500S__

   __NX_8_500S__ expands to a decimal constant 1, if the C source program is compiled for /nX500S CPU core. Otherwise this macro is not defined.

11. __BASEPTR__

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

12. __NO_BASEPTR__

   __NO_BASEPTR__ expands to a decimal constant 1, if the C source program is compiled without /SD option. Otherwise this macro is not defined.

13. __MS__

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

14. __ME__

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

15. __MM__

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

16. __MC__

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

17. __MK__

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

18. __ML__

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

19. __MIXC__

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

20. __MIXM__

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

21. __MIXL__

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

22. __UNSIGNEDCHAR__

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

Consider the source filename as file1.c, the number of the source line being compiled as 200, the date of compilation as 23 December 1992 and the time of compilation as 10 hours : 20 minutes : 30 seconds. The predefined macros expand as follows.

Example 1.58

MACRO CALL	REPLACEMENT TEXT
__LINE__	200
__FILE__	“file1.c”
__DATE__	“Dec 23 1992”
__TIME__	“10:20:30”
__STDC__	0

Example 1.59

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

If the above C source program is compiled with small C memory model option, then it is equivalent to

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

2. LEXICAL CONVENTIONS

2.1 CHARACTER SET

This section describes the lexical conventions adopted by CC665S. After preprocessing, the source program is reduced to a series of tokens based on the lexical conventions.

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

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

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

CC665S treats upper-case and lower-case letters as distinct characters.

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

2.2 TOKENS

In a ‘C’ source program, the basic element recognized by the compiler is the character group known as a “token”. A token is source program text, the compiler will not attempt to further analyze into component elements. The tokens recognized by CC665S are :

* Identifiers
* Keywords
* Comments
* Constants
* Operators

2.2.1 Identifiers

An identifier is a sequence of letters, digits and underscores. The first character must be a letter or underscore. By default, CC665S assumes maximum identifier length as 31. If an identifier exceeding this length is specified, CC665S outputs a warning message and considers only the first 31 characters. However, CC665S provides a command line option /SL for the user to specify the maximum length of an identifier. User may specify a length ranging from 31 to 254, inclusive of both.

Following are examples of identifiers :

Example 2.1

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

2.2.2 Keywords

Identifiers which are set aside by the compiler for its use are keywords. They cannot be redeclared. They identify data types, storage class and statements in CC665S. Keywords must be expressed in lower-case letters. CC665S reserves the following words as keywords :

auto break case char const
continue default do double else
enum extern float for goto
if int long register return

short signed sizeof static struct

switch typedef union unsigned void

volatile while __accpass __asm __divbu

__divqu __divu __far __interrupt __modbu

__modqu __modu __mulbu __mulu __nfar

__noacc

2.2.3 Comments

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

Example 2.2

i. /* This is a comment */

ii. /* Comments /* nesting */ is not allowed */

The second line (ii) would result in error.

Each comment is replaced by a single space.

2.2.4 Constants

Constants in ‘C’ refer to fixed values, characters and character strings, which cannot be altered by the program. CC665S supports four types of constants - integral, floating, character and strings.

2.2.4.1 INTEGRAL CONSTANTS

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

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

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

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

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

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

The following table shows the range of values and the corresponding type for octal and hexadecimal constants in CC665S where int type is 16 bits long.

Hexadecimal range	**Octal range**	**Type**
0x0 to 0x7fff	0 to 077777	int
0x8000 to 0xffff	0100000 to 0177777	unsigned int
0x10000 to 0x7fffffff	0200000 to 017777777777	long
0x80000000 to 0xffffffff	020000000000 to 037777777777	unsigned long

The following table shows the range of values and the corresponding type for decimal constants.

Decimal range	**Type**
0 to 32767	int
32768 to 2147483647	long
2147483648 to 4294967295	unsigned long

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

Some examples of integer constants are shown below:

Example 2.3

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

2.2.4.2 FLOATING-POINT CONSTANTS

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

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

Example 2.4

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

2.2.4.3 CHARACTER CONSTANTS

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

Example 2.5

‘ ’ Single blank space

‘z’ Lower-case z

‘\n’ Newline character

‘\\’ Backslash

‘\’’ Single quote

2.2.4.4 STRING CONSTANTS

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

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

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

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

is same as

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

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

“This is first,”

“ this is second.”

will be concatenated as

“This is first, this is second.”

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

Example 2.6

i. “One\\two”

ii. “\”Do it\” Mike said.”

2.2.4.5 ESCAPE SEQUENCES

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

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of non-printable characters and characters that normally have special meanings, such as the double quotation mark character (“). The following table lists the CC665S escape sequences.

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

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

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

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

2.2.5 Operators

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

UNARY OPERATORS

! ~ - * & + ++ -- sizeof

BINARY OPERATORS

+ - * / % << >>

< <= > >= == != &

| ^ && || , = +=

-= *= /= %= >>= <<= &=

|= ^=

TERNARY OPERATOR

?:

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

3. PROGRAM STRUCTURE

3.1 SOURCE PROGRAM

This section defines the terms that are used later in the manual to describe the ‘C’ language as implemented by CC665S and discusses the structure of ‘C’ programs.

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

DIRECTIVES

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

DECLARATIONS AND DEFINITIONS

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

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

Example : 3.1

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

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

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

3.2 SOURCE FILES

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

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

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

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

3.3 FUNCTIONS AND PROGRAM EXECUTION

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

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

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

Arguments are passed between functions using call by value method.

3.4 LIFETIME AND VISIBILITY

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

3.4.1 Blocks

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

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

3.4.2 Lifetime

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

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

3.4.3 Visibility

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

3.5 NAMING CLASSES

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

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

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

Statement labels

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

Variables and Functions

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

Formal parameters

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

typedef names

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

Tags

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

Members

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

Example 3.2

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

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

3.6 DATA TYPES

CC665S supports several basic data types and derived data types.

BASIC TYPES

There are several fundamental types supported by CC665S. They include **char**, **int**, **long**, **float** and **double**. Three sizes of integers are available namely, **short int**, **int**, and **long int**. Both **signed** and **unsigned** objects of **char** and **int** types can be declared. The size as well as the smallest and largest values of each type are mentioned in section 4.2.

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

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

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

DERIVED TYPES

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

Arrays of objects of a given type.

Functions returning objects of a given type.

Pointers to objects of a given type.

Structures containing a sequence of objects of various types.

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

4. DECLARATIONS

4.1 INTRODUCTION

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

```
declaration : [ declaration_specifiers ] [init_declarator_list] ;

declaration : __asm (string)
```

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

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

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

init_declarator_list :
        init_declarator
        init_declarator_list , init_declarator

init_declarator :
        declarator
        declarator = initializer
```

4.2 TYPE SPECIFIERS

The type specifiers supported by CC665S are listed below :

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

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

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

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

The keyword **void** has three uses :

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

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

Type	Storage	Range of values
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short,int	2 bytes	-32,768 to 32,767
unsigned short,unsigned int	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE-standard notation 3.4e-38 to 3.4e+38
double	8 bytes	1.7e-308 to 1.7e+308

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

4.3 TYPE QUALIFIERS

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

Types may also be qualified, to indicate special properties of the objects being declared. The type qualifiers supported by CC665S are **const** and **volatile.**

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

CC665S allocates such variables in Code memory (ROM). The **const** type qualifier may be used only with global variables. If **/WIN** option is specified in the command line, then CC665S allocates **const** variables in the ROMWINDOW region. If these variables are modified, warning message is issued by the compiler.

CC665S ignores **const** qualifier for local automatic variables and function parameters, after issuing a warning message. However, if **/WIN** option is specified and function parameters are qualified with **const**, no warning message is issued. If the **const** qualified function parameter is modified, warning message is issued.

In case of structure and union, tags cannot be qualified by **const**. Only **struct/union** variables can be qualified by **const**. CC665S ignores **const** in case of **structure** and **union** tags. The **const** qualifier along with the struct/union tags are taken as qualifier for the variables, if any, specified along with the **struct/union** tag declaration.

Example 4.1

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

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

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

Individual members of a structure cannot be qualified by const.

Example 4.2

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

struct tag1 var2 ;
```

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

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

Example 4.3

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

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

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

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

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

Example 4.4

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

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

4.4 DECLARATORS

declarator :
[pointer] direct_declarator

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

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

type_qualifier_list :
type_qualifier
type_qualifier_list type_qualifier

memory_function_qualifier_list :
function_qualifier
memory_function_qualifier_list memory_model_qualifier
memory_model_qualifier
memory_function_qualifier_list function_qualifier

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

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

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

Example 4.5

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

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

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

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

Example 4.6

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

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

4.4.1 Memory Model Qualifiers

Memory model qualifiers can be used in a declaration, to explicitly specify the addressing type of the variable. **__far** and **__nfar** are the two keywords supported by CC665S that can be used to specify the addressing type of an object. A memory model specifier affects the token immediately to it’s right. Memory model qualifiers can qualify only objects and pointers to object.

Example 4.7

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

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

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

__far keyword can be used to qualify data, table and functions. If the object is qualified by **const** and **__far**, storage will be allocated for the object in any one of the Code Memory segments. Similarly, if a non-const object is qualified by **__far**, storage will be allocated for the object in any one of the Data Memory segments. Functions qualified by **__far** will be allocated in any one of the Code Memory segments.

__nfar keyword can be used to qualify functions only. If a function is qualified by **__nfar ,** it is allocated in the default segment. A function cannot be qualified with both **__far** and **__nfar.**

Structure and union**,** tags cannot be qualified by **__far**. Only **struct/union** variables can be qualified by **__far**. CC665S issues error if struct/union tags are qualified with **__far**.

Example 4.8

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

               } var0 ;

struct tag {
            int a ;
            char b ;

        } var0 ;

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

In the above example, CC665S issues error for **struct** ‘tag1’ declaration, as the **struct** tag is qualified with **__far**. The variable ‘var2’ is qualified with **__far** and therefore it is allocated in any one of the Data Memory segments.

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

Example 4.9

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

struct tag1 var2 ;
```

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

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

Example 4.10

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

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

4.4.2 Function Qualifiers

CC665S supports the following function qualifiers.

1. __accpass
2. __noacc
3. __interrupt

The above listed function qualifiers can qualify functions only. If they are used to qualify any other object, error is issued.

If a function is qualified with **__accpass**, it informs the compiler that the first argument is available in the Accumulator and the return value should be placed in the Accumulator. However, if the size of the first argument is greater than 2 bytes or the first argument is a structure/union, the first argument is not placed in the Accumulator. Similarly, if the size of the return value is greater than 2 bytes or if the function returns structure/union, the return value is not placed in the Accumulator. If a function is qualified with more than one **__accpass**, error is issued.

Example 4.11

```
int __accpass fn1 ( int arg1 ) ;      /*  value of arg1 is available in accumulator and the return value
                                          will be placed in the accumulator */
int __accpass fn2 ( long arg) ;       /*  the first argument value is not placed in the accumulator as
                                          the size is more than 2 bytes */
long __accpass fn3 (int arg) ;        /*  the return value is not placed in the accumulator as the size is
                                          more than 2 bytes */
int __accpass var ;                   /*  error : as var is not a function */
```

If a function is qualified with **__noacc**, it informs the compiler not to use accumulator for the first argument and the return value. If a function is qualified with more than one **__noacc** qualifier, error is issued. If a function is qualified with both **__accpass** and **__noacc**, error is issued.

Example 4.12

```
int __noacc fn1 ( int arg ) ;             /*          fn1 will not assume that value of arg is
                                          available   in the argument. */
int __noacc __accpass fn2 ( int arg ) ;   /*          error : as a function cannot be qualified with
                                             both __accpass and __noacc */
```

If **/REG** option is specified in the command line, by default all functions are assumed to be qualified with **__accpass**, unless they are qualified with **__noacc**.

If a function is qualified with **__interrupt**, it informs the compiler that the function is an interrupt routine function. If a function qualified with **__interrupt** has either return value or takes any argument, warning is issued and the **__interrupt** qualifier is ignored. If a **__interrupt** qualified function is qualified with either **__far** or **__nfar**, error is issued.

Example 4.13

```
void __interrupt  fn1 () ;            /* fn1 will be treated as interrupt function */
int __interrupt fn2 () ;              /* warning will be issued and __interrupt will be ignored */
void __interrupt fn3 ( int arg ) ;    /* warning will be issued and __interrupt will be ignored */
```

4.4.3 Interpreting Declarations

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

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

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

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

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

Example 4.14

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

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

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

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

Example 4.15

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

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

Example 4.16

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

Example 4.17

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

In example 4.17, a declaration to a function returning a pointer to a **char**, and which takes two arguments as **int** and a **char** is specified.

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

Example 4.18

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

A declaration for a function returning a pointer to far memory is given below:

Example 4.19

```
int __far * ffn () ;            /* ffn is function returning a far pointer */
```

A declaration for a pointer to far function is given below:

Example 4.20

```
int (__far * pfn) () ;          /* pfn is pointer to far function */
```

A declaration for a far function returning a pointer to far memory is given below:

Example 4.21

```
int __far * __far ffnfptr () ;          /* ffnfptr is a far function returning a far pointer */
```

4.5 VARIABLE DECLARATIONS

This section describes the form and meaning of variable declarations.

Syntax :

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

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

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

4.5.1 Simple Variable Declarations

Syntax :

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

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

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

Example 4.22

```
int x ;                         /*  declares a simple integer variable */
unsigned long int lvar1, lvar2 ;  /*   two variables unsigned long int type is declared */
const int init = -1 ;           /*  declares an int variable, qualified by const and
                                    initialized to -1 */
int __far fvar ;                /*  declares fvar as int variable that is located in far data
                                    memory */
int __far fvar, nvar ;          /*  declares fvar as int variable that is located in far data
                                    memory , but nvar is in near memory*/
```

4.5.2 Structure Declarations

Syntax :

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

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

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

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

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

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

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

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

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

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

A bit-field declaration has the following form :

type-specifier [identifier] : constant-expression;

The constant-expression specifies the number of bits in the bit-field. The type specifier may be **unsigned char** or **unsigned int**. If the type specified is **signed char**, **signed int**, **int** or **char**, CC665S issues warning message and treats them as unsigned. However, if **/J** option is specified no warning is issued for **char** specified bit fields, as it is treated as **unsigned char**. Constant-expression must be a non-negative integer value which takes values from 0 to 8 for **unsigned char** members and 0 to 16 for **unsigned int** members. Array of bit-fields, pointers to bit-fields and functions returning bit-fields are not allowed. The optional identifier names the bit-field. Named bit-fields cannot have bit-width of 0. Unnamed bit-fields can be used as dummy fields for alignment purposes. An unnamed bit-field whose width is specified as 0 guarantees that storage for the member following it in the member-declaration-list begins on an integral boundary.

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

Storage

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

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

Example 4.23

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

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

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

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

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

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

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

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

4.5.3 Union Declarations

Syntax :

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

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

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

Storage

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

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

Example 4.24

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

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

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

4.5.4 Enumeration Declarations

Syntax :

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

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

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

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

* In the first form, enum-list specifies the values and names of the enumeration set. (The enum-list is described in detail below.) The optional tag is an identifier that names the enumeration type defined by enum-list. The declarator names the enumeration variable. Zero or more enumeration variables may be specified in a single enumeration declaration.
* The second form of the enumeration declaration uses a previously defined enumeration tag to refer to an enumeration type defined elsewhere. The tag must refer to a defined enumeration type, and that enumeration type must be currently visible. Since the enumeration type is defined elsewhere, enum-list does not appear in this type of declaration. Declarations of pointers to enumerations and **typedef** declarations for enumeration types can use the enumeration tag before the enumeration type is defined. However, the enumeration definition must be encountered prior to any actual use of the **typedef** declaration or pointer.

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

An enum-list has the following form:

identifier [= constant-expression]

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

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

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

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

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

Example 4.25

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

} levels ;
```

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

Example 4.26

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

const enum constants speed = high ;
```

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

4.5.5 Array Declarations

Syntax :

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

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

Array declarations have two forms as shown in the syntax.

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

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

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

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

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

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

Storage

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

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

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

Limitations

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

Example 4.27

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

struct tag
{
            int ivar ;
            long lvar ;

} array_of_structures [10] ;
                               /* Declares an array of structures having 10 elements. */

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

4.5.6 Pointer Declarations

Syntax :

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

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

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

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

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

Example 4.28

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

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

Example 4.29

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

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

A pointer can also be qualified as a far pointer by specifying **__far** keyword immediately to it's left. A far pointer contains a far address of an object. Each level of indirection in a pointer declaration must be qualified as **__far** if that indirection points to a far memory location.

Example 4.30

```
int * __far fptr1 ;          /*     fptr1 is a location in far segment, whose content points to an
                                object in default segment */

int __far * fptr2 ;          /*     fptr2 is a location in default segment, whose content points to
                                an object in far segment */

int __far * __far fptr3 ;    /*     fptr3 is a location in far segment , whose contents also point
                                to a location in far segment */
```

Storage

The amount of storage required for an address depends on the memory model selected. If the pointer points to near or effective near memory, the size of the pointer is 2 bytes. If the pointer points to far, nfar, xnear, effective xnear memory or large memory, the size of the pointer is 4 bytes.

Example 4.31

```
char *string ;                  /*  a pointer to character named string */
long *arr_of_pntrs [10] ;       /*  array of pointers to long */
void (*pf)(int) ;               /*  pointer to a function returning no values. The function
                                    takes an integer argument */
 struct inv_type *left, *right ; /*  declares two pointers to a structure of inv_type */
char **p ;                      /*  declares a pointer to pointer of characters */
```

4.6 FUNCTION DECLARATIONS AND PROTOTYPES

Syntax :

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

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

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

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

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

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

4.6.1 Formal Parameters

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

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

Example 4.32

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

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

Example 4.33

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

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

4.6.2 Return Type

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

Example 4.34

```
struct tag
{

        int a ;
        long b ;
} input_structures[10] ;

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

4.6.3 List Of Formal Parameters

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

Syntax :

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

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

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

Example 4.35

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

4.6.4 Memory Model Qualifiers For Functions

A function can be qualified as **__far** or **__nfar**. Functions qualified with **__far** may not be placed in default segment. These functions are called through large addressing. Functions qualified with **__nfar** are placed in default segment. They too are called through large addressing, but their segment address is always 0. A function qualified with **__far** cannot call a near function. However, it can call a function that is qualified with either **__far** or **__nfar.** Functions that are not qualified with **__far** can call near, **__nfar** and **__far** functions. Therefore, if a function that is qualified with **__far** has to call a near function, then it has to call a nfar qualified function, which in turn calls a near function.

Example : 4.36

```
int __far ffn () ;
int nfn () ;
int __nfar nffn () ;
int __far ffn ()
{
      nfn () ;                /* error : a far function cannot call near functions */
      nffn () ;
}
int __nfar nffn ()
{
      nfn () ;                /* nfar functions can call near functions */
      nffn () ;
}
```

4.6.5 Function Qualifiers For Functions

CC665S supports **__accpass**, **__noacc** and **__interrupt** keywords that can qualify functions only. When a function is qualified with **__accpass,** the first argument and the return value are placed in the Accumulator. However, if **/REG** option is specified all functions are assumed to be qualified with **__accpass**, except those that are qualified with **__noacc**. If a function is qualified with both **__accpass** and **__noacc**, error is issued.

__interrupt keyword can be used to qualify a function as an interrupt function. These functions cannot take or return values.

4.7 STORAGE CLASS SPECIFIERS

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

All functions have global lifetimes.

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

CC665S provides the following 5 storage class specifiers.

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

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

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

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

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

Example 4.37

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

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

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

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

∗ Whether the declaration appears at the external or internal level

∗ Whether the item being declared is variable or function.

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

4.7.1 Variable Declarations At The External Level

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

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

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

A definition at the external level can take several forms :

∗ A variable declared with the **static** storage class specifier is a definition of that variable. Both **const** and non-const **static** variable can be initialized with a constant-expression. For example **static int** x; and **const static int** y = 10; are considered definitions of variables ‘x’ and ‘y’.

∗ A variable that is explicitly initialized at the external level are definitions of that variable. CC665S allows initialization of both **const** and non-const specified variables at the external level. For example, **const int** i = 10 and **int** y = 20 are the definitions of the variable ‘i’. and ‘y’ respectively.

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

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

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

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

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

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

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

Example 4.38

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

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

int global_variable ;              /* definition of global_variable */

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

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

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

4.7.2 Variable Declarations At The Internal Level

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

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

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

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

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

Example 4.39

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

        file2 () ;
}

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

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

4.7.3 Function Declarations At The Internal And External Levels

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

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

* A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call **static** functions, but functions in other source files cannot. Another **static** function with the same name in a different source file can be used without conflict.

* Functions declared as **extern** are visible throughout all the source files that make up the program, unless it is later redeclared as static. Any function can call an **extern** function.

* Function declarations that omit the storage class specifier are **extern** by default.

4.8 INITIALIZATION

Syntax :

= *initializer*

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

The following rules apply for initialization :

* Both const and non-const qualified variables declared at the external level can be initialized. If **const** qualified variables are not initialized at external level, they are assigned value 0.

* Variables declared with auto storage class specifier are initialized each time control passes to the block in which they are declared. If an initializer is omitted from the declaration of an auto variable, the initial value of the variable is undefined. Both aggregate (array, structure and unions) and non aggregate variables can be initialized.

- * The initial values for external variable declaration and for all static variables, whether external or internal, must be constant expressions. Either constant or variable values can be used to initialize auto variables.
- * The const qualifier also causes an item to be placed in Code Memory (ROM). Strings and values used for initialization are placed in Code Memory.

Example 4.40

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

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

4.8.1 Fundamental And Pointer Types

Syntax :

= expression

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

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

Example 4.41

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

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

4.8.2 Aggregate Types

Syntax :

= {initializer-list}

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

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

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

Example 4.42

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

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

Example 4.43

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

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

Example 4.44

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

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

The initialization

Example 4.45

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

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

4.8.3 String Initializers

Syntax :

= "characters"

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

Example 4.46

```
char str_arr [] = "abc" ;
```

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

Example 4.47

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

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

4.9 TYPE DECLARATION

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

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

4.9.1 Structure And Union Types

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

∗ In structure and union type declarations variable is omitted.

∗ In structure and union type declaration tag is required; it names the structure or union type.

∗ The member declaration list defining the type must appear in the structure and union type declaration.

Example 4.48

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

The above example declares a structure type named tag.

4.9.2 Typedef Declarations

Syntax :

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

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

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

Example 4.49

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

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

COMPLEX *sp ;
```

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

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

4.10 TYPE NAMES

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

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

Formal parameter lists are discussed in section 4.6.1.

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

type-specifier abstract-declarator

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

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

Example 4.50

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

4.11 FUNCTIONS

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

4.11.1 Function Definitions

Syntax :

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

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

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

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

4.11.1.1 STORAGE CLASS

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

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

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

4.11.1.2 RETURN TYPE AND FUNCTION NAME

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

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

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

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

4.11.1.3 FORMAL PARAMETERS

Syntax :

form1:

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

form2:

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

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

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

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

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

Example :4.51

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

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

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

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

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

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

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

4.11.1.4 FUNCTION BODY

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

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

4.11.2 Function Prototypes

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

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

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

Function prototypes have the following important uses :

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

4.11.3 Function Calls

Syntax :

expression([expression-list])

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

When the function is executed :

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

4.11.3.1 ACTUAL ARGUMENTS

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

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

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

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

4.11.3.2 RECURSIVE CALLS

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

4.12 ASM DECLARATION

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

__asm (string)

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

5. EXPRESSIONS AND OPERATORS

5.1 OPERATORS

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

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

∗ Unary operators, which take single operand.

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

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

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

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

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

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

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

Precedence and Associativity of ‘C’ Operators :

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

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

5.2 LVALUES AND RVALUES

A variable identifier is one of the ‘C’ primary expressions. This type of expression yields a single value, the object of the variable. However, when using the variable identifier with other operators, the expression evaluates to the location of the variable in memory. The address of the variable is the lvalue. The object stored at the address is the rvalue. CC665S uses rvalue and lvalue of variables in evaluation of an expression given below :

```
x = y ;
```

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

The following ‘C’ expressions may be lvalue expressions :

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

The above is expressed as the following syntax for lvalue :

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

All lvalue expressions represent a single location in memory.

5.3 CONVERSIONS

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

5.3.1 Integral Promotion

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

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

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

5.3.2 Arithmetic Conversions

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

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

5.3.3 Pointer Conversions

When two pointers are operated upon, they are converted to same size. Pointer size depends upon the memory model and the memory model qualifier specified for the pointer. When a pointer is qualified with **__far**, the size of the pointer is 4 bytes, as in case of pointers in large memory model. In small memory model, the default pointer size is 2 bytes.

Expressions may contain, both near pointers (2 bytes) and far pointers (4 bytes). When two pointers of different size are operated upon, they are promoted to same size. The near pointer is converted to far pointer, with default segment address in the upper two bytes.

5.4 PRIMARY EXPRESSIONS AND OPERATORS

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

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

5.4.1 Identifiers

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

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

5.4.2 Constants

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

5.4.3 Strings

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

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

Strings cannot be specified as far strings. The segment address of strings are fixed for a specified memory model. Therefore, no segment register switching is done under any memory model option.

5.4.4 Parenthesized Expression

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

Example 5.1

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

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

5.5. ARRAY REFERENCES

array_reference :
    expression1 [expression2]

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

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

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

5.6 FUNCTION CALLS

```
function calls :
        expression ( )
        expression ( argument_list )
```

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

```
argument_list :
        expression
        argument_list, expression
```

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

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

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

The arguments undergo integral promotion before being sent.

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

If no prototype is specified for a function and if its body is not defined, the above mentioned checks are not performed. If prototype is not specified, CC665S assumes the prototype from the body definition, if specified. If the prototype and the body definition differs, body definition overwrites the prototype.

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

When a far pointer is passed as an argument to a function which actually takes a near pointer, the segment information of the argument pointer is lost. CC665S issues error when a far pointer is passed as argument in place of near pointer. However, if a near pointer is passed as argument to a function which actually takes a far pointer, a warning message is issued. The near pointer is converted to far pointer with the default segment address in the upper two bytes of the converted pointer.

The following built-in functions are supported.

__mulu __mulbu __divu __divqu
__divbu __modu __modqu __modbu

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

unsigned long __mulu(unsigned int, unsigned int) ;
unsigned int __mulbu(unsigned char, unsigned char) ;
unsigned long __divu(unsigned long, unsigned int) ;
unsigned int __divqu(unsigned long, unsigned int) ;
unsigned int __divbu(unsigned int, unsigned char) ;
unsigned int __modu(unsigned long, unsigned int) ;
unsigned int __modqu(unsigned long, unsigned int) ;
unsigned char __modbu(unsigned int, unsigned char) ;

These built-in functions may be called as any other function is called. Argument conversions are performed similar to other functions.

5.7 STRUCTURE AND UNION REFERENCES

structure_reference :
expression . identifier
expression -> identifier

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

Dot operator

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

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

Arrow operator

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

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

Example 5.2

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

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

In the above example:

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

5.8 POST INCREMENT

post_increment :
expression ++

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

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

Example 5.3

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

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

5.9 POST DECREMENT

post_decrement :
expression --

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

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

Example 5.4

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

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

5.10 PRE INCREMENT

pre_increment:
++ expression

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

Example 5.5

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

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

5.11 PRE DECREMENT

pre_decrement:
-- expression

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

Example 5.6

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

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

5.12 ADDRESS OPERATOR

address_operator :
    & expression

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

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

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

Example 5.7

```
int x, * p ;

p = &x ;
```

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

When address is taken for a variable that resides in far segment, the address size is 4 bytes. The address contains the offset value in the lower two bytes, and segment address in the upper two bytes.

Example 5.8

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

p = &x ;              /* size of the address of x is 4 bytes */
```

5.13 INDIRECTION OPERATOR

indirection_operator :
    ** expression*

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

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

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

Example 5.9

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

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

5.14 UNARY PLUS OPERATOR

```
unary_plus_operator :
        + expression
```

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

5.15 UNARY MINUS OPERATOR

```
unary_minus_operator :
        - expression
```

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

Example 5.10

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

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

5.16 ONE’S COMPLEMENT OPERATOR

bit_not_operator :
~ expression

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

Example 5.11

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

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

5.17 LOGICAL NOT OPERATOR

logical_not_operator :
! expression

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

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

Example 5.12

```
int x, y ;

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

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

5.18 SIZEOF OPERATOR

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

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

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

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

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

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

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

Example 5.13

```
long array [10] ;

z = sizeof ( array ) ;
```

The value of z is 40.

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

Example 5.14

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

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

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

Example 5.15

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

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

In the above example, the size of the pointer variable ‘dptr’ depends on the C memory model in which it is compiled. If the program is compiled in small ‘C’ memory model, then the value of ‘i’ is 2. If the program is compiled in large ‘C’ memory model, then the value of ‘i’ is 4.

5.19 CAST OPERATOR

cast_operator :
(typename) expression

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

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

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

An object in data memory may not be casted to a type resulting in code memory and vice versa. In such cases, CC665S issues an error. However, if /WIN option is specified in the command line, warning will be issued.

Example 5.16

```
(int (*)[]) p1 ;
```

In the above example, p1 is cast as a pointer to array of int.

A near variable cannot be casted to a far variable. However, a pointer to near memory can be casted to a pointer to a far memory, provided the memory model specified in the command line supports far qualifier for that data type. When a pointer to far memory is casted to a pointer to near memory, CC665S issues error. However, if a pointer to near memory is casted to a pointer to far memory, pointer conversion is performed.

Example 5.17

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

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

5.20 MULTIPLICATIVE OPERATORS

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

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

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

The binary * operator denotes multiplication.

The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second operand. If the second operand is zero, the result is undefined. If CC665S detects the second operand as zero, warning message is displayed.

Example 5.18

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

5.21 ADDITIVE OPERATORS

additive_expression :
expression + expression
expression - expression

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

The result of + operator is the sum of the operands. A pointer to an object and a value of any integral type may be added. CC665S calculates the size of one object, multiplies this by the integer thus obtaining the offset value, and then adds the offset value to the address of the designated element. The result is a pointer of the same type as the original pointer, and points to another object, appropriately offset from the original object. Thus if P is a pointer to an object, the expression P + 1 is a pointer to the next object.

CC665S issues an error if two pointers are added.

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

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

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

Example 5.19

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

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

5.22 SHIFT OPERATORS

shift_expression :
    expression << expression
    expression >> expression

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

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

The result of e1 << e2 is the value of the expression e1 shifted to the left by e2 bits. CC665S clears vacated bits.

The result of e1 >> e2 is the value of the expression e1 shifted to the right by e2 bits. CC665S clears vacated bits if the left operand e1 is unsigned; otherwise, vacated bits are filled with a copy of e1's sign bit.

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

Example 5.20

```
unsigned int x, y, z ;

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

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

5.23 RELATIONAL OPERATORS

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

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

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

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

A pointer may be compared to a constant integral expression with value 0, or to a pointer to void.

CC665S displays error when pointers to different memory (one pointing to code memory and the other to data memory) are compared. However, if /WIN option is specified only warning message will be issued. If both operands are pointers, pointer conversion is performed.

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

Example 5.21

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

z = x > y ;
```

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

Example 5.22

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

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

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

5.24 EQUALITY OPERATORS

equality_expression :
expression == expression
expression != expression

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

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

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

Example 5.23

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

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

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

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

Example 5.24

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

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

     s [j] = '\0' ;
}
```

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

5.25 BITWISE AND OPERATOR

bit_and_expression :
expression & expression

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

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

5.26 BITWISE EXCLUSIVE OR OPERATOR

```
bit_xor_expression :
        expression ^ expression
```

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

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

5.27 BITWISE OR OPERATOR

```
bit_or_expression :
        expression | expression
```

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

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

5.28 LOGICAL AND OPERATOR

```
logical_AND_expression :
        expression && expression
```

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

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

If CC665S is able to make an evaluation by examining only the left operand, it does not evaluate the right operand.

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

5.29 LOGICAL OR OPERATOR

```
logical_OR_expression :
        expression || expression
```

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

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

If CC665S is able to make an evaluation by examining only the left operand, it does not evaluate the right operand.

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

Example 5.25

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

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

5.30 CONDITIONAL EXPRESSION AND OPERATORS

```
conditional_expression :
        expression1 ? expression2 : expression3
```

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

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

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

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

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

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

- If expression2 or expression3 has integral or floating type, the operator performs usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both expression2 and expression3 have the same structure, union or pointer type, the type of the result is the same structure, union or pointer type.
- If both operands have void type, the result has type void.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to void, the pointer to the object is converted to a pointer to void and the result is pointer to void.
- If either of expression2 or expression3 is a near pointer and the other is a far pointer, pointer conversion is performed.
- If either expression2 or expression3 is a pointer and the other operand is a constant expression with the value 0, the type of the result is pointer type.

Example 5.26

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

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

5.31 ASSIGNMENT EXPRESSIONS AND OPERATORS

assignment_expression :
expression assign_op expression

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

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

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

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

- Both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment.
- One operand is a pointer and the other is a pointer to void.
- The left operand is a pointer and the right operand is constant expression with value 0.
- Both operands are pointers to functions or objects whose types are the same except for the possible absence of const or volatile in the right operand.
- If a pointer to far memory is assigned to a pointer to near memory, the segment information is lost. Further operations using the pointer may result in undefined behavior. CC665S issues error message if a far pointer is assigned to a near pointer. However, if a near pointer is assigned to a far pointer there is no loss of segment information. Default segment address will be assigned to the upper two bytes of the near pointer. CC665S issues warning message, when a near pointer is assigned to a far pointer.

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

Example 5.27

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

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

Example 5.28

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

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

5.32 COMMA EXPRESSION AND OPERATOR

comma_expression :
    expression, expression

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

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

Example 5.29

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

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

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

5.33 CONSTANT EXPRESSIONS

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

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

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

6. STATEMENTS

6.1 INTRODUCTION

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

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

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

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

6.2 LABELED STATEMENT

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

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

The scope of an identifier is the current function. Labels cannot be redeclared within the same function. Label names do not collide with identifiers with same name in other declarations (local as well as global). Because CC665S uses a separate name space for labels.

A label, consisting of the keyword **case** followed by a constant expression, is a case label. A label consisting of the keyword **default** is called a default label. Case labels and default labels are used within the **switch** statements. If used elsewhere, error is displayed by CC665S. The constant expression of the case label must be of integral type. Case labels and default labels are explained in detail in the section for **switch** statement.

Labels in themselves do not alter the flow of control.

6.3 EXPRESSION STATEMENT

```
expression_statement :
        expression ;
        ;
```

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

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

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

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

The following are examples of expression statements :

Example 6.1

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

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

Example 6.2

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

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

6.4 COMPOUND STATEMENT

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

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

The compound statement contains optional declarations followed by a list of statements which is also optional, all enclosed in braces. If declarations are included, the variables declared are local to the block, and, for the rest of the block, they supersede any declarations of the variables of the same name. The outer declaration becomes valid at the end of the block.

Initializations of automatic objects included in the block are performed each time the block is entered in the order of the declarators. Initializations of static objects inside the block are performed only once.

Example 6.3

```
int y, z ;

fn ()
{
      int x = 10 ;

      z = 1 ;

      if (x > y)
            x++ ;
      else
            y++ ;
}
```

6.5 SELECTION STATEMENTS

Selection statements test the specified condition and depending on the result, one of several flows of control is chosen. There are two selection statements.

1. if statement
2. switch statement

6.5.1 if Statement

```
selection_statement :
            if (expression) statement
            if (expression) statement1 else statement2
```

'if' statements may or may not have the '**else'** part. The '**if**' statement must have an expression in parentheses following the keyword '**if**'. Expression must be of arithmetic or pointer type. The expression is evaluated with all side effects.

If result of the expression is non-zero, then statement1 is executed. If the expression is zero, statement1 is not executed and statement2 is executed, if present.

When '**if**' statements are nested within '**else**' clauses, an '**else**' clause matches the most recent '**if**' statement that does not have an '**else**' clause.

Example 6.4

```
int i, j, x ;

if (i < j)
        function (i) ;
else
{
        i = x ++ ;
        function (i) ;
}
```

6.5.2 switch Statement

selection_statement :
switch (expression) statement

The '**switch'** statement transfers control to a statement within its body. Control passes to the statement whose **case** constant-expression matches the value of the **switch** expression. The **switch** statement may include any number of **case** instances. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **'break'** statement.

The **'default'** statement is executed if no **case** constant expression is equal to the value of **switch** expression. If the **'default'** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed. The **'default'** statement need not come at the end; it can appear anywhere in the body of the **'switch'** statement.

The type of the **switch** expression is integral. Each **case** constant expression is converted using the usual arithmetic conversions (explained in section 5.3.2). The value of each **case** constant expression must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. All statements, between the statement where execution starts and the end of the body, are executed regardless of their labels unless a statement transfers control out of the body entirely.

The following example illustrates the use of switch to display three different LED display items :

Example 6.5

```
display_fn ()
{
/* This program displays three types of displays based on the input */

int display_item ;

        while ( ( display_item = get_display_item () ) )
        {
              switch ( display_item )
              {
                      case DATE : display_date () ;
                      break ;

                      case DAY : display_day () ;
                      break ;
                      case TIME : display_time () ;
                      break ;
              }
        }
}
```

Declarations Within A Switch

Declarations may appear at the head of the compound statement forming the switch body. But initializations included in these declarations are not performed. The switch statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

Example 6.6

```
int y ;
switch (character)
{
        int x = 1 ;     /* Improper initialization */

        case ‘a’ :
                      {
                            int x = 10 ;        /* Proper initialization */
                            y = x ;
                            break ;
                      }

        case ‘b’ :
        .
        .
}
```

6.6 ITERATION STATEMENTS

Statements in the following subsections execute repeatedly (loop), until an expression evaluates to false.

6.6.1 for Statement

```
iteration_statement :
        for ([expression1] ; [expression2] ; [expression3]) statement
```

The **'for'** statement evaluates three expressions and executes a statement (loop body) until expression2 evaluates to false. The '**for**' statement is particularly useful for executing a loop body a specified number of times.

The '**for**' statement executes the loop body zero or more times. It uses three optional control expressions as shown. A '**for**' statement executes the following steps :

1. The optional expression1 is evaluated only once before the iteration of the loop. It usually specifies the initial values for variables.
2. The optional expression2 is evaluated before each iteration. If the expression evaluates to false, execution of the '**for**' loop body terminates. If the expression is evaluated to true, the body of the loop is executed.
3. The optional expression3 is evaluated after each iteration. It usually specifies step value for variables initialized by expression1.
4. Iterations of the **'for'** statement continue until expression2 produces a false value, or until some statement such as **break** or **goto** or **return** interrupts.

Example 6.7

```
int i ;
char string1 [20], string2 [20] ;

for (i = 0; i < 15; i++)
        string1 [i] = string2 [i] ;
```

The above example copies the first 15 characters of string2 to string1.

The following 'for' statement illustrates an infinite loop :

Example 6.8

```
int i, j ;

for ( ;; )
{
        j = i + 10 ;
}
```

Infinite loops can be terminated with a **goto**, **break** or **return** statement.

6.6.2 while Statement

iteration_statement :
while (expression) statement

The '**while**' statement evaluates an expression and executes a statement (loop body) zero or more times, until the expression evaluates to false.

If the expression in parentheses evaluates to false at the first time, the loop body never executes.

Example 6.9

```
int x, array [15] ;

fn ()
{
    x = 0 ;

    while (x < 10)
    {
       array [x] = x ;
       x++ ;
    }
}
```

The above example assigns the values 0 to 9 to the first ten elements of array.

6.6.3 do Statement

iteration_statement :
do statement while (expression) ;

The '**do**' statement executes a statement (the loop body) one or more times until the expression in the while clause evaluates to false.

The statement is executed at least once, and the expression is evaluated after each subsequent execution of the loop body. If the expression is true the statement is executed again.

Example 6.10

```
int num ;

do
{
      num = get_number () ;
} while (num <= 100) ;
```

The above example gets a number until it is greater than 100.

6.7 JUMP STATEMENTS

Jump statements transfer control unconditionally. The following statements are classified as jump statements.

1. goto statement
2. break statement
3. continue statement
4. return statement

Statements other than the '**goto**' statement may be used to interrupt the execution of another statement. These statements are primarily used to interrupt 'switch' statements and loops.

6.7.1 goto Statement

```
jump_statement :
        goto identifier ;
```

The '**goto**' statement transfers control automatically to a labeled statement, where the label identifier must be located in the scope of the function containing the goto statement.

Like other 'C' statements, any of the statements in a compound statement can carry a label. A **goto** statement can transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declaration that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initialization. The results are undefined.

The following example illustrates both the '**goto**' statement and the labeled statement :

Example 6.11

```
int error_no ;

fn ()
{
    extern int error ;
    if (error )
        goto error_process ;

    .
    error_process :
    return (error_no) ;
}
```

In the above example, ‘**goto**’ statement transfers control to the point labeled error_process.

6.7.2 break Statement

jump_statement :
    break ;

The **break** statement terminates the immediately enclosing **while**, **do**, **for** or **switch** statement. Control passes to the statement following the loop body.

Example 6.12

```
int x ;

while (1)
{
   x = fn () ;

   if (x == 1)
      break ;
}
```

In this example the while loop is executed until the function returns a value 1.

6.7.3 continue Statement

jump_statement :
    continue ;

The **continue** statement passes control to the end of the immediately enclosing **while**, **do** or **for** statement. The control passes to the next iteration of the **while**, **do** or **for** statement in which it appears, bypassing any remaining statements in the loop body.

Example 6.13

```
even_fn ()
{
    int x ;

    for (x = 0; x <= 100; x++)
    {
        if (x%2)
          continue ;

        print (x) ;
    }
}
```

The above function prints all the even numbers between 0 and 100.

6.7.4 return Statement

jump_statement :
    return [expression] ;

The return statement causes a return from a function with or without a return value.

CC665S evaluates the expression, if one is specified, and returns the value to the calling function. If necessary, compiler converts the value to the declared type of the function. If there is no specified return value, the value is undefined.

Example 6.14

```
max (int a, int b)
{
   if (a > b)
      return (a) ;
   else
      return (b) ;
}
```

The above function returns the larger of its two integer arguments.

When a function which is declared to return nothing (void) returns a value, compiler issues a error message.

6.8 ASM STATEMENTS

asm_statement :
    __asm (“string”)

The asm_statement can be used to output the string contents in the assembly output file directly. The string is not processed by the compiler.

Example 6.15

INPUT:

```
__asm (“; Test for __asm”)
int gvar ;

fn (int arg)
{
    gvar = arg ;

    __asm (“\tinc          dir _gvar\n”) ;
}
```

OUTPUT:

```
;Test for __asm

        rseg $$NCODt

CFUNCTION 0
_fn     :

;;{
CLINE 7
        pushs   usp
        mov     usp,    ssp

;;      gvar = arg ;
CLINE 8
        mov     dir _gvar,      6[usp]

;;      __asm ("\tinc   dir _gvar") ;
CLINE 9
        inc     dir _gvar

;;}
CLINE 10
        pops    usp
        rt
```

7. VARIATIONS FROM ANSI STANDARD

The implementation of C language in CC665S differs from the standard ANSI X3. 159-1989 and ISO/IEC 9899 proposed by ANSI (American National Standards Institute) due to the following features :

1. Supports specification of INTERRUPT functions.
2. Qualifying a variable by ‘const’ causes the variable to be allocated in code memory (Read Only Memory).
3. Members of a structure or union cannot be qualified by ‘**const**’.
4. **Char** bit fields are allowed in structures and unions
5. Arguments cannot be qualified by ‘**const**’. However, if **/WIN** option is specified, arguments can be specified as **const.**
6. A declaration without a type specifier, a type qualifier or a storage class specifier is considered as declaration of type ‘**int**’.
7. A pointer can be compared to a constant integral expression with value 0, or to a pointer to void using relational operators also.
8. **__far** and **__nfar** memory model qualifiers are supported.
9. **__accpass**, **__noacc** and **__interrupt** function qualifiers are supported.
10. **__asm** keyword is supported.
11. The following built in functions are supported

    __mulu **__mulbu** **__divu** **__divqu**
    __divbu **__modu** **__modqu** **__modbu**