



SOFTWARE DEVELOPMENT SYSTEM

W65C02S C COMPILER/OPTIMIZER



THIS PAGE LEFT INTENTIONALLY BLANK



TABLE OF CONTENTS

CHAPTER 1 Introduction 9

- Compiler Operation 9**
- Input File 9**
- Output Files 9**
 - Creating an Object File 9
 - Creating an Assembly Language File 10
- Searching for #Include Files 10**
 - Search Order #include 11
- Compiler Options 11**
 - Compiler Option Philosophy 11
 - CCOPT6502 Environment Variable 11
- C Programs in ROM 12**
 - C Program Organization 12
 - System Organization 12
 - Creating A ROM Program 12

CHAPTER 2 WDC02CC 15

- Configuring the Program 15**
 - User Registers 15
 - Stack Registers 16
 - Compiler Registers 16
 - Work Area 17
 - Floating Point Registers 17
- Running the Program 17**
- Option Summary 18**
- Option Descriptions 19**

CHAPTER 3 Technical Notes 27

- Pseudo-Registers 27**
- Pseudo-Stack Frame 27**
- Function Calls and Argument Passing 28**
 - Function Arguments 28
 - Function Return Value 29
- Startup Code 29**
 - Identifier Name Prefixes 29
- Memory Management 29**
- CAVEATS 30**
 - Floating Point Considerations 30
 - Section Pragma 30
- Consts and Strings 31**
- In-Line Assembly Code 31**
- ASM Keyword 32**
- Other Preprocessor Features 32**
 - Predefined Names 32
 - #line 32
- Producing Optimum Code 33**
- Optimizer 33**
- Floating Point 33**
- Referencing I/O 33**
- Prototyping Functions 35**
- Variable Name Length 35**



Debugging..... 35

Assembling Compiler Output Considerations 35

Volatile Qualifiers..... 36

Path size Limitation..... 36

Global Function and Variable Definition 36

Factorization of Source Code..... 36

CHAPTER 4 Libraries..... **37**

Library Names 37

ANSI Functions..... 37

Heap Functions 37

APPENDIX A WDC Supported C Functions **39**

APPENDIX B Description of Compiler Error Messages..... **45**

 1: bad digit in octal constant 45

 2: obsolete..... 45

 3: unterminated string 45

 4: argument type mismatch..... 45

 5: invalid type for function 45

 6: inappropriate arguments 45

 7: bad declaration syntax 45

 8: syntax error in typecast..... 46

 9: invalid operand of & (address of)..... 46

 10: array size must be positive integer..... 46

 11: obsolete..... 46

 12: invalid pointer reference 47

 13: obsolete..... 47

 14: obsolete..... 47

 15: storage class conflict..... 47

 16: data type conflict..... 47

 17: internal..... 47

 18: data type conflict..... 48

 19: bad syntax..... 48

 20: structure redeclaration 48

 21: missing }..... 48

 22: syntax error in structure declaration 48

 23: syntax error in enum declaration 49

 24: need right parenthesis or comma in arg list 49

 25: structure member name expected here..... 49

 26: must be structure/union member..... 49

 27: invalid typecast 50

 28: incompatible structures 50

 29: invalid use of structure..... 50

 30: missing : in ? conditional expression 50

 31: call of non-function..... 50

 32: invalid pointer calculation 50

 33: invalid type 51

 34: undefined symbol 51

 35: typedef not allowed here..... 51

 36: obsolete..... 51

 37: invalid or missing expression 51

 38: obsolete..... 51

 39: enum redeclaration..... 52

 40: internal error 52

 41: initializer not a constant..... 52



42: too many initializers 52
43: initialization of undefined structure 53
44: missing right paren in declaration 53
45: bad declaration syntax 53
46: missing closing brace 53
47: open failure on include file 54
48: invalid symbol name 54
49: multiply defined symbol 54
50: missing bracket 54
51: lvalue required 54
52: obsolete 55
53: multiply defined label 55
54: obsolete 55
55: missing quote 55
56: missing apostrophe 55
57: obsolete 55
58: invalid # encountered 55
59: macro too long 55
60: loss of const/volatile info 55
61: reference to undefined structures 56
62: function body must be compound statement 56
63: undefined label 56
64: inappropriate arguments 56
65: invalid function argument 56
66: expected comma 57
67: invalid else 57
68: bad statement syntax 57
69: missing semicolon 57
70: goto needs a label 58
71: statement syntax error in do-while 58
72: statement syntax error in for 58
73: statement syntax error in for body 58
74: expression must be integer constant 58
75: missing colon on case 58
76: obsolete 58
77: case outside of switch 58
78: missing colon on default 58
79: duplicate default 59
80: default outside of switch 59
81: break/continue error 59
82: obsolete 59
83: too many nested includes 59
84: constant expression expected 59
85: not an argument 59
86: null dimension in array 59
87: invalid character constant 59
88: not a structure 60
89: invalid use of register storage class 60
90: symbol redeclared 60
91: invalid use of floating point type 60
92: invalid type conversion 60
93: invalid expression type for switch 61
94: invalid identifier in macro definition 61
95: obsolete 61



96: missing argument to macro.....	61
97: too many arguments in macro definition	61
98: not enough args in macro reference	61
99: internal error	61
100: internal error	61
101: missing close parenthesis on macro reference	61
102: macro arguments too long.....	61
103: #else with no #if.....	61
104: #endif with no #if.....	62
105: #endasm with no #asm.....	62
106: #asm within #asm block	62
107: missing #endif.....	62
108: missing #endasm.....	62
109: obsolete.....	63
110: invalid use of : operator	63
111: invalid use of a void expression.....	63
112: obsolete.....	63
113: duplicate case in switch	63
114: macro redefined	63
115: keyword redefined	64
116: field width must be > 0	64
117: invalid 0 length field.....	64
118: field is too wide	64
119: field not allowed here	64
120: invalid type for field	65
121: ptr/int conversion.....	65
122: ptr & int not same size	65
123: far/huge ptr & ptr not same size.....	65
124: invalid ptr/ptr expression	65
125: too many subscripts or indirection on integer.....	65
126: too many arguments.....	66
127: too few arguments.....	66
128: #error	66
129: #elif with no #if	66
130: obsolete.....	66
131: ## at the beginning/end of macro body.....	66
132: obsolete.....	66
133: # not followed by a parameter	66
134: obsolete.....	66
135: attempt to undefined a predefined macro.....	66
136: invalid #include directive.....	66
137: obsolete.....	66
138: missing right paren	66
139: missing identifier	67
140: obsolete.....	67
141: obsolete.....	67
142: range-modifier ignored	67
143: range-modifier syntax error	67
144: invalid operand for sizeof	67
145: function called without prototype	67
146: constant value too large	67
147: invalid hexadecimal constant.....	67
148: invalid floating constant	67
149: invalid character on control line	67



150: unterminated comment	68
151: no block level extern initialization.....	68
152: missing identifier in parameter list	68
153: missing static function definition.....	68
154: function definition can't be via typedef	68
155: file must contain external definition	68
156: wide string literal not allowed here.....	68
157: incompatible function declarations	68
158: called function may not return incomplete type.....	69
159: syntax error in #pragma	69
160: auto variable not used in function.....	69
161: function defined without prototype.....	69
162: can't take address of register class.....	69
163: upper bits of hex character constant ignored	69
164: non-void type function must have return value	70
165: item not previously declared found in prototype	70
166: enum must be declared outside prototype.....	70
167: can't take address of stack in this memory model	70
168: missing semicolon in asm block	70
169: can't convert far pointer to near.....	70
170: can't use TSB/TRB on volatile values.....	71
INDEX.....	73



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 1 Introduction

The WDC 65C02 C Development system provides the tools needed to do effective C and assembly language development for the 65C02 family of microprocessors. The compiler is built on top of the WDC W65CXX assembly language development system. The assembly language development system consists of a full macro assembler, an object file linker and an object file librarian.

Compiler Operation

The WDC C compiler is a full ANSI standard implementation. There are also a number of useful extensions to the ANSI standard that are controlled by compiler switches. The compiler reads the input file function by function and produces a parse tree for the entire function. A few optimizations are performed on the tree before generating code. The code generator then reads the parse tree and generates a list of assembly language instructions that are written to a temporary file. If the optimizer has been invoked, it reads the assembly language file, improves the code where possible and writes it back out. The assembler is then executed to convert the assembly language instructions into object format. The assembler deletes the temporary file after finishing.

Input File

The input file is a text file which contains the C source code. The file can be specified using a full path such as:

```
WDC02CC C:\SRC\HELLO.C
```

or by the file name alone if the file is located in the current directory such as:

```
WDC02CC HELLO.C
```

If the command that starts the compiler does not specify the extension of the file containing the C source, the compiler assumes that the extension is `.C`. For example, the command

```
WDC02CC PROG
```

compiles a file named *PROG.C* in the current directory. Although `.C` is the recommended file extension name, it is not mandatory. The specification

```
WDC02CC PROG.PRG
```

reads the file *PROG.PRG* from the current directory as the input to the compiler.

Input files can be created with the text editor of your choice, but the file must be straight text and can not contain any formatting commands such as those produced by a word processor.

Output Files

Creating an Object File

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file. By default, the object code generated by a compiler-started assembler is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.OBJ*. This file is placed in the directory that contains the C source file. For example, if you started the compiler with the command:



WDC02CC PROG.C

the file *PROG.OBJ* is created, containing the relocatable object file for the program. You may explicitly specify the name of the object file using the compiler option **-O**. For example, the command

WDC02CC -O MYOBJ.REL PROG.C

compiles and assembles the C source that is in the file *PROG.C*, writing the object code to the file *MYOBJ.REL*. When the compiler is going to start the assembler automatically, by default it writes the assembly language source to the file *CTMPXXX.XXX*, where `XXX' are numbers chosen such that the file name is unique. The file is placed in the directory defined by the **CCTEMP** environment variable. If **CCTEMP** is not defined, the file is placed in the current directory. The **CCTEMP** environment variable can be used to pass the intermediate assembly language file to the assembler through a RAM disk.

Creating an Assembly Language File

In some programs, you may not want the compiler to start the assembler automatically. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, use compiler option **-A**, which prevents the compiler from starting the assembler. When you specify option **-A**, by default the compiler sends the assembly language source to a file whose name is derived from that of the C source file, by changing the extension to *.ASM*. This file is placed in the same directory as the one that contains the C source file. For example, the command

WDC02CC -A PROG.C

compiles, without assembling, the C source that is in *PROG.C*, sending the assembly language source to *PROG.ASM*. When using option **-A**, option **-O** specifies the name of the file to which the assembly language source is sent. For example, the command

WDC02CC -A -O RAM:TEMP.ASM PROG.C

compiles, without assembling, the C source in *PROG.C*, sending the assembly language source to the file *TEMP.ASM* on the volume named *RAM:*. When option **-AT** is used, it causes the compiler to include the C source statements as comments in the assembly language source.

Searching for #Include Files

By default, the WDC C compiler searches the current directory to locate files specified in **#include** statements. It can also search a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories. Compiler option **-I** and the environment variable **WDC_INC_6502** define the directories in which the compiler searches for **#include** files. The compiler automatically searches the current directory for a **#include** file if the following conditions are met:

1. the compiler is started without specifying option **-I**,
2. there is not an **WDC_INC_6502** environment variable, and
3. the **#include** statement does not specify the drive and/or, directory containing the file.

If a **#include** statement specifies either the drive or directory, only that location is searched for the file.



Search Order #include

When the compiler encounters a **#include** statement, it searches directories for the file specified in the statement in the following order:

- if the filename is delimited by double quotes, ```filename"`, the current directory is searched.
- if the filename is delimited by angle brackets, `<filename>`, the current directory is searched only if no **-I** options are specified and if the **WDC_INC_6502** environment variable does not exist.
- directories specified in option **-I** are searched, in the order listed on the line that started the compiler.
- directories specified in the **WDC_INC_6502** environment variable are searched, in the order listed.

Compiler Options

Compiler Option Philosophy

Most of the compiler options are set up as toggles, which means that they can be either on or off. Most options default to off. The defaults can be changed by creating an environment variable, **CCOPT6502**. Options specified directly to the compile command will override options specified in the **CCOPT6502** environment variable. With a few exceptions, options are grouped around a common function. The first letter of an option identifies the group. The group letters are:

- A** Assembly language output control
- B** Debugging control
- M** Memory model control
- P** Parser control
- Q** Output control
- S** Optimization control
- W** Warning control

After the group letter, one or more individual options may be specified. If an individual option letter occurs and is not preceded by a 0(zero), the associated option is turned on. Multiple individual options can be specified. To turn an option off, the character 0(zero) must appear after the group letter and before the options to be turned off. **-P0T**, for instance, turns off trigraphs and **-PT** or **-P1T** turns them on. Combinations of options can be used to produce very specific results. To enable full ANSI syntax checking with the singular exception of trigraphs, for example, you would use the option **-PA0T**. The **A** option of the **P** group specifies full ANSI which includes trigraphs. The **0T** option turns trigraphs off. Since options are scanned left to right the combination **-PA0T** produces the desired result. **-P0T1A** would not produce the intended result. Since the **1A** option is scanned after the **0T** option, the **0T** option is cancelled.

CCOPT6502 Environment Variable

You can override the default settings of the compiler by using the environment variable, **CCOPT6502**. If you want to disable C++ style `//` comments as the default, for example, you could place the following line in your *AUTOEXEC.BAT* file:

```
SET CCOPT6502=-P0X
```

which would prevent the compiler from considering characters following a `//` as a comment till the end of the line. The **CCOPT6502** specification should have no blanks on either side of the equal sign. Options passed directly to the compiler override the **CCOPT6502** environment variable. If the **CCOPT6502** environment variable was set to **-P0X** and you specified **-PX** as a direct option to the compiler, then the **CCOPT6502 -P0X** option would be reversed. If you wish to specify more than one option with the **CCOPT6502** environment variable, then each option group must be separated by a blank. For example,

```
SET CCOPT6502=-P0X -MO -WO
```

would set the **-P0X**, **-MO** and **-WO** options. Note that **CCOPT6502** must be specified in upper case.



C Programs in ROM

This section discusses the general procedure of placing C code in ROM. It describes some of the choices that are available and the steps required to create the final output.

C Program Organization

After compilation and linking, a C program consists of three sections: code, initialized data, and uninitialized data. The difference between initialized and uninitialized data is demonstrated by the following two C statements:

```
int x = 1;
int y;
```

In the first statement, the global variable *x* has memory space allocated for it which is initialized to contain the value 1. The second statement allocates memory space for the variable *y*, and by C convention, is initialized to zero by default. The second variable, *y*, is considered uninitialized data since it is never explicitly set to a particular value. In a C program, all uninitialized data is collected together for efficiency. Otherwise, there might be a lot of zeros spread throughout the data segment. This is especially important in a ROM-based system where instead of copying zeros from ROM to RAM, it is much more efficient just to clear the uninitialized area of RAM to zero.

System Organization

To operate correctly, a system must contain as much ROM and RAM as are needed to get the job done. The ROM is located at whatever address is convenient. Usually the ROM is located at the high end of bank zero so that the interrupt and reset vectors have defined values when the system is powered up. Some amount of RAM is also required for the stack and direct page access. The main program code is stored in the ROM which is where it is usually executed from since the code itself does not change. Initialized data is also stored in ROM so that it is available when the system is powered up. However, initialized data often represents the initial state of variables that may be changed by the program. Since variables in ROM can't change, the variables must be in RAM. When the system is reset, the initial values in ROM are copied to the RAM locations. Thus, initialized data has two locations, the RAM address where the program code will access it and a ROM address where it is copied from. The WDC C development system is set up so that the initialized data can be stored in ROM immediately following the program code. This allows the startup code to know where to find the initialized data so it can be copied to RAM.

Creating A ROM Program

Let's assume that we are creating a ROM program. To compile we would use:

```
WDC02CC MYPROG.C
```

Included in the `WDC_SDS\LIBSRC\S65C02` directory is an example startup assembly language source file. Copy the one called `STARTUP.ASM` into the current directory. This file contains the reset and interrupt vectors and a short sequence of code that sets up the machine stack pointer, the pseudo-stack pointer, copies the initialized data to RAM, clears out the uninitialized data and then transfers control to `main()`. The default is for the machine stack to start at `0x1FF` and the pseudo-stack to start at `0x7fc0` and we'll leave them there. Assemble the startup file by using the command:

```
WDC02AS STARTUP.ASM
```

Now, we need to create the output file. For this example, let's say that we want the code to be located in ROM at location `0x8000` and that the data will be located in RAM starting at location `0x200`. The initial values of the data will actually reside in the ROM immediately following the program code. To link the modules together, we would use the command:

```
WDCLN -HM28 -C8000 -D200, MYPROG.OBJ STARTUP.OBJ -LC
```



The first option to the linker specifies that the output will use Motorola S-28 records which have a 24 bit address field. (A full description of the linker and its options can be found in the Assembly Language Development System manual.) Next, the address of the code is specified to be at location 0x8000. The **-D** option tells the linker to locate the address of the data at 0x200, but to place the actual output immediately following the code. The address options seem complicated but are not really too difficult. In general, if no addressing option is given, then one section is placed immediately following the preceding section. The first section is the code, followed by initialized data, and finally uninitialized data. Each address option consists of an address where the code or data is eventually expected to be located. For code, this is the address where subroutine calls will be made to and is usually in ROM. For data, this is the address where program code will load and store into and is usually in RAM. Following the 'use' address is an optional comma and a physical location address. The physical location address is the address that will be placed in the hex records generated by the linker. For program code, the physical and use addresses are almost always the same. Thus, to locate code at location 0x8000, the option:

```
-C8000,8000
```

would be correct. However, the linker assumes that if no comma and second argument are present, then the physical and use addresses are the same. Thus, the preceding option could also be given as:

```
-C8000
```

For initialized data, the use address is typically in RAM, while the initial physical location is in ROM. One way to handle this is to specify the exact address in RAM and in ROM as in:

```
-D200,8800
```

which would place the data at location 0x8800 in ROM although the program code would be looking for it at location 0x200. One of the problems with this approach is that we would have to know that the size of the program was less than 0x800 bytes or the data would overwrite the program code or vice-versa. An easier method would be to use the option:

```
-D200,
```

which tells the linker to locate the data for use at location 0x200, and by using a comma without an address, to place it physically right after the preceding section which would be the program code. The assembly language startup routine in *STARTUP.ASM* assumes that the initialize data is located immediately following the code section and copies it to its 'use' location in RAM. Further discussion of sections and linking as well as a full analysis of a startup routine can be found in the Assembly Language Development System manual.



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 2 WDC02CC

The WDC C compiler is a full ANSI standard implementation. There are also a number of useful extensions to the ANSI standard which are controlled by compiler switches.

Configuring the Program

The WDC 65C02 C compiler makes use of a number of zero-page locations to generate smaller and faster code and to implement a 16-bit stack pointer. The exact locations used by the compiler are specified in two files: *WDC_SDS\BIN\WDC02CC.CFG* and *WDC_SDS\INCLUDE\ZPAGE.INC*. The first file is used by the compiler and debugger to know the locations. The second file is included by assembly language programs in the library. **Note that any changes to these files must be made to both files and the library and any user programs must be recompiled and reassembled.** Because different applications may have different zero-page requirements and availability, it is possible to change the configuration to match the project at hand. This is done by editing the two files mentioned above and then rebuilding the library and application. **Note that if the default values are acceptable, no changes need to be made.** The zero-page locations are broken into groups which do not need to be contiguous. The following sections describe the zero page locations and their use.

User Registers

The compiler uses some zero page locations for its own temporary pseudo-registers. The compiler can also use zero page locations for user defined pseudo-register variables. This is very important for generating smaller faster code, especially when pointers are used. For example, when a pointer is dereferenced like **cp*, the code normally generated if *cp* is a local on the pseudo-stack would be:

```
LDY    #250           ; offset of cp within frame
LDA    (FP),Y
STA    R0
INY
LDA    (FP),Y
STA    R0+1
LDA    (R0)
```

If *cp* is a register variable, the code would be:

```
LDA    (cp)
```

which is clearly smaller and faster. Register variables can be allocated using the *register* keyword or by using the compiler -**SR** option, they can be allocated automatically. The configuration for user registers consists of two values, the base address for the user registers and the number of bytes available for user registers. The *WDC_SDS\BIN\WDC02CC.CFG* file contains the following two lines:

```
User Register Count=16           ; count of user register bytes
User Register Start=0x20         ; N byte user register area
```

where *User Register Count* is the number of bytes available (the default is 16) and *User Register Start* is the base address (the default is 0x20). If zero-page space is at a premium, the *User Register Count* can be set to zero with no adverse effects. This is the only area with a variable size.

The *INCLUDE\ZPAGE.INC* file contains the following lines:



***** User Register Start and Count *****

```

        org    $20    ;16 byte User Register area
REGS   rmb    16    ; User Register Count

```

These lines should be changed to reflect any changes made to the *WDC_SDS\BIN\WDC02CC.CFG* file.

Stack Registers

There is an eight byte stack register are which holds the return address, the pseudo-stack pointer and two frame pointers. The line to be changed in *WDC_SDS\BIN\WDC02CC.CFG* is:

```

Stack Register Start=0x30          ; 8 byte stack area

```

The default start of the stack area is 0x30.

The lines to be changed in *WDC_SDS\INCLUDE\ZPAGE.INC* are:

***** Stack Register Start *****

```

        org    $30          ; 8 byte Stack Register area
VAL     rmb    2
SP      rmb    2
AFRAME
FRAME   rmb    2
LFRAME
PC      rmb    2

```

Only the **ORG** statement needs to be adjusted.

Compiler Registers

There are five four-byte pseudo-registers used by the compiler to hold intermediate results and return values from functions. The twenty byte area is defined in the configuration file with the following line:

```

Compiler Register Start=0x38      ; 20 byte compiler temporary register area

```

The default start of this area is 0x38.

The lines to change in the include file are:

***** Compiler Register Start *****

```

        org    $38      ; 20 byte tmp register area
R0     rmb    4
R1     rmb    4
R2     rmb    4
R3     rmb    4
R4     rmb    4

```

Only the **ORG** statement needs to be changed.



Work Area

There is an eight byte general purpose work area used by the compiler support subroutines. This area is defined by the following line in the configuration file:

```
Work Register Start=0x4c           ; 8 byte general purpose work area
```

and by the following lines in the include file:

```
***** Work Register Start *****
```

```
org    $4c           ;8 byte zero page temporary area
```

```
TMP    rmb    8
```

Floating Point Registers

The compiler and floating point library functions perform all floating point operations using two zero-page floating point registers. Each register is eleven bytes for a total of twenty-two bytes. The format is “double” IEEE-754 standard 1985, 64 bit mantissa (8 bytes), 16 bit exponent (2 bytes), 1 bit sign. If floating point is not used, then these bytes are free for use for other purposes.

The line in the configuration file is:

```
Float Register Start=0x54         ; 22 byte floating point area
```

while the lines in the include file are:

```
***** Float Register Start *****
```

```
org    $54
```

```
Exp0   rmb    2           ; 11 byte area for FP0
Sign0  rmb    1           ; must be in this order
FP0    rmb    8
```

```
Exp1   rmb    2           ; 11 byte area for FP1
Sign1  rmb    1           ; must be in this order
FP1    rmb    8
```

Running the Program

The format of the WDC C compiler command is:

```
WDC02CC [OPTIONS] SRCFILE
```

where *SRCFILE* is the name of the C source file, and the brackets around options indicate that they are not required. The compiler only deals with a single file at a time and must be restarted to compile any additional files.



Option Summary

- A** Causes the compiler not to start the assembler after it has compiled a program. Only an assembly file is created.
- AT** Same as **-A**, but also imbeds C source statements into the assembly code.
- BS** Generate source level debugging information.
- D** Defines a symbol for the preprocessor.
- HI** Read pre-compiled header file into symbol table.
- HO** Create a pre-compiled header file.
- I** Specifies path for include files.
- L** Creates a listing from the compiler output to be passed to the assembler output.
- LT** Generates listing with embedded source statements
- LW** Modifies the **-L** option to set the listing to a wide format of 132 columns
- MO** Generate module headers for library functions.
- MU** Place initialized const data in the compiler CONST section.
- MV** Place string data in the compiler STRING section.
- O** Specifies name to be used for output files.
- PA** Turns on ANSI preprocessor and trigraphs. Turns off non-ANSI options.
- PB** Make bitfields unsigned by default.
- PC** Allows extra characters after **#endif** or **#else**.
- PE** Causes enums to occupy only the amount of space needed.
- PP** Make characters default to signed.
- PT** Looks for trigraphs in the input stream.
- PX** Allows C++ style comments.
- QA** Causes generated prototypes to use **__PARAMS()** syntax.
- QP** Generates prototypes for all non-static functions.
- QQ** Disables startup and error messages from being displayed.
- QS** Generates prototypes for all static functions.
- QV** Generates verbose information on memory usage.
- SF** Generates an optimized for(;;) loop.
- SM** Defines the **__C_MACROS__** macro.
- SO** A shorthand way of specifying **-SFMR**.
- SP** Invoke the post-pass peephole optimizer, **WDC02OPT**.
- SR** Automatically use register variables.
- SX** Enable factorizing in the Optimizer on a source file. (**-SP** must also be specified!)
- WA** Complains on arguments which do not match the prototype specification.
- WD** Generates warnings for non-prototype style function definitions.
- WE** Quit on warnings. Treats warnings as errors.
- WL** Shorthand for **-WARU** and stands for lint.
- WN** Do not generate warnings on direct pointer to pointer conversions.
- WO** Causes pointer/int conflicts to generate warnings rather than errors.
- WP** Generates a warning if a function is called without a prototype being defined for the function.
- WQ** Print warnings and errors to the file **WDC.ERR**.
- WR** Warns if function return type does not match declared type.
- WS** Ignores all warnings.
- WU** Warns about unused local variables.
- WW** Allows compiler to continue beyond 5 errors.



Option Descriptions

-A

Normally, the compiler generates assembly language to a temporary file that is assembled by the assembler and then deleted. This option tells the compiler to only generate an assembly language file. The default name of the created assembly language file is formed by taking the root part of the source file name and appending *.ASM*. The **-O** option can be used to exactly specify the name of the assembly language output file.

For example:

```
WDC02CC -A MYFILE.C
WDC02CC -A -O SAVE.ASM MYFILE.C
```

The first command compiles the file *MYFILE.C* and places the assembly language output into a file called *MYFILE.ASM*. The second example compiles the same source file, but places the output into the file *SAVE.ASM*.

-AT

This option is identical to the **-A** option except that the compiler copies the C source statements to the assembly language output file as comments.

For example:

```
WDC02CC -AT MYFILE.C
```

This command compiles the file *MYFILE.C* and places the assembly language output into a file called *MYFILE.ASM* with the C statements embedded as comments.

WARNING: The option `-AT` can hide error messages and cause problems for the optimizer.

-BS

This option tells the compiler to add source level debugging information to the statements it generates in the output assembly language file.

WARNING: Do not use the `-MO` option with the `-BS` option. This will cause a conflict.

For example:

```
WDC02CC -AT -BS MYFILE.C
```

This command compiles the file *MYFILE.C* and places the assembly language output into a file called *MYFILE.ASM*. This file has the C statements embedded as comments and additional statements that specify line number and source level debugging information.

-D

This option defines a symbol in the same way as the preprocessor directive, **#define**. Its usage is as follows:

```
-DMACRO[=TEXT]
```

For example,

```
-D MAXLEN=1000
```

is equivalent to inserting the following line at the beginning of the program:

```
#define MAXLEN 1000
```



The separating space following the **-D** is optional. The following formats are equivalent:

```
-D MAXLEN=1000  
-DMAXLEN=1000
```

Since option **-D** causes a symbol to be defined for the preprocessor, it can be used in conjunction with the preprocessor directive, **#ifdef**, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef DEBUG  
    printf("value: %d\n", i);  
#endif
```

This debugging code would be included in the compiled source by the following command:

```
WDC02CC -DDEBUG PROGRAM.C
```

When no substitution text is specified, the symbol is defined as the numerical value one.

-HI

This option specifies the name of an input file containing pre-compiled header file information. The input file is created using the **-HO** option. For example:

```
-HI MYPROG.DMP.
```

-HO

This option specifies the name of an output file for pre-compiled header information. This information is later used with the **-HI** option to compile programs that use the same header files over and over again. The use of pre-compiled header files can significantly shorten compile times. Only one pre-compiled header file may be used per compilation, but it may contain information from numerous header files. For example, if the C source files for a program included the header files *STDIO.H*, *STDLIB.H* and *STRING.H*, you could speed up compilations by pre-compiling these files. To do that, create a dummy file named *X.C* that contained the following lines:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>
```

Create the pre-compiled header file *X.DMP* using the following command line:

```
WDC02CC -HO X.DMP X.C
```

Then, when compiling a source file that includes these header files you can use a command like:

```
WDC02CC -HI X.DMP MYPROG.C
```

When the compiler encounters a **#include** that references one of the files in the pre-compiled header file, it just skips it.



-I

Compiler option **-I** defines a single directory to be searched for the file that was specified in a **#include** statement. The path descriptor follows option **-I**. A space between **-I** and the path descriptor is allowed but not required.

The command

```
WDC02CC -I C:\OTHER\INCLUDE
```

directs the compiler to search the *c: ++ other ++ include* area when looking for an include file. Multiple **-I** options can be specified when starting the compiler, if desired, thus defining multiple directories to be searched. You can also specify where include files are kept using the **WDC_INC_6502** environment variable.

-L

This option creates a listing from the compiler output to be passed to the assembler output. The 'C' source code statements are added as comments to the assembly code listing.

-LT

This option is used to generate a listing that contains embedded source statements.

-LW

This option modifies the **-L** option to set the listing to a wide format of 132 columns. This option is the same as the PW 132 option in the assembler.

-MO

This option causes the compiler to generate a module header and ending for each function and global variable in the source file. This allows each function to be loaded separately from a library of functions. This option is usually only used when compiling source files for inclusion in a library.

WARNING: Do not use the **-BS** option with the **-MO** option. This will cause a conflict.

-MU

This option forces initialized data declared using the *const* keyword to be placed in the compiler's **CONST** section which defaults to **KDATA**. See the discussion of *CONSTS AND STRINGS* in CHAPTER 3 for more details and an example.

-MV

This option forces string data to be placed in the compiler's **STRING** section which defaults to **KDATA**. See the discussion of *CONSTS AND STRINGS* in CHAPTER 3 for more details and an example.

-O

This option is used to override the default output filename. The separating space between the option **-O** and the filename is optional. A detailed discussion of compiler output files appears in CHAPTER 1.

-PA

This switch is used to turn on all the ANSI checking and turn off any special extensions. A program which compiles with the **-PA** flag should compile with any other ANSI standard C compiler.



-PB

This option causes bit fields to be treated as unsigned. The default is signed. For example,

```
int pFlags : 3;
```

by default defines a bit field whose value ranges from -4 to +3. If the **-PB** option is specified, the range is from 0 to 7.

-PC

This option allows extra characters to occur on the line following the pre-processor directives **#else** and **#endif**. This option is provided for compatibility with pre-ANSI compilers.

-PE

By default the size of enums is the same as that of an int. This option places them in the smallest space that will contain them. For example, the definition:

```
enum colors { blue, red, yellow, white};
```

would use a single byte to represent enums of this type if the **-PE** option was used. The definition:

```
enum countries { USA, England, France=500, Germany};
```

would use a two byte word to represent enums of this type when using the **-PE** option.

-PP

The 65C02 does not have an instruction which will sign extend an eight bit quantity to a sixteen bit one. Thus, unsigned character operations are much more efficient than signed character operations. The WDC C compiler chooses to treat *char* directives as specifying *unsigned char* quantities. The *signed char* directive may be used to declare characters that can be positive or negative. The **-PP** option changes the default *char* declaration to be *signed char*.

-PT

Trigraphs are an attempt by the ANSI committee to support foreign keyboards and printers. Certain characters are represented as two question marks followed by another character which indicates the true character intended. For example, `??=` is equivalent to `#` and `??(` is equivalent to `[`. In most instances, this simply slows down the parser, but it must be supported for ANSI compliance. The option **-PA** turns on trigraph checking, but this option is included to turn off trigraphs while retaining the rest of the ANSI compliance.

-PX

This option allows C++ style single line comments. When this option has been turned on, when the compiler pre-processor sees `//`, it treats the two slash characters and the rest of the line as a comment.



-QA

This option controls how prototypes are generated by the **-QP** and **-QS** options. If **-QA** is specified, then a typical prototype is generated as:

```
int func _PARMS((int x, int y));
```

instead of the default:

```
int func(int x, int y);
```

The first form is used for maximum portability to pre-ANSI compilers. The following sequence is usually placed at the beginning of the header file containing the portable style prototypes:

```
#if __STDC__
#define _PARMS(x) x
#else
#define _PARMS(x) ()
#endif
```

Then, if `__STDC__` is defined, the macro will change the first prototype to:

```
int func (int x, int y);
```

Otherwise, if it not defined, the prototype becomes:

```
int func ();
```

which is the standard declaration of an external function defining the type returned by the function.

-QP

This option tells the compiler to generate a file with the extension *.PRO* that contains prototypes for all the non-static functions found in the current source file. The file is not compiled.

-QQ

This option prevents the compiler from displaying the startup message and the number of errors.

-QS

This option is similar to the **-QP** option except that it causes the compiler to generate prototypes for static functions.

-QV

This option causes the compiler to display information on memory usage.

-SF

This option causes the compiler to generate more efficient **for** loops. This is an option since the less efficient loops are easier for source level debugging.



-SM

This option defines the macro `_ _C_MACROS_ _`. Some header files use this macro to redefine functions as macros that removes the function call overhead in favor of in-line code. See `Wdc_Sds\Include\ctype.h` and `Wdc_Sds\Include\stdio.h`.

-SO

This option is the general purpose optimization option that is the same as specifying **-SFMR**.

-SP

This options causes the post-pass peephole optimizer, WDC02OPT, to be invoked on the assembly language output of the compiler. If the **-A** option has not been specified, then WDC02OPT will invoke the assembler, WDC02AS on it's output.

-SR

This option instructs the compiler to automatically allocate register variables. These register variables are pseudo-registers located in zero-page. The compiler allocates registers to local variables based on how often they are used, with extra weight given to pointers. Without this option, the compiler will honor any register declarations on a first come first served basis.

-SX

This option enables factorizing in the Optimizer. Factorization is done on a single source file. The **-SP** option **MUST** also be specified for factorization to take place. Please see Chapter 3, Technical Notes, for more information on factorizing,

-WA

Normally, if you call a function in the presence of a prototype, and the type of a function parameter does not match the type specified in the prototype, the type is coerced as if by assignment. Thus if you pass an int to a function expecting a long, the int is quietly cast to a long. This option causes the compiler to generate a warning if a quiet cast is generated. This is useful, since it allows you to change the code to an explicit cast that is portable to pre-ANSI compilers.

-WD

This option generates a warning if a function is defined using the old pre-ANSI style definition of the form:

```
int function(a,b)
int a, b;
{ }
```

instead of:

```
int function(int a, int b)
{ }
```

This is a useful tool for ``ANSI-izing" your programs.

-WE

Normally warnings and errors generated by the program are displayed on the screen of the computer. If errors occur, the compiler stops without producing an object file. If only warnings occur, an object file is produced. This option converts all warnings into errors.



-WL (Currently not available)

This option is a short-hand way of specifying the **-WA**, **-WR** and **-WU** options. It is used to force the compiler to do maximum type checking.

-WN

This option suppresses warning messages for direct pointer to pointer conversions which do not contain an explicit cast, such as the following code fragment:

```
int *iptr;
char *cptr;

cptr = iptr;
```

Because the ANSI standard defines that such direct conversions are illegal, the **-PA** (full ANSI) option will always generate an error for the above code. To eliminate the error when **-PA** is specified, the above could be changed to:

```
cptr = (char *)iptr;
```

-WO

Under ANSI C, pointer-integer conversions are illegal, and are usually a problem when 16 bit ints are used. This option changes pointer-integer conversion errors into warnings, and has been included for compatibility with non-ANSI compliant compilers.

-WP

This is useful for determining if a header file is not being included. For example, if you use the *strlen()* function without including the file *string.h*, then **-WP** will generate a warning.

-WQ

By default the compiler displays error messages on the screen. This option sends them in an abbreviated form to the file *WDC.ERR*.

-WR

Normally, a function is declared without a type, which implies that it returns an int. However, if there are any return statements, explicit or implicit, that do not return a value or that return a value whose type disagrees with the type of the function, then a warning is generated for the return statements. Thus, the function:

```
foo()
{
    printf("hello world\n");
}
```

would generate a warning since it should have been defined:

```
void foo ()
```

since there is no value returned. The warning would be on the `}` which is an implicit return statement.



-WS

This option causes all warning messages to be suppressed.

-WU

Since the compiler has the entire tree of the function in memory, it can detect that a variable has been declared but has not been used. This option directs the compiler to check for such variables and generate a warning for each that is not used. For example, the function:

```
void foo()
{
    int j;

    printf("hello world!\n");
}
```

would generate a warning that *j* is not used in the function.

-WW

Normally, the compiler will pause after encountering five errors and ask if compilation should continue. This option indicates that the compiler should not pause and should continue to the end of compilation.



CHAPTER 3 Technical Notes

This section contains information about the implementation of the compiler for the W65C02S processor.

Pseudo-Registers

The 65C02 has very limited resources for generating code from a compiled language like C. To facilitate the code generation and to generate programs that are as small as possible, the code generated by the compiler uses a set of zero-page memory locations as a set of pseudo-registers. These registers implement full 16 bit stack and frame pointers, compiler temporary registers, floating point registers and user-defined register variables. The use of user-defined register variables in particular can greatly reduce the size and increase the speed of program execution. The primary cost for these pseudo-registers occurs when a function is called and when a function returns. When a function is called, the arguments are pushed onto the machine stack and a JSR is executed to the function start address. At the function start address is a JSR to the csav routine followed by four bytes of data. The csav routine is an assembly language routine that performs a number of function startup actions. First, it uses the in-line data, to determine the number of arguments on the stack, the amount of stack space to allocate for automatic storage and the amount of user register space used within this function. Second, it adjusts the pseudo-stack pointer and saves the old pseudo-stack pointer, the old frame pointers and the return address on the pseudo-stack. Third, it copies the arguments from the machine stack to the pseudo-stack. Fourth, it saves any user-registers that will be used by the function. Finally, it pushes the address of the cret routine onto the machine stack and jumps to the address after the in-line data. All code generated by the compiler makes use of the pseudo-frame pointers to access automatic variables and arguments passed to the function. The code also uses and zero-page user register variables allocated.

NOTE: Because of the interdependence between the compiler support library functions, the code generated by the compiler and the debugger, the locations used for pseudo-registers must be consistent. To guarantee this consistency, any changes made to zero-page pseudo-register allocation must be made to both the WDC_SDS\INCLUDE\ZPAGE.INC and WDC_SDS\BIN\WDC02CC.CFG files and the libraries must be rebuilt.

Pseudo-Stack Frame

The pseudo-stack frame created by the csav routine has the following structure:

```
*****
* copy of args *
* from machine *
* stack *
* ( N bytes ) *
*****
* # of bytes of *
* user registers *
* ( see below ) *
* ( 1 byte ) *
*****
* local frame ptr *
* ( 2 bytes ) *
*****
* args frame ptr *
* ( 2 bytes ) *
*****
* stack pointer *
* ( 2 bytes ) *
```



```

*****
* return address *
* ( 2 bytes ) *
*****
* any saved user *
* registers *
* ( N bytes ) *
* ( see above ) *
*****
* local variables *
* allocated on *
* pseudo-stack *
* ( N bytes ) *
*****

```

Function Calls and Argument Passing

When a function is called with arguments, there are two different methods that can be used to restore the stack to its original condition. The more traditional C method is to have the calling function pop the arguments after the function call returns. A more efficient method is to have the function that is called pop the arguments before returning. This method is more efficient since the code to pop arguments occurs once, instead of each place that the function is called. There are two drawbacks to this approach. First, if a variable number of arguments are passed (printf() for example), there is no way for the called function to know how many arguments to pop since it may vary from call to call. Second, if the wrong number of arguments is passed, the stack pointer becomes inconsistent which almost always results in a program error. Both of these problems are completely avoided by the use of function prototypes. If a function prototype declares a function to have a variable number of arguments, the compiler automatically loads the accumulator with the number of bytes of arguments to pop off the stack. The function itself must be declared with a variable number of arguments so that it knows to use extra data when popping arguments. The use of prototypes also avoids the second problem by always guaranteeing that the correct number and types of parameters are passed.

Function Arguments

When calling a function the arguments are pushed onto the stack in reverse order, starting with the last argument. The last argument pushed is the first one listed in the function call. For example, when calling a function with two *in* arguments, the following shows the C version of the function call and the assembly language generated:

```

;      func(a, b);

      LDA      b+1
      PHA
      LDA      b
      PHA
      LDA      a+1
      PHA
      LDA      a
      PHA
      JSR      _func

```



If there are a variable number of arguments, after the last argument is pushed, the total number of argument bytes is loaded into the Accumulator. For example:

```
    ;      func(a, b);  
  
    LDA      b+1  
    PHA  
    LDA      b  
    PHA  
    LDA      a+1  
    PHA  
    LDA      a  
    PHA  
    lda      #4  
    JSR      _func
```

Function Return Value

Functions called by a C function and C functions themselves return values in the R0 pseudo-register as defined in the ZPAGE.INC header file. Floating point return values are returned in FP0.

Startup Code

In the WDC_SDS\LIBSRC\S65C02 directory, there is an example assembly language startup file called *STARTUP.ASM*. This file is designed to be customized for the particular program being designed. It contains a set of interrupt and reset vectors, initialization code sets the machine stack pointer and the pseudo-stack pointer, copies any initialized data from ROM to RAM, clears any uninitialized data to zero and calls the *main()* routine. A detailed description of a similar routine called *STARTUP.ASM* can be found in the Assembly Language Development Manual.

Identifier Name Prefixes

The compiler automatically generates a prefix for all global symbols. This prefix is in the form of a leading underscore for all global symbols. By convention, library functions that are used directly by the compiler begin with a tilde character.

Memory Management

The 65C02 memory is organized as a single 64K bank which may need to be broken into sections to accommodate holes in the memory map. The problem with breaking a program and its data into sections is that there needs to be a way to indicate which functions and data are placed in which section. Currently, the compiler generates three sections, CODE, DATA, and UDATA, for program code, initialized data and uninitialized data. An additional section, KDATA, is generated for constant data and/or string data if the appropriate option is specified. Using #pragmas or compiler options, these can be changed to any section name desired. Then the location of the section in memory can be specified at link time or within the pragma or option itself. This provides total control over the location of each function and data item in memory.



CAVEATS

Floating Point Considerations

All operations are assumed to be “double”.

The compare, subtraction, multiplication, etc. are treated as “double” operations. The constants are converted to “doubles”. If you follow the constants with an f, as in 0.0f, it will do a single precision compare or subtract. If the constant is a double, (which it is by default), then the operations used are promoted to double as well.

Manual optimization of double to float can be accomplished. The “f” modifier can be used to generate faster and perhaps smaller code. Be especially watchful for compound equivalencies, i.e. +=, -=, etc...

Example #1: (where x is a float)
if (x<0.0) can also be written as: if (x<0.0f) or if (x< (float)0.0)
The addition of the “f” forces the compiler to stay in floating point mode.

The transcendental functions can also be rewritten to stay mostly in floating point, so as to be about three (3) times faster.

Example #2:
X += 0.1234567; should be written as x+= 0.1234567f;

Section Pragma

The SECTION pragma is used to redefine one of the names of the predefined sections used by the compiler. The default section names and their use are:

- CODE - all program code
- DATA - all initialized data
- UDATA - all uninitialized data
- KDATA - all initialized, non-modifiable data

By default, the compiler only uses the first three. Constant initialized data is treated as modifiable initialized data unless the -MU option has been specified. This option places all initialized data that has been qualified as not being modifiable in the CONST section which defaults to KDATA. Strings are handled separately and by default are placed in the DATA section. If the -MV option is specified, then strings are placed in the STRING section which also defaults to the KDATA section. The syntax of the SECTION pragma is:

```
#pragma SECTION TYPE=name[,qualifier[,...]]
```

The type names recognized by the pragma and their default values are:

<u>TYPE</u>	<u>DEFAULT</u>	<u>QUALIFIER</u>
CODE	CODE	OFFSET
DATA	DATA	REF_ONLY
UDATA	UDATA	
CONST	DATA	unless '-MU' then KDATA
STRING	DATA	unless '-MV' then KDATA

The qualifiers are the same as those for the SECTION directive in the assembler. The compiler will generate a SECTION directive the first time that a new section name is defined. Any extra qualifiers will be copied exactly to the SECTION directive in the assembly language output file. See the SECTION directive in the Assembly Language manual for more information.



Consts and Strings

Two options are available for manipulating constant data and strings. Constant data is any initialized data declared using the *const* keyword. Normally both are just considered part of initialized data and are not handled any differently. By using the **-MU** option, constant data will be placed in what the compiler considers the CONST section. This section defaults to the KDATA section at the assembly level. To change the CONST section, use the *section #pragma* statement.

```
#pragma section CONST=yournamehere
```

Similarly, using the **-MV** compiler option will place strings in the compiler STRING section. This section also defaults to KDATA.

In-Line Assembly Code

Assembly language statements can be embedded within C source code by surrounding the assembly code with the statements **#asm** and **#endasm**. Embedded assembler code should make no assumptions about the contents of the registers.

For example,

```
#define NUM 3

void
test(void)
{
#asm
    lda #%%NUM
    sta $c0
#endasm
}
```

Since **#asm** and **#endasm** are pre-processor directives, the lines between would normally be considered white space by the compiler. To avoid this, a null statement is generated for the parser. Support is also provided to access predefined macros, file scope static identifiers, and global variables. To access these from within a **#asm** block, precede the identifier name with **%%**. When passing output to the assembly language file, the compiler checks for the pattern **% %symbol**. When found, it checks to see if *symbol* matches any pre-defined macros and replaces it with the actual macro value. If a macro is not found, it next checks for variable name matches following the standard C scoping rules. If found, the proper variable name is substituted. For global variables, the appropriate **`_'** character is prepended to the global symbol name.

The following directives can be used before and after in-line assembly statements:

asmstart and **asmend**

The Optimizer will treat any lines that are in between these two directives as comments.



ASM Keyword

The compiler now supports an *asm* keyword. The statement following the keyword is passed to the output file directly and is not interpreted by the compiler except for macro substitution. The statement must be terminated by a `;`. Multiple statements, each terminated by a `;` may be enclosed within braces. Variables may be referenced by name using the `%%` convention discussed under the *#asm* preprocessor directive. Labels must be followed by a colon.

Example

```
#define VALUE 20
#define COUNT 100

void test( void )
{
    asm cli;

    asm { lda %%VALUE; ldx %%COUNT;
          mylab: sta >$1000,X; dex; bne mylab; }

#define clrmem asm { lda %%VALUE; ldx %%COUNT; \
                    loop: sta >$1000,X; dex; bne loop; }

    clrmem;
}
```

Other Preprocessor Features

Predefined Names

`__DATE__` A string literal containing the date of compilation, in the form of “Mmm:dd:yyyy”.

`__TIME__` A string literal containing the time of compilation, in the form of “hh:mm:ss”.

`__STDC__` The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations.

#line

The *#line* directive changes the contents of `__LINE__` and `__FILE__`, which are predefined identifiers in the compiler. The `__LINE__` identifier contains the line number of the currently compiled line code. The `__FILE__` identifier is a string that contains the name of the source being compiled. The general form for *#line* is

```
#line number “filename”
```

where *number* is any positive integer and becomes the new value of `__LINE__`, and the optional *filename* is any valid file identifier, which becomes the new value of the `__FILE__`. *#line* is primarily used for debugging and special applications.



Producing Optimum Code

There are many things that can greatly improve the quality of the code generated by the compiler. The most important is to use unsigned ints and chars whenever possible. Signed chars and ints require many more instructions when sign extending or comparing. As a result chars are unsigned by default and you must use the `signed` keyword to get signed chars.

Optimizer

This version of the compiler comes with a post-pass-peephole optimizer called WDC02OPT. This optimizer operates on the assembly language output of the compiler and performs a number of optimizations that generally save about 10 percent on code size depending on the program. To invoke the optimizer, use the **-SP** option. If **-SP** is used in conjunction with **-A**, the output file will contain the optimized assembly language. Note that the optimizer makes assumptions about the format of the assembly language output of the compiler and will not operate properly on hand-written assembly language programs.

The following optimizer options support the corresponding compiler options:

- L Call the assembler with the listing option
- LW Call the assembler with the wide listing option.
- K Causes the path name specifying the name of the listing file to be placed in the reserved word `__FILE__`.

Floating Point

The compiler supports IEEE single and double precision floating point variables and arithmetic. No option is necessary to access the floating point feature. All floating point operations with the exception of assignment are performed by assembly language routines using two pseudo-registers located in zero-page. These assembly language routines are located in a special floating point library along with the standard mathematical library routines. Please see the library CHAPTER for the names and linking conventions. The floating point library also contains versions of **printf** and **scanf** that work with floating point values.

Referencing I/O

Hardware addressing of I/O can be done from 'C' in many ways. The following examples we compiled as the Small Memory Model.

The `#define` has advantages that it can easily address across page boundaries. One of the fastest & least amount of code generated is:

For 8 bit I/O you can use

```
#define comx ((volatile uchar *)0x200003) // used for indexed addressing
x = *comx;
```

This produces the code:

```
sep    #$20
lda    >2097155          ;This is a long addressing mode (32 bit)
sta    <L3+x_1
rep    #$20
```

To index

```
comx[2] = 0x02;
```

This produces the code:

```
longa  off
sep    #$20
lda    #$2
sta    >2097157
```



comx[0] = 0x31; //Xmit char '1'

This produces the code:

```
lda    #$31
sta    >2097155
longa  on
rep    #$20
```

Therefore to address a structure for a UART, one way could be to:

```
#define comx ((volatile uchar *)0x200003) // used for indexed addressing
```

```
#define RECV 0
```

```
#define IER 1 //Interrupt Enable Register
```

```
#define ISR 2 //Interrupt Status Register
```

```
#define LCR 3 //Line Control Register
```

```
#define MCR 4 //Modem Control Register
```

```
#define LSR 5 //Line Status Register
```

```
#define MSR 6 //Modem Status Register
```

```
#define SPR 7 //Scratchpad Register
```

to address the Scratchpad Register would be:

```
comx[SPR] = 0x00;
```

Standard Addressing modes for 'C' product larges & slower code because all references are indirect (i.e. pointers)

For 8 bit I/O you can use

```
volatile unsigned char *rtc2_year = (unsigned char *)0xA080; //
    x=*rtc2_year; //Read data from Address
```

This produces the code:

```
lda    |__rtc2_year
sta    <R0
sep    #$20
lda    (<R0)
sta    <L3+x_1
rep    #$20
```

For 16 bit I/O you can use

```
volatile int *hw16_multiplier = (int *)0xA080; //
    *hw16_multiplier = y;
```

This produces the code:

```
lda    |__hw16_multiplier
sta    <R0
lda    <L3+y_1
sta    (<R0)
```

For 32 bit I/O you can use

```
volatile long *hw32_multiplier = (long *)0xA080; //
    *hw32_multiplier = z;
```

This produces the code:

```
lda    |__hw32_multiplier
sta    <R0
lda    <L3+z_1
sta    (<R0)
lda    <L3+z_1+2
ldy    #$2
```



Another example of referencing I/O using the pre-processor:

```
#define VIA_BASE    F0H
#define IO_VIA_DDRBNH  (VIA_BASE+2)
#define io_and_port2(port, mask, twenty) asm{ php; longa off; sep twenty; lda port; and mask; sta port; plp;}
io_and_port2(IO_VIA_DDRBNH, #$72, #$20)
```

Prototyping Functions

The W65C816 C Compiler supports the new definitions of declaring parameters. If the old style declaration is used, an error will occur. For Example;

OLD STYLE:

```
//Prototype
VOID foo (float f);
.
VOID foo (f)
float f;
{
.
}
```

The above will produce an error and needs to be replaced by the newer style, as shown below.

```
//Prototype
VOID foo (float f);
.
VOID foo (float f)
{
.
}
```

Variable Name Length

The maximum number of characters used in a variable name is 64 characters. After the first 64 characters, the internal variable name is truncated.

Debugging

The linker does not generate any information of static variables declared inside a function. This is because a static variable defined may be declared in more than one function. Thus, the compiler simply generates an internal label for it and allocates space for it.

Assembling Compiler Output Considerations

DO NOT use the `-g` option when assembling the output from the compiler. The `-g` option is only to be used when you want debug information from an assembly language file. Using the `-g` option to assemble the output from the compiler will cause an error in the symbol file.

The following example will generate an error in the generated symbol file:

```
WDC816CC.EXE -at -ms -mu -mv -bs -sop -wl -wr -wd
WDC816AS.EXE -g -l -DDEBUG
```



Volatile Qualifiers

Please note that local variables CANNOT be declared as volatile inside a function body. To declare a variable as volatile inside a function body, it must be static. This is a limitation of the compiler.

Declaring a variable as volatile OUTSIDE a function body, (global scope):

```
volatile unsigned char    char_volatile;
```

Declaring a variable as volatile INSIDE a function body, (static local variable):

```
static volatile unsigned char    char_volatile
```

The following WILL NOT work when declared INSIDE a function body, (local variable):

```
volatile unsigned char    char_volatile
```

Path size Limitation

The maximum allowable length of the path that follows the command WDC02CC is 256 characters.

Global Function and Variable Definition

The C standard states that you cannot have a global variable and a function declared with the same name. If you do this, you will get the following error: Error 90: Symbol redeclared. The following is invalid code:

```
Unsigned char error;
```

```
Void error()
```

```
{  
}
```

Factorization of Source Code

When factorization is used, the Optimizer looks for repeated patterns in the compiler output. These patterns are replaced by a call to a subroutine, thus reducing the overall code size. It should be noted that while factorization may reduce the code size, cycle count may increase. Therefore it is up to the developer to decide which is more important for their application, speed or code size, and enable the factorization accordingly.

To enable factorization, you must include the following options on the command line: **-SX AND -SP**.

For example: WDC02CC -lt -bs -ms -sp -sx filename.c

will enable factorization on the compiler output of the source filename.c. It may be desirable to enable all optimizations for the source, thus you can specify **-SOPX** on the command line. Please see Chapter 2, Option Descriptions, for more information on the compiler options.



CHAPTER 4 Libraries

This CHAPTER describes the standard library functions provided with the WDC 65C02 C Development System.

Library Names

There are two main libraries provided with the WDC 65C02 C Development System. The *C.LIB* contains all the standard C functions and some special functions need by compiler generated code for operations like multiply and divide. The *M.LIB* contains the standard floating point math functions and floating point emulation routines. If the **WDC_LIB** environment variable has been set to point to the directory containing the library files, then the **-L** option can be used to access the libraries when linking. For example, the command:

```
WDCLN PROG.OBJ -LC
```

will look for functions in the library *C.LIB*. When linking with the floating point library, be sure to specify it before the standard library in the linker command line. This will insure that the proper version of *printf* and *scanf* are used. If floating point numbers do not print out at all, check to see if the libraries are in the proper order.

Example:

```
WDCLN PROG.OBJ -LM -LC
```

will look for functions in the libraries *M.LIB* and *C.LIB*.

NOTE: If the floating point library is not needed, do not link it as this will result in larger code size!

ANSI Functions

For a list of the standard ANSI functions, in the form of function prototypes, that are supported by the W65cSDS, please refer to Appendix A.

Heap Functions

The ANSI standard library includes functions for allocating, freeing and reallocating memory from a memory heap. The WDC C Development System supports a heap built on top of the *sbrk()* function. This function allocates memory from a heap area that is specified by the programmer. The beginning of the heap is specified by *declaring* and initializing the two variables, *heap_start* and *heap_end*. The *sbrk()* function begins by allocating memory starting at *heap_start* and will continue to allocate until *heap_end* is reached.

For example:

```
void *heap_start = (void *)0xa000, *heap_end = (void *)0xd000;
```

This example allocates 0x3000 bytes of space starting at 0xa000. The ANSI functions that use the heap are *malloc*, *calloc*, *realloc*, and *free*. The *sbrk* function is not an ANSI function.

To have the heap start at the end of the “Uninitialized Data” area, use the following statements:

```
extern char    _END_UDATA;  
void *heap_start = ( void * )&_END_UDATA, *heap_end = ( void * )( &_END_UDATA + nnn );
```



THIS PAGE LEFT INTENTIONALLY BLANK



APPENDIX A WDC Supported C Functions

The following is a list of the standard ANSI functions in the form of function prototypes. These functions are listed by the header file in which they are defined.

Assert.h

assert - Debugging macro
void _assert(char *, char *, unsigned int);

Ctype.h

Character tests and conversions, some are also macros

int isalnum(int _c); - test for letter or digit
int isalpha(int _c); - test for alphabetic
int isascii(int _c); - test for < 0x80
int iscntrl(int _c); - test for control character
int isdigit(int _c); - test for digit
int isgraph(int _c); - test for non-printable chars
int islower(int _c); - test for lower case letter
int isprint(int _c); - test for printable chars
int ispunct(int _c); - test for punctuation character
int isspace(int _c); - test for space (0x09 to 0x0d or 0x20)
int isupper(int _c); - test for upper case letter
int isxdigit(int _c); - test for hex digit (0..9 or a..f or A..F)
int toupper(int _c); - convert to upper case (if a letter)
int tolower(int _c); - convert to upper case (if a letter)
int toascii(int _c); - convert to ascii (remove sign)

Errno.h

Error code definitions and table returned by the errno() function.

extern char *sys_errlist[]; - pointer to system standard error list
extern int sys_nerr; - variable of system standard errors

Fcntl.h Low Level I/O

int creat(const char *_name, int _mode);
int open(const char *_name, int _mode);
int close(int);
size_t read(int, void *, size_t);
size_t write(int, void *, size_t);
long lseek(int, long, int);
int unlink(const char *);
void _exit(int _code);
void _abort(void);

Float.h

Floating point definitions for values of constants. (Ranges of max/min & standard floating point values for math symbols.)

Io.h

Not Defined by WDC.



Limits.h

Limits that apply to variable "types". i.e. unsigned short variable value range.

Locale.h

How Time, Date and Currency are formatted in other countries

Setlocale - selects the appropriate programs locale LC_ALL, etc.

```
char *setlocale(int _category, const char *_locale);
```

NON-ANSI

struct lconv *localeconv(void); - sets the components of an object with type struct lconv with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

Math.h

Math Functions

double acos(double); - inverse cosine

double asin(double); - inverse sine

double atan2(double, double); - inverse tangent of y/x

double atan(double); - inverse tangent

double atof(const char *); - ASCII to binary Floating point equivalent

double ceil(double); - smallest integer \geq argument (as double)

double cos(double); - cosine

double cosh(double); - hyperbolic cosine

double cotan(double); - cotangent

double exp(double); - exponential

double fabs(double); - double floating point to absolute (Assembly Language)

double floor(double); - largest integer \leq argument

double fmod(double, double); - floating modulus

double frexp(double, int *); - returns mantissa of floating point number as a normalized fraction in the range [0.5,1.0] (Assembly Language)

double ldexp(double, int); - calculates X times 2 raised to the exp power

double log10(double); - base ten logarithm

double log(double); - natural logarithm

double modf(double, double *); - return integer and fractional parts of number

double pow(double, double); - x raised to the power of y

power - See pow. (These are equated, therefore they are interchangeable)

double sin(double); - sine

double sinh(double); - hyperbolic sine

double sqrt(double); - square root

double tan(double); - tangent

double tanh(double); - hyperbolic tangent

NON-ANSI

double ran(void); - REAL pseudo-random number generation uniformly distributed on (0.0 to 1.0) [This program was written by W. J. Cody, Jr., as part of the ELEFUNT test package.]

double randl(double); - REAL pseudo-random number logarithmically distributed on (1,exp(x)). [This program was written by W. J. Cody, Jr., as part of the ELEFUNT test package.]

void sran(long); - seed for ran and randl



Setjmp.h (Not Yet Implemented)

Jump functions

```
int setjmp(jmp_buf _env); - save context
void longjmp(jmp_buf _env, int _val); - jump to location saved in setjmp
```

Signal.h (Not Yet Implemented)

Defines the conditions for error traps.

```
void (*signal(int _sig, void (*_func)(int)))(int); - make function a signal handler
int raise(int _sig); - generate signal
```

Stdarg.h

Defines for using variable number of arguments for functions

```
va_arg(char *ap, type) where type is int, char, long, float, etc... - Variable number of Arguments
va_end(char *ap) - Variable number of Arguments End
va_start(char *ap, char *parmN) - Variable number of Arguments Start of Definitions
```

Stddef.h

Many common constants, identifiers, typedefs, and variables.

Stdio.h

Higher level I/O Functions.

```
void clearerr(FILE *_stream); - reset error conditions on a file
int fclose(FILE *_stream); - close a file (file pointer)
int feof(FILE *_stream); - check for end of file (file pointer)
int ferror(FILE *_stream); - return error status for a file
int fflush(FILE *_stream); - write file buffers to a file
int fgetc(FILE *_stream); - read a single character from a file
int fgetpos(FILE *_stream, fpos_t *_pos); - return current file position
char *fgets(char *_s, int _n, FILE *_stream); - read string from selected file
FILE *fopen(const char *_filename, const char *_mode); - open a file
int fprintf(FILE *_stream, const char *_format, ...); - write a formatted line to a file
int fputc(int _c, FILE *_stream); - write a single character to a file
int fputs(const char *_s, FILE *_stream); - write string to selected file
size_t fread(void *_ptr, size_t _size, size_t _nmemb, FILE *_stream); - read from file using file pointer
FILE *freopen(const char *_filename, const char *_mode, FILE *_stream); - reopen a file
int fscanf(FILE *_stream, const char *_format, ...); - get a formatted line from a file
int fseek(FILE *_stream, long int _offset, int _whence); - seek to position using file pointer
int fsetpos(FILE *_stream, const fpos_t *_pos); - position file to a location previously returned by fgetpos
long int ftell(FILE *_stream); - return current file position
size_t fwrite(const void *_ptr, size_t _size, size_t _nmemb, FILE *_stream); - write to a file with item size & item count
int getc(FILE *_stream); - get character from a file
int getchar(void); - get character from stdin (also a macro in stdio.h)
char *gets(char *_s); - read string from stdin
void perror(const char *_s); - maps the error number in the integer expression errno to an error message
int printf(const char *_format, ...); - write a formatted line to a stream
int putc(int _c, FILE *_stream); - write a character to a file
int putchar(int _c); - write a character to stdout (also a macro in stdio.h)
int puts(const char *_s); - write string to stdin
```



The Western Design Center, Inc.

September 2005

W65C02S C Compiler/Optimizer v3.49

`int remove(const char *_filename);` - causes the file whose name is the string pointed to by filename to be no longer accessible by that name

`int rename(const char *_old, const char *_new);` - causes the file whose name is the string pointed to by old to be henceforth known by the name given by the string pointed to by new

`void rewind(FILE *_stream);` - position a file to the beginning

`int scanf(const char *_format, ...);` - get a formatted line from a stream

`void setbuf(FILE *_stream, char *_buf);` - set the address of the buffer for a file

`int setvbuf(FILE *_stream, char *_buf, int _mode, size_t _size);` - set the address and size of the buffer for a file

`int sprintf(char *_s, const char *_format, ...);` - equivalent to `fprintf`, except that the arguments specifies an array into which the generated output is to be written

`int sscanf(const char *_s, const char *_format, ...);` - equivalent to `fscanf`, except that the argument specifies a string from which the input is to be obtained, rather than from a stream

`FILE *tmpfile(void);` - creates a temporary binary file that will automatically be removed when it is closed or at program termination

`char *tmpnam(char *_s);` generates a string that is a valid file name and is not the same as the name of an existing file

`int ungetc(int _c, FILE *_stream);` - reverse of `getc`

`int vfprintf(FILE *_stream, const char *_format, char *_arg);` - equivalent to `fprintf`, with the variable argument list replaced by arg, which has been initialized by the `va_start` macro

`int vprintf(const char *_format, char *_arg);` - equivalent to `fprintf`, with the variable argument list replaced by arg

`int vsprintf(char *_s, const char *_format, char *_arg);` `vsprintf` - equivalent to `sprintf`, with the variable argument list replaced by arg, which has been initialized by the `va_start` macro

NON-ANSI

`int _filbuf(FILE *);` - internal function

`int _flsbuf(FILE *, int);` - internal function flushes the specified stream

Stdlib.h

Commonly used Library Functions.

`void abort(void);` - abnormal termination

`int abs(int _j);` - absolute value of integer

`int atexit(void (*_func)(void));` - set function to call during exit

`double atof(const char *_nptr);` - convert ascii to floating point

`int atoi(const char *_nptr);` - convert a string to an integer

`long int atol(const char *_nptr);` - convert a string to a long integer

`void *bsearch(const void *_key, const void *_base, size_t _nmemb, size_t _size, int (*_compar)(const void *, const void *));` - performs a binary search on the elements of a sorted array in order to locate an element that contains a specific value

`void *calloc(size_t _nmemb, size_t _size);` - allocate with size, count

`div_t div(int _numer, int _denom);` - Divide and return both quotient and remainder

`void exit(int _status);` - normal exit & close files

`void free(void *_ptr);` - causes the space pointed to by ptr to be deallocated, that is, made available for further allocation

`void ftoa(double _val, char *_buf, int, int);` - convert floating point to ascii

`long int labs(long int _j);` - long absolute value

`ldiv_t ldiv(long int _numer, long int _denom);` - long divide and return both quotient and remainder

`void *malloc(size_t _size);` - allocate with block size

`int mblen(const char *_s, size_t _n);` - determines the number of bytes comprising the multibyte character pointed to by s

`size_t mbstowcs(wchar_t *_pwcs, const char *_s, size_t _n);` - converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by s into a sequence of corresponding codes and stores not more than n codes into the array pointed to by pwcs

`int mbtowc(wchar_t *_pwc, const char *_s, size_t _n);` - stores the code in the object pointed to by pwc

`void qsort(void *_base, size_t _nmemb, size_t _size, int (*_compar)(const void *, const void *));` - sorts an array of nmemb objects, the initial member of which is pointed to by base

`int rand(void);` - integer random numbers (0 – 32565)

`void *realloc(void *_ptr, size_t _size);` - expand memory block



`void srand(unsigned int _seed);` - seed integer random number generator
`long int strtol(const char *_nptr, char **_endptr, int _base);` - convert a string to a long integer
`unsigned long int strtoul(const char *_nptr, char **_endptr, int _base);` - convert a string to an unsigned long integer
`double strtod(const char *_nptr, char **_endptr);` - convert a string to a double
`int system(const char *_string);` - passes the string pointed to by string to the host environment to be executed by a "command processor" in an implementation- defined manner
`size_t wcstombs(char *_s, const wchar_t *_pwcs, size_t _n);` - converts a sequence of codes that correspond to multibyte characters from the array pointed to by pwcs into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by s
`int wctomb(char *_s, wchar_t _wchar);` - determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is wchar

NON-ANSI

`void far *faralloc(unsigned long _nmemb, unsigned long _size);` function allocates space for an array of nmemb objects, each of whose size is size. The space is initialized to all bits zero
`void farfree(void far *_ptr);` - space pointed to by ptr to be deallocated
`void far *farmalloc(unsigned long _size);` - allocates space for an object whose size is specified by size and whose value is indeterminate.
`void far *farrealloc(void far *_ptr, unsigned long _size);` - changes the size of the object pointed to by ptr to the size specified by size
`long double strtold(const char *_nptr, char **_endptr);` - converts ASCII string to Long Double representation

String.h

String conversion Functions and Memory Functions

`void *memchr(const void *_s, int _c, size_t _n);` - search memory for character
`int memcmp(const void *_s1, const void *_s2, size_t _n);` - compare memory
`void *memcpy(void *_dst, const void *_src, size_t _n);` - copy memory, byte access, allows overlap
`void *memmove(void *_dst, const void *_src, size_t _n);` - move memory, byte access, allows overlap
`void *memset(void *_s, int _c, size_t _n);` - fill a block of memory with a character
`char *strcat(char *_dst, const char *_src);` - string concatenate
`strchr` - search string for character
`int strcmp(const char *_s1, const char *_s2);` - compare strings
`char *strcpy(char *_dst, const char *_src);` - copy string
`size_t strcspn(const char *_s1, const char *_s2);` - search for character not in set
`char *strerror(int _errnum);` - maps the error number in errnum to an error message string
`size_t strlen(const char *_s);` - return length of a string
`char *strncat(char *_dst, const char *_src, size_t _n);` - string concatenate, check for length limit
`char *strchr(const char *_s, int _c);` - search string for last occurrence of character
`int strncmp(const char *_s1, const char *_s2, size_t _n);` - compare strings for limited length
`char *strncpy(char *_dst, const char *_src, size_t _n);` - copy string, limited length
`char *strpbrk(const char *_s1, const char *_s2);` - search for character in set
`char *strrchr(const char *_s, int _c);` - locate the last occurrence of a character in a string
`size_t strspn(const char *_s1, const char *_s2);` - search for character not in set
`char *strstr(const char *_s1, const char *_s2);` - search for one string in another
`char *strtok(char *_s1, const char *_s2);` - split string into tokens

NON-ANSI

`char *index(char *_s, int _c);` - search string for character
`void *memccpy(void *_dst, const void *_src, int _c, size_t _n);` - copy memory & stop on character match
`char *rindex(char *_s, int _c);` - search string for last occurrence of character
`int strcoll(const char *_s1, const char *_s2);` - string collation – See Locale
`char *strdup(char *_s);` - make copy of a string in the heap
`size_t strxfrm(char *_s1, const char *_s2, size_t _n);` - string transformation – See Locale
`void swapmem(void *_s1, void *_s2, size_t _n);` - Swap the blocks of memory addressed by s1 and s2.



Time.h

Time and date conversion functions

clock_t clock(void); - return execution time for current task
time_t mktime(struct tm *_timeptr); - convert time as a structure to seconds
time_t time(time_t *_timer); - get current time in seconds
char *asctime(const struct tm *_timeptr); - convert binary time to a character string
char *ctime(const time_t *_timer); - current time and date as a character string
struct tm *gmtime(const time_t *_timer); - convert time in seconds to structure (Greenwich)
struct tm *localtime(const time_t *_timer); - convert time in seconds to structure (local time zone)
size_t strftime(char *_s, size_t _maxsize, const char *_format, const struct tm *_timeptr); - ascii time/date according to format string
double difftime(time_t _time1, time_t _time2); - difference between to times

Zardos.h

Zardos defines. (Compiler version, structures, and source level debug information.)

Zpage.inc

Page Zero temporary memory allocation for Functions and Floating point variables for the W65c02 and W65c134.



APPENDIX B Description of Compiler Error Messages

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen.

3: unterminated string

All strings must begin and end with double quotes (“”). This message indicates that a double quote has remained unpaired.

4: argument type mismatch

This warning is given if the argument specified in a function call does not match that of the function’s prototype. Although the warning is given, the argument will be converted to the appropriate type before being passed. To avoid the warning, the argument can be preceded by a type cast to the appropriate type.

5: invalid type for function

Functions may be declared to return any scalar type as well as certain aggregate types such as structures. Functions are not allowed to return arrays. All definitions or declarations of a function or a function pointer that return an array will generate this error message. For example:

```
char (*f)0[];
```

6: inappropriate arguments

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to **int**, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all. No other inappropriate symbols should appear before the left (open) brace.

```
badfunction( arg1, arg2 )
shrt arg1;    // misspelled or invalid keyword
double arg2;
{             // function body
}

goodfunction( arg1, arg2 )
float arg1;
int arg2;     // this line is not required
{             // function body
}
```

7: bad declaration syntax



A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;      // correct
char c d;     // error 7
char *s1, *s2 // error 7 detected here
float k;
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a **#include**'d file will be detected back in the file being compiled or in another **#include** file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in typecast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * ( int number ); // incorrect usage
i = 3 * (( int )number ); // correct usage
```

9: invalid operand of & (address of)

This error is given if the program attempts to take the address of something that does not have an address associated with it.

```
#define FOUR 4
char *addr;

addr = &FOUR; // error 9, can't take address of a constant
```

10: array size must be positive integer

The dimension of an array must be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[ 0 ]; // meaningless
extern char goodarray[]; // good
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, **goodarray** is external. Function arguments should be declared with a null dimension:

```
func( s1, s2 )
char s1[], s2[];
{
...
}
```

11: obsolete



12: invalid pointer reference

This error message will occur if pointer indirection is attempted on a type which cannot physically represent a pointer value. The only types, other than pointers themselves, which can hold pointer values, are **int**, **short**, and **long** (as well as their unsigned counterparts). All other C types will generate this error. For example,

```
char c;  
  
*c = 5;
```

will generate this error.

13: obsolete

14: obsolete

15: storage class conflict

Only automatic variables and function parameters can be specified as **register**.

This error can be caused by declaring a **static register** variable. While structure members cannot be given a storage class at all, function arguments can be specified only as **register**.

A **register int i** declaration is not allowed outside a function – it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say **long int I**, and **unsigned int j**, it is meaningless to use **double int k** or **float char c**. In this respect, the compiler checks to make sure that **int**, **char**, **float** and **double** are used correctly.

<i>data type</i>	<i>interpretation</i>	<i>size(bytes)</i>
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
unsigned short	unsigned integer	2
long/long int	long integer	4
unsigned long/unsigned long int	unsigned long integer	4
float	floating point number	4
long float/double	double precision float	8

17: internal

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code.



18: data type conflict

This message indicates an error in the use of the **long** or **unsigned** data type. **long** can be applied as a qualifier to **int** and **float**. **unsigned** can be used with **char**, **short**, **int** and **long**.

```
long i;           // a long int
long float d;     // a double
unsigned u;       // an unsigned int
unsigned char c;
unsigned long l;
unsigned float f; // error 18
```

19: bad syntax

This error occurs if the **#line** preprocessor directive is followed by something other than a numeric constant or macro that expands to one.

```
#line 100 "filename" // correct
#line "filename"     // error 19
```

20: structure redeclaration

This message informs you that you have tried to redefine a **structure**.

21: missing }

The compiler requires a comma after each member in the list of fields for a structure initialization. After the last field, it expects a right (close) brace.

For example, this program fragment will generate error 21, since the initialization of the structure named **emily** does not have a closing brace:

```
struct john {
    int bone;
    char license[ 10 ];
} emily = {
    1,
    "23-4-1984";
```

22: syntax error in structure declaration

This error occurs in a structure declaration that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
struct           // error 22, missing left curly brace
    int a;
    long b;
}
```




23: syntax error in enum declaration

This error occurs in an **enum** specification that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
enum colors {  
}           // error 23, nothing in enumerator list
```

24: need right parenthesis or comma in arg list

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that **getchar** is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funcall( arg1, arg2 arg3 );
```

25: structure member name expected here

The symbol following the dot operator or the arrow must be valid. A valid name is a string of alphanumeric characters or underscores. It must begin with an alphabetic character (a letter of the alphabet or an underscore). In the last line of the following example, (**salary**) is not valid because '(' is not an alphanumeric character.

```
empptr = & anderson;  
empptr->salary = 12000;           // these three lines  
(*empptr).salary = 12000;       // are  
anderson.salary = 12000;         // equivalent  
empptr = &anderson.;            // error 25  
empptr- = 12000;                 // error 25  
anderson.(salary) = 12000;      // error 25
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the **-s** option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.



27: invalid typecast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure david { ... } amy;  
amy = ( struct david )( expression );    // error 27
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct david emily;  
struct david amy;  
  
emily = amy;
```

29: invalid use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand(&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

expression ? statement1 : statement2

It is not desirable to use **?:** for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

Error 31 is generated by an expression that attempts to call a data item. The following code will generate an error 31:

```
int a;  
a();
```

Error 31 is often caused by an expression that is missing an operator. For example, Error 31 will be generated if the expression **a * (b + c)** is coded as **a (b + c)**.

32: invalid pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be of the same size.



33: invalid type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct joey, alias;

a = -array;
b = -alice;
c = -function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of `sizeof(expression)` and the cast operator. Compare the accompanying examples:

```
struct lucille {
    int i;
} andrew;
typedef double bigfloat;
typedef struct lucille foo;

j = 4 * bigfloat f;           // error 35
k = &foo;                     // error 35
x = sizeof( bigfloat );
y = sizeof( foo);            // good
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as `andrew`). It is no more meaningful to take the address of a structure type than any other data type, as in `&int`.

36: obsolete

37: invalid or missing expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and `sizeof`.

38: obsolete



39: enum redeclaration

This error occurs when an **enum** identifier is used more than once in defining the value of enumeration constants.

```
enum states { NY, CA, PA };
enum states { IL, FL, NJ };           // error 39
```

40: internal error

41: initializer not a constant

In certain initializations, the expression to the right of the equal sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
    int i = 3;
    static int j = ( 2 + i );         // illegal
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
    struct {
        char array[ 1 ];
    } substruct;
} superstruct =

version 1:
{
    "aBDdefghij"
};

version 2:
{
    {
        { 'a', 'b', 'c', ..., 'i', 'j' }
    }
};
```

In **version 1**, the initializers are copied byte-for-byte onto the structure **superstruct**.

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[ 10 ] = "aBDdefghij";
```

This will generate **error 42** because the string constant on the right is null-terminated. The null terminator ('\0' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.



43: initialization of undefined structure

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct david { ... };  
struct dog david = { 1, 2, 3 };           // error 43
```

44: missing right paren in declaration

This error occurs in the declaration of a function pointer when the right parenthesis is left out.

```
int ( * fp );                             // error 44  
int ( * fp);                             // error 44
```

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate **error 46** at the end even though the human error probably occurred in the **while** loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the **while** loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()  
{  
    int i, j;  
    char array[ 80 ];  
  
    gets( array );  
    i = 0;  
    while ( array[ i ] ) {  
        putchar( array[ i ] );  
        i++;  
        for ( i=0; array[ i ]; i++ ) {  
            for ( j=i+1; array[ j ]; j++ ) {  
                printf( "elements %d and %d are ", i, j );  
                if ( array[ i ] == array[ j ] )  
                    printf( "the same\n" );  
                else  
                    printf( "different\n" );  
            }  
            putchar( "\n" );  
        }  
    }  
}
```



47: open failure on include file

When a file is **#included**, the compiler will look for it in a default area. This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in include statement, but this diminishes flexibility somewhat.

48: invalid symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumeric characters (alphabetic and numerals). The following symbols will produce this error code.

```
2nd_time,  
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
Int I, j, k, l;
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only lvalues are allowed to stand on the left-hand side of an assignment. For example:

```
Int num;  
Num = 7;
```

They are distinguished from rvalues, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An lvalue may be thought of as a bucket into which an value can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue, **y**, be assigned to another lvalue, **x**:

```
#define NUMBER 512  
  
x = y;  
1024 = z;           // wrong, rvalues are reversed  
NUMBER = x;       // wrong, NUMBER is an rvalue
```

Some operators which require lvalues as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;  
  
i = 3;  
j = &i;
```



52: obsolete

53: multiply defined label

On occasions when the **goto** statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: obsolete

55: missing quote

The compiler found a mismatched double quote (") in a **#define** preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"  
"this is a string with an embedded quote: \"". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a **#define** preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';           // c is initialized to a single quote
```

57: obsolete

58: invalid # encountered

The pound sign (#) begins each command for the preprocessor: **#include**, **#define**, **#if**, **#ifdef**, **#ifndef**, **#else**, **#endif**, **#asm**, **#endasm**, **#line** and **#undef**. These symbols are strictly defined.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of *identifier* with the *substitution text* that was specified by the **#define**.

This error code refers to the *substitution text* of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (\), for practical purposes the size of a macro has been limited to 255 characters.

60: loss of const/volatile info

This error occurs when passing the address of a variable that is declared as **const** and/or **volatile**.

```
extern const volatile int clock_time;  
  
set_time( &clock_time );           // error 60
```



61: reference to undefined structures

This message comes in two forms:

- 1) As a warning, due to referencing an undefined structure member.
- 2) As an error, when trying to obtain the size of an undefined structure.

```
a = sizeof( struct nodef );           // error 61 if nodef not defined
```

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()  
{  
    return 1;  
}
```

This error can be caused by an error inside a function declaration list, as in:

```
func( a, b )  
int a; chr b;  
{  
    ...  
}
```

63: undefined label

A **goto** statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to go to a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding **goto**, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function( string )  
char *string;  
{  
    char *func1();           // correct  
    double func2(x, y );    // wrong;  
    ...  
}
```

In this example, **function()** is being defined, but **func1()** and **func2()** are being declared.

65: invalid function argument

This error occurs in a function definition that contains an argument that is not a valid identifier.

```
sub( a, 2b ) {           // error 65 because identifiers can't begin with a  
                        // numeric character
```


66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An **else** was found which is not associated with an **if** statement. **Else** is bound to the nearest **if** at its own level of nesting. So **if-else** pairings are determined by their relative placement in the code and their grouping by braces.

```

If ( ... ) {
    ...
    if ( ... ) {
        ...
    } else if ( ... )
        ...
    } else {
        ...
    }
}

```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the **if** and **else-if** means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding **if** statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the **else** statement. As shown here, the **else** is paired with the first **if**, not the second.

68: bad statement syntax

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as **char** or **int** must not lead a statement; compare the use of the casting operator:

```

func()
{
    int i;
    char array[ 12 ];
    float k = 2.03;

    i = 0;
    int m;                // error 68
    j = i + 5;
    i = ( int )k;        // correct
    if ( i ) {
        int i = 3;
        j = i;
        printf( "%d", i );
    }
    printf( "%d%d\n", i, j );
}

```

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is similar to error code 7. It will remain undetected until the following line and is often spuriously caused by a previous error.



70: goto needs a label

Compare your use of **goto** with this example. This message says that you did not specify where you wanted to **goto** with **label**:

```
        goto label;
        ...
label:
        ...
```

It is not possible to **goto** just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a **do-while** may consist of one statement or several statements enclosed in braces. A **while** conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, do not forget the **while** conditional.

72: statement syntax error in for

This error occurs when the first of the two semicolons that separate the three expressions found in a **for** loop condition are missing.

```
        for ( i=0 i; i++ ) {           // error 72 due to missing semicolon
```

73: statement syntax error in for body

This error occurs when the second of the two semicolons that separate the three expressions found in a **for** loop condition is missing.

```
        for ( i=0; i i++ ) {           // error 73 due to missing semicolon
```

74: expression must be integer constant

This error occurs when a variable occurs instead of an integer constant in declaring the size of an array, initializing an element in an **enum** list, or specifying a **case** constant for a **switch**.

75: missing colon on case

This should be straightforward. If the compiler accepts a **case** value, a colon should follow it. A semi-colon must not be accidentally entered in its place.

76: obsolete

77: case outside of switch

The keyword, **case**, belongs to just one syntactic structure, the **switch**. If **case** appears outside the braces which contain a **switch** statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon on default

This message indicates that a colon is missing after the keyword, **default**. Compare with error 75.



79: duplicate default

The compiler has found more than one **default** in a **switch**. **switch** will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a **case** statement. It is analogous to the **else** companion to the conditional, **if**. Just as there is one **else** for every **if**, only one default case is allowed in a **switch** statement. However, unlike the **else** statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, **default**, is used just like **case**. It must appear with the brackets which delimit the **switch** statement.

81: break/continue error

break and **continue** are used to skip the remainder of a loop in order to exit or repeat the loop. **break** will also end a **switch** statement. But when the keywords, **break** or **continue**, are used outside of these contexts, this message results.

82: obsolete

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than ten levels into a nest.

84: constant expression expected

This error occurs when an integer constant is missing, such as in initializing an element in an **enum** list, specifying a **case** constant for a **switch**, or for a **#if** preprocessor directive.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an **extern** declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];           // correct
extern char badarray[5][];        // wrong
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as **'a'** or **'ab'**. There is no analog to a null string, so **''** (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (**\b**, **\n**, **\t**, etc.) are singular, so that the following are valid: **'\n'**, **'\na'**, **'a\n'**. **'aaa'** is invalid.



88: not a structure

Occurs only under compilation without the `-s` option. A name used as a structure does not refer to a structure, but to some other data type:

```
int i;  
i.member = 3;           // error 88
```

89: invalid use of register storage class

A globally defined variable cannot be specified as a register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: invalid use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: invalid type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;  
float j;  
char *ptr;  
  
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type **char** and **short** become **int**, and **float** becomes **double**. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a **float** will evaluate to a **double**.

Types have the following hierarchy:

```
double float  
long  
unsigned  
int, short, char
```

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()  
{  
    struct tag sam;  
    return sam;  
}
```



93: invalid expression type for switch

Only a **char**, **int** or **unsigned** variable can be switched. See the example for error 74.

94: invalid identifier in macro definition

This error occurs in a macro definition that contains one or more arguments that are not valid *identifiers*.

```
#define add( a, 2b ) ( a+2b )           // error 94 because identifiers can't  
                                       // begin with a numeric character
```

95: obsolete

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a “double comma” will produce this error.

```
#define reverse( x, y, z ) ( z, y, x )  
  
func( reverse( i, , k ) );
```

97: too many arguments in macro definition

This error occurs in a macro definition that contains more than 32 arguments in its definition.

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define exchange( x, y ) ( y, x )  
  
func( exchange( i ) );           // error 98
```

99: internal error

100: internal error

101: missing close parenthesis on macro reference

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's argument is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between **#if** and **#else** is analogous to that which exists between the control flow statements, **if** and **else**. Obviously, much depends upon the relative placement of the statements in the code. However, **#if** blocks must always be terminated by **#endif**, and the **#else** statement must be included in the block of the **#if** with which it is associated. For example:



```
#if ERROR 0
    printf( "there was an error\n" );
#else
    printf( "no error this time\n" );
#endif
```

#if statements can be nested, as below. The range of each **#if** is determined by a **#endif**. This also excludes **#else** from **#if** blocks to which it does not belong:

```
#ifdef JAN1
    printf( "happy new year!\n" );
    #if sick
        printf( "I think I'll go home now\n" );
    #else
        printf( "I think I'll have another\n" );
    #endif
    #else
        printf( "I wonder what day it is\n" );
    #endif
```

If the first **#endif** was missing, error 103 would result. And without the second **#endif**, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest **#if**, **#ifdef** or **#ifndef** which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated **#asm**. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a **#endasm** without having found a previous **#asm**. If the **#asm** was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the **#asm** keyword must not appear between a paired **#asm/#endasm**. When a piece of in-line assembly is augmented for temporary purposes, the old **#asm** and **#endasm** can be enclosed in comments as place-holders.

```
#asm
    // temporary asm code
    // #asm old beginning
    // more asm code
#endasm
```

107: missing #endif

A **#endif** is required for every **#if**, **#ifdef** and **#ifndef**, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first **#endif**. Backtrack to the previous **#if** and form the pair. Assign the next **#endif** with the nearest unpaired **#if**. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a **#endasm** in all cases. **#asm** must always be paired with a **#endasm**.



109: obsolete

110: invalid use of : operator

The colon operator occurs in two places:

- 1) following a question mark as part of a conditional, as in (**flag ? 1 : 0**).
- 2) Following a label inserted by the programmer or following one of the reserved labels, **case** and **default**.

111: invalid use of a void expression

This error can be caused by assigning a **void** expression to a variable, as in this example:

```
void func();  
int h;  
  
h = func();
```

112: obsolete

113: duplicate case in switch

A **switch** statement has two **case** values which are the same. Either the two cases must be combined into one, or one must be discarded. For instance:

```
switch( c ) {  
case NOOP:  
    return 0;  
case MULT:  
    return x * y;  
case DIV:  
    return x / y;  
case NOOP:  
default:  
    return 1;  
}
```

The **case** of **NOOP** is duplicated and will generate an error.

114: macro redefined

For example,

```
#define islow( n ) ( n >= 0 && n < 5 )  
...  
#define islow( n ) ( n >= 0 && n <= 5 )
```

The macro, **islow**, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

If the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:



```
#define islow( n ) n > 0 && n <= 5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define NULL 0  
...  
#define NULL 0
```

But these are different from:

```
#define NULL '\0'
```

In practice, this error message does not affect the compilation of the source code. The most recent “revision” of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define int foo
```

If you have a variable which may be either, for instance, a **short** or a **long** integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef LONGING  
    long i;  
#else  
    short i;  
#endif
```

Another possibility is through a typedef:

```
#ifdef LONGINT  
    typedef long VARTYPE;  
#else  
    typedef short VARTYPE;  
#endif  
VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can not have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can not have zero bits.

118: field is too wide

A field in a bit field structure can not have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.



120: invalid type for field

The type of a bit field can only be of type **int** or **unsigned int**.

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to **int** or **long**, or vice versa.

If the program explicitly casts a pointer to an **int**, this message will not be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;

...
i = cp;          // implicit conversion of char to int
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an **int**, and the sizes of the two items differ, the compiler will issue this warning message. The code that is generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an **int**.

123: far/huge ptr & ptr not same size

This error occurs when trying to assign a near pointer to a **far** or **huge** pointer. A warning is generated when casting a **far** or **huge** pointer to a near pointer.

124: invalid ptr/ptr expression

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an **int** are the same, the generated code will access the correct memory location, but if they are not, it will not.

For example:

```
char c;
long g;

*0x5c = 0;      // warning 125, because 0x5c is an int
c[ i ] = 0;     // warning 125, because c+i is an int
g[ i ] = 0;     // error 12, because g+i is a long
```



126: too many arguments

This error occurs when a function is invoked with more arguments than is specified in its prototype or definition. The only exception allowed is when a variable number of arguments is specified in the prototype.

127: too few arguments

This error occurs when a function is invoked with less arguments than is specified in its prototype or definition.

128: #error

This error was generated by the **#error** directive and is followed by the optional sequence of preprocessing tokens found in the source code.

129: #elif with no #if

This error occurs when the **#elif** preprocessor directive is used without a preceding **#if** directive.

130: obsolete

131: ## at the beginning/end of macro body

This error occurs when a **##** is found as the first or last end of a macro definition body.

```
#define TWOSHARP 2##           // error 131
```

132: obsolete

133: # not followed by a parameter

This error may occur in a **#define** macro in which the **#** operator is applied to a parameter in the replacement list. If the **#** token is not followed by a parameter, this error message will be generated.

134: obsolete

135: attempt to undefined a predefined macro

This error occurs when attempting to use a **#undef** on those macros that are predefined by the compiler, such as **__STDC__**, **__TIME__**, **__DATE__**, **__FILE__**, **__LINE__**, and **__FUNC__**.

```
#undef __TIME__           // error 135
```

136: invalid #include directive

This error occurs when the **#include** directive is not followed by a string literal or a *filename* enclosed in **< >** signs.

```
#include filename           // error 136
```

137: obsolete

138: missing right paren

This error occurs when attempting to use the defined directive with a left parenthesis and no matching right parenthesis.



```
#if defined( FOOBAR          // error 138
```

139: missing identifier

This error occurs when attempting to use the defined directive with no identifier following the defined keyword.

```
#if defined          // error 139
```

140: obsolete

141: obsolete

142: range-modifier ignored

Using a range modifier (**near**, **far**, etc.) on a structure or union member is allowed by the parser but has no effect and is ignored.

143: range-modifier syntax error

A range-modifier is illegal as part of a function declaration. You cannot say that a function is near, far, huge, etc.

144: invalid operand for sizeof

This error occurs when attempting to obtain the **sizeof** of something other than a previously defined data structure.

```
Sub() {  
    Int I = sizeof( sub );          // error 144  
}
```

145: function called without prototype

This warning is generated for functions that are called without having been prototyped. It only occurs when compiling with the **-wp** option.

146: constant value too large

This error occurs when attempting to use a constant larger than the unsigned long **0xffffffff** in an expression.

147: invalid hexadecimal constant

This error occurs when the character following a 0x or 0X is not a valid hexadecimal constant.

```
int i = 0xg1;          // error 147, 'g' is not a valid hex constant
```

148: invalid floating constant

This error occurs if the first letter excluding the optional sign following the **e** or **E** in a floating point number is something other than a digit.

```
double d = 123e+f;    // error 148, 'f' is not a digit
```

149: invalid character on control line

This error occurs on conditional preprocessor lines that expect a single constant expression but get extra information.

```
#if CONST invalid    // error 149 due to extra characters "invalid"
```



150: unterminated comment

This error occurs if the start of a comment (*/**) is not terminated with (**/*) before the end of the file.

151: no block level extern initialization

This error occurs when initialization of an **extern** variable is attempted inside a function. Initialization of **externs** is permissible outside of functions, or the function can declare the variable as **extern** and then initialize it further down in the code using an assignment statement.

152: missing identifier in parameter list

This error occurs when the type of an argument is specified in a function definition without being followed by the argument itself.

```
sub( int ) {           // error 152, missing the name of the int arg
```

153: missing static function definition

This error occurs if a function has been declared as static in a file and has not been followed by its actual definition further down in the file.

154: function definition can't be via typedef

This error occurs when incorrectly defining a function using a **typedef**. It is possible to define a **typedef** that is a function such as:

```
typedef int F( void );
```

which sets the type **F** to be a function with no arguments returning **int**. Then, a function can be declared such as:

```
F f;
```

which *is* legal. However, the function definition:

```
F f ()
```

is illegal.

155: file must contain external definition

This error occurs in a file with no external data or function definitions when compiling with the **-pa** option to use the ANSI preprocessor.

156: wide string literal not allowed here

This error occurs when attempting to use a wide string literal with the **#include** or **#line** directives.

```
#include L"filename" // error 156
```

157: incompatible function declarations

This error occurs if a function declaration does not match a previous definition or declaration for the same function.



158: called function may not return incomplete type

This error occurs when a function attempts to return a structure which has not been defined. If a function is called that returns a structure, but the size of the structure is unknown, then it is not possible for the compiler to know how much data is being returned by the function and how much space to reserve for the return value.

For example:

```
Struct foo x();

Main()
{
    x();
}
```

159: syntax error in #pragma

This error occurs if the #pragma is used for a function call and does not match the following syntax.

```
#pragma regcall( [return=] func( arg1, arg2, ..., argn ) ]
#pragma amicall( base, offset, func( arg1, arg2, ..., argn))
#pragma libcall func base offset regmask
#pragma syscall func offset regmask
```

160: auto variable not used in function

This warning occurs when compiling with the **-wu** option and a function containing a local variable has not been used.

161: function defined without prototype

This warning occurs when compiling with the **-wp** option and a function does not have its arguments prototyped or if there are no arguments but you have not specified void.

162: can't take address of register class

This error occurs when attempting to take the address of a variable that has been declared as a register class variable.

```
register int a;
int *ip;

ip = &a; // error 162
```

163: upper bits of hex character constant ignored

This warning occurs if the compiler encounters a hexadecimal character constant (specified by \x) whose value cannot fit within a single byte. For example:

```
char *cptr = "\x9b7";
```

Will generate a warning because **"\x9b7"** cannot be stored within one byte. The compiler will ignore the most significant bits, and use the least significant bits. In the example the compiler will treat **"\x9b7"** as **"\xb7"**. This warning will occur if you accidentally place a digit from 0 through 9 or a letter from a through f immediately after a \x escape sequence. In the example if you intended to have 0x9b followed by the ASCII digit 7, you could use string concatenation to produce the desired result:

```
char *cptr = "\x9b" "7";
```



164: non-void type function must have return value

This error message can occur only if the `-wr` compiler option is used. If the compiler encounters a function which is defined as returning a value (`int`, `char`, or the like) but which does not have an explicit return. For example:

```
int func()
{
    printf( "hello\n" );
}
```

would generate this message. Replacing `int func()` with `func()` will not correct the error. The specification `void func()` will correct the problem. The specification of an explicit `return` will also.

165: item not previously declared found in prototype

This warning message will be generated if a `struct` appears as an argument in a function prototype and there is no previous declaration for the `struct`. This warning will be generated if there is no `struct` declaration or if the `struct` declaration occurs after the prototype statement. The sequence `struct` declaration, prototype statement, function definition will correct the problem as in this example:

```
struct astruct {
    int a;
    char c;
}

void func( struct astruct arg );

void func( struct astruct arg )
{
    ...
}
```

This problem presents special difficulties when it arises because the intended `struct` declaration occurs after the prototype definition. A strict interpretation of ANSI rules, in this case, produces results that may seem arbitrary and illogical. Positioning the `struct` declaration so that it occurs before the prototype definition corrects the problem. If the problem is not corrected, it is unlikely that the program will run correctly.

166: enum must be declared outside prototype

This problem is similar to error 165 but pertains to the `enum` datatype.

167: can't take address of stack in this memory model

This error occurs when an attempt is made to take the address of a stack item.

168: missing semicolon in asm block

When the `asm` keyword is used to declare a block of assembly language statements, each must be terminated with a semicolon.

169: can't convert far pointer to near

A far pointer cannot be converted to a near pointer because of the difference in the size of the pointers.



170: can't use TSB/TRB on volatile values

This warning message is generated to alert the developer of the inability to use TSB/TRB on volatile values.



THIS PAGE LEFT INTENTIONALLY BLANK



INDEX

#asm, 31
#else, 22
#endasm, 31
#include, 10
#pragma, 31

C_MACROS, 24
CCOPT6502, 11
CCTEMP, 10

endif, 22

I, 11
initialized data, 29

KDATA, 31

MO, 11

ORG, 16

P0X, 11
printf, 33
-PX, 11

-**QP** and -**QS** options, 23

scanf, 33
-SFMR, 24
-SP option, 33
STARTUP, 29
STARTUP.ASM, 29

uninitialized data, 29

Volatile Qualifiers, 36

WDC_LIB, 37
WDC_INC_6502, 10
WO, 11