

# Controlling Diagnostic Messages

---

This appendix describes the conditions under which the compiler is likely to produce diagnostic messages, shows what the messages look like, and presents ways to control message selection and output format. It contains the following sections:

§H.1: *Types of Diagnostic Messages*

§H.2: *File I/O Errors*

§H.3: *System Errors*

§H.4: *User Error and Warning Messages*

§H.5: *Changing the Message Format*

§H.6: *Controlling Warning Messages*

---

## H.1 Types of Diagnostic Messages

Diagnostic messages from the High C/C++ compiler report the following:

- file I/O errors
- system errors
- user errors and warnings

The following sections discuss these types.

---

## H.2 File I/O Errors

File I/O errors are fatal to your compilation. They can occur when you try to open a non-existent file, or when the compiler writes an output file and not enough disk space is available.

---

**WARNING:** Fatal errors can result in compiler temporary files being left on disk. These are files with a `.tmp` extension. You should remove them.

---

These are the I/O error messages you are most likely to encounter:

**`fff: file not found.`**

This message is produced when a source file specified on the command line cannot be found. In addition, when you run the compiler in ANSI mode, the files `hcansi.pt` and `hcansip.pt` must be accessible.

**`***Cannot write to intermediate file.`**

This message is usually caused by too little space on disk. Remove unnecessary disk files and try again.

---

## H.3 System Errors

System errors are fatal to your compilation; you should rarely encounter them. The error messages take this form:

```
>>>> S Y S T E M   E R R O R   n <<<<, in Module:Function
Error message text.
```

where the components of the message have the following meanings:

<i>n</i>	numbers the occurrences of system errors
<i>Module</i>	the module name
<i>Function</i>	the function name

These are the only system error messages that you should encounter:

**`Dynamic array allocation/reallocation failed. Out of memory.`**

This error indicates that there was too little memory to compile the program. This usually occurs when at least one function definition is pathologically large. Such functions should be broken up into smaller ones.

**`Recover: Exceeded the following limit: Limit.`**

A table overflowed while a syntax error was being repaired. The table limit is fixed, so no increase in memory can improve the situation. Repair the error.

Other system error messages are associated with internal compiler errors or inconsistencies that normally will not occur. If they do occur, send a test case to your technical-support provider.

---

## H.4 User Error and Warning Messages

User error messages are grouped in three categories:

- lexical error messages
- syntactic error messages
- constraint error and warning messages

Error messages begin with “E” and warnings with “w”. Warnings do not ordinarily suppress object-file generation; errors always do. However, some warnings become errors when you run the compiler in ANSI mode, and you can change warnings to errors with `pragma Onwarn_error` (see §H.6.2: *Controlling Individual Warnings by Number*).

Messages that report errors terminate compilation after the phase issuing the diagnostic, so errors that would otherwise have been detected by later phases are not reported until you have repaired all earlier errors and reinvoke the compiler.

*Format of diagnostic messages* All user diagnostics are accompanied by the file name, a line number *n*, a column number *m*, and (for front-end messages) a message number *xxx*, in the form "*filename*", Ln/Cm/#*xxx*. Line and column numbers show where in your source the compiler detected the error was detected.

For lexical and syntactic errors, the erroneous line is displayed with a caret “^” beneath it at the point of error detection.

---

### H.4.1 Lexical Error Messages

Lexical error messages are produced when an improperly formed word is detected, such as a string with a missing closing quote.

```

Example: String constant not terminated
Levels  LINE # |-----1-----2-----3-----4-----
          1 | void main() {
          2 |     char *S;
          3 |     S = "Hello;
          C16 |-----^
E "ex1.c",L3/C16(#485):(lexical) String constant not terminated

```

---

## H.4.2 Syntactic Error Messages

Syntactic error messages are produced for programs that are ill-formed at the phrase level, such as a missing “;” or inserted spurious symbol. The message is accompanied by a statement of the REPAIR that the compiler performed so it could keep processing input.

```

Example: Unexpected symbol
Levels  LINE # |-----1-----2-----3-----4-----
          1 | void main() {
          2 |     printf "Hello");
          C12 |-----^
          1 |     }
E "ex2.c",L2/C12:(syntactic) unexpected symbol:'<STRING>':"Hel
REPAIR:  '(' was inserted before '<STRING>':"Hello"@ "ex2.c",L2
1 user error  No warnings
End of processing, Mon Feb 14 11:22:51 1994          ex2.c

```

---

## H.4.3 Constraint Error and Warning Messages

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are several hundred such diagnostics, each of which is meant to be self-explanatory. Most of them prevent the generation of object code, but some are merely warnings intended to assist the programmer.

```

Example: Identifier is undeclared
Levels  LINE # |-----1-----2-----3-----4-----
          1 | void main() {
          2 |     int i;
          3 |     i = Undeclared_identifier;
          4 |     }
E "ex3.c",L3/C9(#237):Undeclared_identifier: Identifier is unde
1 user error  No warnings
End of processing, Mon Feb 14 11:23:07 1994          ex3.c

```

```

Example: Variable Levels  LINE # |-----1-----2-----3-----4-----
is never referenced      1 |void main() {
                        2 |    int Unused;
                        3 |    }
No user errors      No warnings
End of processing, Mon Feb 14 11:23:44 1994      ex5.c
w "ex5.c",L2: Variable "Unused" is never referenced.
No errors      1 warning

```

---

## H.5 Changing the Message Format

You can modify the format of error messages and warning messages with command-line options **-Hefmat** and **-Hwfm**, respectively. These options direct the compiler to format error and warning messages according to a quoted string containing a sequence of characters with special meanings. (You can look at the `.cnf` file to view the actual text of the format directive.)

The formatting characters include the following:

- %c Column number
- %f File name
- %l Line number
- %m Comma: direct the compiler to include a comma
- %n Diagnostic message number
- %q Double quote: use to avoid quoting-convention problems of some shells
- %t Tab

*Microsoft C error-message format* To change the default MetaWare High C/C++ error message format, such as:

```
E "test.c",L1/C7/#237: J: Identifier is undeclared.
```

to match Microsoft C error messages, you can specify **-Hmscerr**, or modify the `ARGS` line in the `.cnf` file to include:

```
"-Hefmat=%f(%l): error C%n: "
```

which generates the Microsoft C error message format:

```
test.c(1): error C237: J: Identifier is undeclared.
```

*Microsoft C warning-message format* Similarly, to change the default MetaWare High C/C++ warning message format:

```
w "test.c",L2/C7/#291:  printf: Function called but
not defined.
```

to match Microsoft C warning messages, you can specify **-Hmscerr**, or modify the ARGs line in the .cnf file to include:

```
-Hwformat="%f(%l): warning C%n: "
```

which generates the Microsoft C warning message format:

```
test.c(2) : warning C291: printf: Function called
but not defined.
```

---

## H.5.1 Predefined Message Formats

You can specify several predefined formats by using special command-line options. Suppose this is the default High C/C++ format of an error message:

```
E "test.c",L1/C7/#237:  J: Identifier is undeclared.
```

This same message would appear as shown for each of the following predefined formats:

Format	Option	Message
Microsoft	<b>-Hmscerr</b>	test.c(1): error C237: J: Identifier is undeclared.
Borland C++	<b>-Hturboerr</b>	Error test.c 1: J: Identifier is undeclared.
UNIX	<b>-Hunixerr</b>	"test.c", line 1: J: Identifier is undeclared.

---

## H.6 Controlling Warning Messages

You can control the warning messages output by the compiler by level or by message number. Warning messages are output by default; you can turn them off with the `Msgno` toggle.

---

## H.6.1 Controlling Groups of Warnings by Level

The compiler supports five levels of warning diagnostics.

- Level 1 warnings are considered to be of the most interest to the programmer.
- Levels 2 and 3 warnings are of less interest.
- Level 4 warnings are informational.
- Warning level 0 suppresses all warnings.

You can specify the level of warning messages to display with command-line option `-wn` or with pragma `Warning_level(n)`. For a description of the `-wn` option, see Chapter 3: *Using Compiler Options*.

The `Warning_level` pragma has the following form:

```
#pragma Warning_level(n)
```

where *n* is a decimal integer constant denoting the level. Setting the warning level to *n* means that warning diagnostics with severity level *n* or lower are displayed. A setting of 0 (zero) suppresses all warnings. The default warning level is 3.

Pragma `Warning_level` is described in more detail in Chapter 5: *Using Compiler Pragmas*.

---

**Note:** Messages and classifications might change as the compiler is revised.

---

---

## H.6.2 Controlling Individual Warnings by Number

Warnings can be controlled individually by message number. These pragmas override the current level of warning messages set by pragma `Warning_level(n)` or command-line option `-wn`:

- `Onwarn`
- `Onwarn_error`
- `Offwarn`
- `Popwarn`

*Suppressing individual warnings* To suppress individual warnings, first find the warning numbers by compiling with toggle `Msgno On` (the default). Then use `pragma Offwarn` to specify the numbers of the warnings you wish to suppress. For example, this warning (#257):

```
'=' encountered where '==' may have been intended.
```

has been disabled in this code:

```
#pragma Offwarn(257)
... if (i = j) ...      /* Warning disabled */
#pragma Popwarn()
... if (j = k) ...      /* Warning restored */
```

*Restoring warnings* You can use `pragma Onwarn` to restore (re-enable) the warnings, although `pragma Popwarn` is preferable.

Use `pragma Onwarn_error` when you want High C/C++ to be more stringent than usual, but you do not want to use full ANSI mode.

*Changing warnings to errors* To change warnings to errors, find the warning numbers by compiling with toggle `Msgno On` (the default). Then use `pragma Onwarn_error` to specify the numbers of the warnings you want to promote to compiler errors.

For example, the compiler emits warning #60 when you use `return`; within a function that returns `int`. You can change this warning to an error, and cause the compilation to fail, by including this pragma:

```
#pragma Onwarn_error(60)
```

in your code.

Not all warnings are emitted at the point in the code where the offending text applies. For example, functions called but not declared are diagnosed at the end of compilation, so that only the last warning pragma has effect for all of them.

For example:

```
#pragma Offwarn(291)
... undef_func1() ... /* Warning not issued here */
#pragma Popwarn()
... undef_func2() ... /* Nor here */
/* Now all undefined function warnings are
   emitted, so that both lines are diagnosed */
```



If your code contains undeclared functions, it is best to disable warning number 291 for the entire compilation. You can use **-Hpragma** to specify this on the command line or on your ARGV line in the .cnf file, like this:

```
hc prog.c -Hpragma=Offwarn(291)
```

