A Guide to Common Compiler Errors and Warnings for C in Keil uVision

Message Format

The error/warning messages in Keil have the same format:

```
main.c(3): error:  #65: expected a ";"
      return 0;
main.c(2): warning:  #177-D: variable "x" was declared but never referenced
      int x
```

The file it occurs in
The line it occurs on
What type it is
What code it is (you won't use this)
The message
Relevant code snippet

If you double click the error/warning message, it will take you to the file and line where the problem is (super handy)!

Note that if you see a message that looks like this:

```
.\keil_test.axf: Error: L6200E: Symbol globalVar multiply defined (by var.o and main.o).
```

Then what you actually have is a *linker error*. The relevant parts are

The message
The files causing it (change the .o into .c and that's the source file that is causing the error)

Even though the linker is separate from the compiler, some linker errors will be discussed in this document. Sadly, double clicking linker errors will not take you to their location (which makes sense since the exact location is not specified).

Errors

```
expected a ";"
```

Anytime you see this, it means that you forgot a semicolon at the end of a line. Note that the compiler actually says the error is on the *next* line.

Keil marks the line that's actually missing the semicolon with an icon.

```
1  int main() {
2      int x = 10
3      int y = x + 5;
4      return 0;
5  }
6
```

Line 2 is missing a semicolon as indicated by the error icon, but the compiler will say the error is on line 3.

```
at end of source expected a "}"
```

This error is caused a missing closing brace somewhere in your source file. Unfortunately, the compiler will always say that it's expecting it at the end of the file, so you have to find out where the closing brace really belongs.

On top of all this you're gonna get a bunch of other compiler errors that seem random.

```
1  void foo() {
2      // Do some stuff...
3
4
5  int main() {
6      int x = 10;
7      int y = x + 5;
8      return 0;
9  }
10
```

Here foo is missing a closing brace at line 3. The compiler reports the error at line 9 (end of the file). It also reports an error at line 5.

```
expected a "{"
```

This error is caused by a missing opening brace somewhere in your source file. It's hard to tell exactly where the compiler will say the opening brace is expected, and just like before, you're gonna get a lot of weird compiler errors that shouldn't be there.
All the errors will be graphically marked, but that's not helpful...

```
1  void foo() {
2      // Do some stuff...
3  }
4
5  void bar()
6      foo();
7  }
8
9  int main() {
10     int x = 10;
11     int y = x + 5;
12     return 0;
13 }
14
```

Here bar is missing an opening brace at line 5. The compiler reports the error at line 7. It also reports an error at line 6.

`identifier is undefined`

This happens when you try to use a variable that you haven't declared.

This error is marked graphically.

```
1  int main() {
2      x + 4;
3      return 0;
4  }
5
```

x was used without declaring it first, giving an error. The line where the error occurs is marked with an icon.

`already declared in current scope`

This error occurs when you use the same name for a variable twice. The solution is simple, either remove the duplicate declaration, or (if it was meant to be a different variable), rename one of the variables to something else.

The error is marked graphically.

```
1  int main() {
2      int x;
3      int x;
4      return 0;
5  }
6
```

The name x was already used.

```
1  int main() {
2      int x1;
3      int x2;
4      return 0;
5  }
6
```

In this case we assumed we wanted to keep both variables, so we renamed them so there was no longer a conflict.

`undefined symbol`

This occurs whenever you try to call a function that was declared but not defined. This can happen for 2 reasons:

1. You called a function that you haven't defined and may or may not have declared
2. You wrote a header file that declared the function, included it, called the function, but didn't write a C file that defined it.
    - Note: you can define the function in *any* C file you want!

Note that this is actually a *linker error*, so Keil will tell you in what *file* the error occurs, but not what line. Keil also will not graphically mark where the error is either.

Scenario 1

```
1  void foo(void);
2
3  int main() {
4      foo();
5      return 0;
6  }
7
```

Here foo was only declared, not defined.

```
1  void foo(void);
2
3  void foo() {
4      // Do something...
5  }
6
7  int main() {
8      foo();
9      return 0;
10 }
11
```

We fixed the error by defining foo

Scenario 2

`foo.h`

```
1  void foo(void);
2
```

`main.c`

```c
#include "foo.h"

int main() {
    foo();
    return 0;
}
```

We included foo.h which declared foo but there is no definition for it.

Scenario 2, Solution 1

foo.c

```c
void foo() {
    // Do something...
}
```

Here we created a file called foo.c that defines foo

Scenario 2, Solution 2

main.c

```c
#include "foo.h"

void foo() {
    // Do something...
}

int main() {
    foo();
    return 0;
}
```

Here we defined foo in main.c. THERE IS NO foo.c IN THIS SOLUTION. YOU CAN ONLY DEFINE foo ONCE!

```
symbol multiply defined
```

This occurs when you define something more than once. You can only define something once across all source (.c) files. If you define the exact same function or global variable in two or more files, you'll get this error.

Note that this is actually a *linker error*, so Keil will tell you in what *file* the error occurs, but not what line. Keil also will not graphically mark where the error is either.

foo.c

```c
int globalVar;

void foo() {
}
```

main.c

```
1  int globalVar;
2
3  void foo() {
4  }
5
6  int main() {
7      return 0;
8  }
9
```

Here we defined foo and globalVar even though both were already defined in foo.c. Both give an error.

A Note about Header Files

If you define something in a header file, and then include that header file in multiple source files, you will get a multiple definition error. This is why things can only be *declared* in header files, never defined. To declare a global variable in a header file, add the extern keyword to the start.

var.h

```
1  int globalVar;
2
```

globalVar is defined in a header file.

foo.c

```
1  #include "var.h"
2
```

Here we include the var.h and in turn the definition of globalVar

main.c

```
1  #include "var.h"
2
3  int main() {
4      return 0;
5  }
6
```

By including var.h, globalVar becomes defined twice, once in foo.c and another in main.c.

Solution

var.h

```
1  extern int globalVar;
2
```

By adding the extern keyword, globalVar is now only declared in var.h

var.c

```
1  int globalVar;
2
```

globalVar is now defined in var.c.

`main.c` and `foo.c` can now include `var.h` safely, and then use `globalVar`.

Note that any file that wants to use globalVar must either include var.h *or* declare `globalVar` themselves using `extern int globalVar` (the extern keyword is crucial to avoid a multiple definition error)

`foo.c`

```
1  #include "var.h"
2
3  void foo() {
4      int x = globalVar + 3;
5  }
6
```

Here we included var.h and used globalVar

`main.c`

```
1  extern int globalVar;
2
3  int main() {
4      int copy = globalVar;
5      return 0;
6  }
7
```

Here we didn't include var.h, but we did declare globalVar, so we're using the same variable as foo.c but with no multiple definition error

---

Warnings

```
last line of file ends without a newline
```

Simply add a blank line to the end of the file as show below.

```
1  int main() {
2      return 0;
3  }
```

This gives a warning.

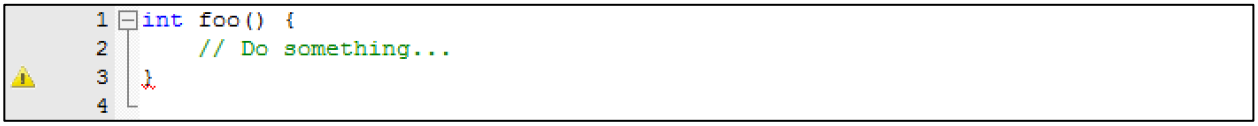
```
1  int main() {
2      return 0;
3  }
4
```

Fixed.

`missing return statement and end of non-void function`

This means you forgot to return a value at the end of a function. This seems like an obvious error but when doing long computations, it's not unheard of to forget to return the result!

Keil will graphically mark this.

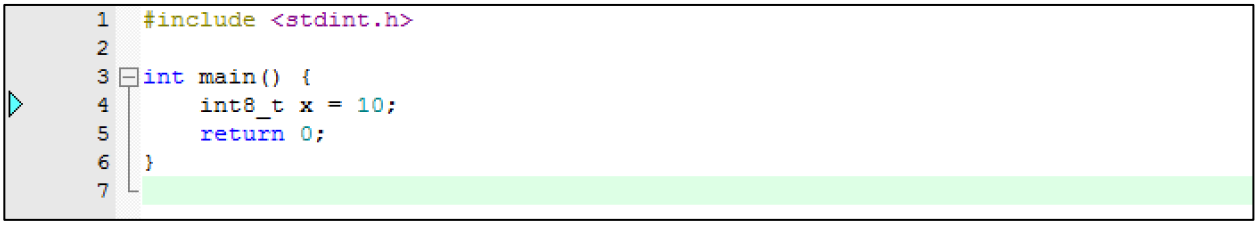


Small icon next to line number with the error and tooltip text that explains the warning.

`variable was declared but never referenced`

This means you declared a variable but you never used it outside the declaration. This error occurs whether or not you initialize the variable while declaring it.

If it turns out that you actually don't need to use this variable, you can safely delete it.



Variable x was declared (and initialized) but is not used. Note that no icon appears near the line number indicating the warning.

Note: declaring the variable as `volatile` also removes this warning (whether or not the variable is initialized). `volatile` tells the compiler that even though we don't use the variable, we want to allocate space for it.

`variable was set but never used`

This means you declared a variable, set its value, but never used it beyond setting its value.

Just like with the previous warning, if you don't need the variable, delete its declaration and all lines where you set it.

```
1  #include <stdint.h>
2
3  int main() {
4      unsigned int delay;
5      delay = 100;
6      return 0;
7  }
8
```

delay was declared (but not initialized) and then set. Note that no icon appears near the line number indicating the warning.

Note that to get rid of the warning we can also declare `delay` as `volatile`.

`function declared implicitly`

This error occurs when you try to call a function before it's been defined (if you've even defined it at all). The compiler says the warning is on the line where the function is *called*.

To get rid of this warning there are 2 solutions:

1. Move code around so that the function is defined before it's called
2. Leave the implementation where it is but *declare* the function before it's used.

```
1  void bar() {
2      int i = foo(4);
3  }
4
5  int foo(int x) {
6      return x * 2;
7  }
8
9  int main() {
10     return 0;
11 }
12
```

Here foo is being called before it's been defined. Note that the warning icon appears on line 2 where the function is called.

```
1  void bar() {
2      int x = foo(4);
3  }
4
5  int main() {
6      return 0;
7  }
8
```

Here foo wasn't even defined but we still get the same warning.

Solution 1

```
1  int foo(int x) {
2      return x * 2;
3  }
4
5  void bar() {
6      int i = foo(4);
7  }
8
9  int main() {
10     return 0;
11 }
12
```

The definition of foo was moved to before the line it was called.

Solution 2

```
1  int foo(int x);
2
3  void bar() {
4      int i = foo(4);
5  }
6
7  int foo(int x) {
8      return x * 2;
9  }
10
11 int main() {
12     return 0;
13 }
14
```

foo was declared before being called and defined afterwards. The compiler only needs to see a declaration so this is a perfectly ok solution.

```
deprecated declaration – give arg types
```

Basically, in Keil, when you declare a function with no parameters, you have to use void as the parameter

This only occurs when you declare a function. If you're defining it, then you can leave out the void parameter and you won't get a warning.

```
1  void foo();
2
3  void foo() {
4  }
5
6  int main() {
7      foo();
8      return 0;
9  }
10
```

foo is declared on line 1 and defined starting at line 3. The definition is fine, but the declaration gives a warning.

```
1  void foo(void);
2
3  void foo() {
4  }
5
6  int main() {
7      foo();
8      return 0;
9  }
10
```

The warning is fixed by saying that foo takes in void in the declaration. We could do the same for the definition, but we don't have to.