CC65 Users Guide

Usage

The compiler translates C files into files containing assembler code that may be translated by the ca65 macroassembler.

Command line option overview

The compiler may be called as follows:

```
---------------------------------------------------------------------------
Usage: cc65 [options] file
Short options:
  -A                      Strict ANSI mode
  -Cl                     Make local variables static
  -Dsym[=defn]            Define a symbol
  -I dir                  Set an include directory search path
  -O                      Optimize code
  -Oi                     Optimize code, inline more code
  -Or                     Enable register variables
  -Os                     Inline some known functions
  -T                      Include source as comment
  -V                      Print the compiler version number
  -W                      Suppress warnings
  -d                      Debug mode
  -g                      Add debug info to object file
  -h                      Help (this text)
  -j                      Default characters are signed
  -o name                 Name the output file
  -r                      Enable register variables
  -v                      Increase verbosity

Long options:
  --add-source            Include source as comment
  --ansi                  Strict ANSI mode
  --bss-name seg          Set the name of the BSS segment
  --check-stack           Generate stack overflow checks
  --code-name seg         Set the name of the CODE segment
  --codesize x            Accept larger code by factor x
  --create-dep            Create a make dependency file
  --data-name seg         Set the name of the DATA segment
  --debug                 Debug mode
  --debug-info            Add debug info to object file
  --help                  Help (this text)
  --include-dir dir       Set an include directory search path
  --register-space b      Set space available for register variables
```

--register-vars	Enable register variables
--rodata-name seg	Set the name of the RODATA segment
--signed-chars	Default characters are signed
--static-locals	Make local variables static
--target sys	Set the target system
--verbose	Increase verbosity
--version	Print the compiler version number

Command line options in detail

Here is a description of all the command line options:

-A, --ansi

This option disables any compiler exensions. Have a look at section 5 for a discussion of compiler extensions. In addition, the macro __STRICT_ANSI__ is defined, when using one of these options.

--bss-name seg

Set the name of the bss segment.

--check-stack

Tells the compiler to generate code that checks for stack overflows. See #pragma checkstack for an explanation of this feature.

--code-name seg

Set the name of the code segment.

--codesize x

This options allows finer control about speed vs. size decisions in the code generation phase. It gives the allowed size increase factor (in percent). The default is 100 when not using -Oi and 200 when using -Oi (-Oi is the same as --codesize 200).

--create-dep

Tells the compiler to generate a file containing the dependency list for the compiled module in makefile syntax. The file is named as the C input file with the extension replaced by .u.

-d, --debug

Enables debug mode, something that should not be needed for mere mortals:-)

-D sym[=definition]

Define a macro on the command line. If no definition is given, the macro is defined to the value "1".

-g, --debug-info

This will cause the compiler to insert a .DEBUGINFO command into the generated assembler code. This will cause the assembler to include all symbols in a special section in the object file.

-h, --help

Print the short option summary shown above.

-o name

Specify the name of the output file. If you don't specify a name, the name of the C input file is used, with the extension replaced by ".s".

-r, --register-vars

-r will make the compiler honor the register keyword. Local variables may be placed in registers (which are actually zero page locations). There is some overhead involved with register variables, since the old contents of the registers must be saved and restored. Since register variables are of limited use without the optimizer, there is also a combined switch: -Or will enable both, the optmizer and register variables.

For more information about register variables see register variables.

--register-space

This option takes a numeric parameter and is used to specify, how much zero page register space is available. Please note that just giving this option will not increase or decrease by itself, it will just tell the compiler about the available space. You will have to allocate that space yourself using an assembler module with the necessary allocations, and a linker configuration that matches the assembler module. The default value for this option is 6 (bytes).

If you don't know what all this means, please don't use this option.

--rodata-name seg

Set the name of the rodata segment (the segment used for readonly data).

-j, --signed-chars

Using this option, you can make the default characters signed. Since the 6502 has no provisions for sign extending characters (which is needed on almost any load operation), this will make the code larger and slower. A better way is to declare characters explicitly as "signed" if needed. You can also use #pragma signedchars for better control of this option.

-v, --verbose

Using this option, the compiler will be somewhat more verbose if errors or warnings are encountered.

-Cl, --static-locals

Use static storage for local variables instead of storage on the stack. Since the stack is emulated in software, this gives shorter and usually faster code, but the code is no longer reentrant. The difference between -Cl and declaring local variables as static yourself is, that initializer code is executed each time, the function is entered. So when using

```
void f (void)
{
    unsigned a = 1;
    ...
}
```

the variable a will always have the value 1 when entering the function and using -Cl, while in

```
void f (void)
{
    static unsigned a = 1;
    ....
}
```

the variable a will have the value 1 only the first time, the function is entered, and will keep the old value from one call of the function to the next.

You may also use #pragma staticlocals to change this setting in your sources.

-I dir, --include-dir dir

Set a directory where the compiler searches for include files. You may use this option multiple times to add more than one directory to the search list.

-O, -Oi, -Or, -Os

Enable an optimizer run over the produced code.

Using -Oi, the code generator will inline some code where otherwise a runtime functions would have been called, even if the generated code is larger. This will not only remove the overhead for a function call, but will make the code visible for the optimizer. -Oi is an alias for --codesize 200.

-Or will make the compiler honor the register keyword. Local variables may be placed in registers (which are actually zero page locations). There is some overhead involved with register variables, since the old contents of the registers must be saved and restored. In addition, the current implementation does not make good use of register variables, so using -Or may make your program even slower and larger. Use with care!

Using -Os will force the compiler to inline some known functions from the C library like strlen. Note: This has two consequences:

- You may not use names of standard C functions in your own code. If you do that, your program is not standard compliant anyway, but using -Os will actually break things.
- The inlined string and memory functions will not handle strings or memory areas larger than 255 bytes. Similar, the inlined is..() functions will not work with values outside char range.

It is possible to concatenate the modifiers for -O. For example, to enable register variables and inlining of known functions, you may use -Ors.

-T, --add-source

This include the source code as comments in the generated code. This is normally not needed.

-V, --version

Print the version number of the compiler. When submitting a bug report, please include the operating system you're using, and the compiler version.

-W

This option will suppress any warnings generated by the compiler. Since any source file may be written in a manner that it will not produce compiler warnings, using this option is usually not a good idea.

#pragmas

The compiler understands some pragmas that may be used to change code generation and other stuff.

#pragma bssseg (<name>)

This pragma changes the name used for the BSS segment (the BSS segment is used to store uninitialized data). The argument is a string enclosed in double quotes.

Note: The default linker configuration file does only map the standard segments. If you use other segments, you have to create a new linker configuration file.

Beware: The startup code will zero only the default BSS segment. If you use another BSS segment, you have to do that yourself, otherwise uninitialized variables do not have the value zero.

Example:

```
#pragma bssseg ("MyBSS")
```

#pragma charmap (<index>, <code>)

Each literal string and each literal character in the source is translated by use of a translation table. This translation table is preset when the compiler is started depending on the target system, for example to map ISO-8859-1 characters into PETSCII if the target is a commodore machine.

This pragma allows to change entries in the translation table, so the translation for individual characters, or even the complete table may be adjusted.

Both arguments are assumed to be unsigned characters with a valid range of 1-255.

Beware of two pitfalls:

- The character index is actually the code of the character in the C source, so character mappings do always depend on the source character set. This means that `#pragma charmap` is not portable - it depends on the build environment.
- While it is possible to use character literals as indices, the result may be somewhat unexpected, since character literals are itself translated. For this reason I would suggest to avoid character literals and use numeric character codes instead.

Example:

```
        /* Use a space wherever an 'a' occurs in ISO-8859-1 source */
        #pragma charmap (0x61, 0x20);
```

#pragma checkstack (on|off)

Tells the compiler to insert calls to a stack checking subroutine to detect stack overflows. The stack checking code will lead to somewhat larger and slower programs, so you may want to use this pragma when debugging your program and switch it off for the release version. If a stack overflow is detected, the program is aborted.

If the argument is "off", stack checks are disabled (the default), otherwise they're enabled.

#pragma codeseg (<name>)

This pragma changes the name used for the CODE segment (the CODE segment is used to store executable code). The argument is a string enclosed in double quotes.

Note: The default linker configuration file does only map the standard segments. If you use other segments, you have to create a new linker configuration file.

Example:

```
        #pragma codeseg ("MyCODE")
```

#pragma dataseg (<name>)

This pragma changes the name used for the DATA segment (the DATA segment is used to store initialized data). The argument is a string enclosed in double quotes.

Note: The default linker configuration file does only map the standard segments. If you use other segments, you have to create a new linker configuration file.

Example:

```
        #pragma dataseg ("MyDATA")
```

#pragma rodataseg (<name>)

This pragma changes the name used for the RODATA segment (the RODATA segment is used to store readonly data). The argument is a string enclosed in double quotes.

Note: The default linker configuration file does only map the standard segments. If you use other segments, you have to create a new linker configuration file.

Example:

```
        #pragma rodataseg ("MyRODATA")
```

#pragma regvaraddr (on|off)

The compiler does not allow to take the address of register variables. The regvaraddr pragma changes this. Taking the address of a register variable is allowed after using this pragma with "on" as argument. Using "off" as an argument switches back to the default behaviour.

Beware: The C standard does not allow taking the address of a variable declared as register. So your programs become non-portable if you use this pragma. In addition, your program may not work. This is usually the case if a subroutine is called with the address of a register variable, and this subroutine (or a subroutine called from there) uses itself register variables. So be careful with this #pragma.

Example:

```
        #pragma regvaraddr(1)   /* Allow taking the address
                                 * of register variables
                                 */
```

#pragma signedchars (on|off)

Changes the signedness of the default character type. If the argument is "on", default characters are signed, otherwise characters are unsigned. The compiler default is to make characters unsigned since this creates a lot better code. This default may be overridden by the --signed-chars command line option.

#pragma staticlocals (on|off)

Use variables in the bss segment instead of variables on the stack. This pragma changes the default set by the compiler option -Cl. If the argument is "on", local variables are allocated in the BSS segment, leading to shorter and in most cases faster, but non-reentrant code.

#pragma zpsym (<name>)

Tell the compiler that the – previously as external declared – symbol with the given name is a zero page symbol (usually from an assembler file). The compiler will create a matching import declaration for the assembler.

Example:

```
        extern int foo;
        #pragma zpsym ("foo");  /* foo is in the zeropage */
```

Inline assembler

The compiler allows to insert assembler statements into the output file. The syntax is

```
        asm (<string literal>[, optional parameters]) ;
```

or

```
        __asm__ (<string literal>[, optional parameters]) ;
```

The first form is in the user namespace and is disabled by -A.

The asm statement may be used inside a function and on global file level. An inline assembler statement is a primary expression, so it may also be used as part of an expression. Please note however that the result of an expression containing just an inline assembler statement is always of type void.

The contents of the string literal are preparsed by the compiler and inserted into the generated assembly output, so that the can be further processed by the backend and especially the optimizer. For this reason, the compiler does only allow regular 6502 opcodes to be used with the inline assembler. Pseudo instructions (like .import, .byte and so on) are *not* allowed, even if the ca65 assembler (which is used to translate the generated assembler code) would accept them. The builtin inline assembler is not a replacement for the full blown macro assembler which comes with the compiler.

Note: Inline assembler statements are subject to all optimizations done by the compiler. There is currently no way to protect an inline assembler statement from being moved or removed completely by the optimizer. If in doubt, check the generated assembler output, or disable optimizations.

The string literal may contain format specifiers from the following list. For each format specifier, an argument is expected which is inserted instead of the format specifier before passing the assembly code line to the backend.

- %b - Numerical 8 bit value
- %w - Numerical 16 bit value
- %l - Numerical 32 bit value
- %v - Assembler name of a (global) variable or function
- %o - Stack offset of a (local) variable
- %s - The argument is converted to a string
- %% - The % sign itself

Using these format specifiers, you can access C #defines, variables or similar stuff from the inline assembler. For example, to load the value of a C #define into the Y register, one would use

```
#define OFFS  23
__asm__ ("ldy #%b", OFFS);
```

Or, to access a struct member of a static variable:

```
typedef struct {
    unsigned char x;
    unsigned char y;
    unsigned char color;
} pixel_t;
static pixel_t pixel;
__asm__ ("ldy #%b", offsetof(pixel_t, color));
__asm__ ("lda %v,y", pixel);
```

Note: Do not embedd the assembler labels that are used as names of global variables or functions into your asm statements. Code like this

```
int foo;
int bar () { return 1; }
__asm__ ("lda _foo");    /* DON'T DO THAT! */
...
__asm__ ("jsr _bar");    /* DON'T DO THAT EITHER! */
```

may stop working if the way, the compiler generates these names is changed in a future version. Instead use the format specifiers from the table above.