Tiny C Compiler Reference Documentation

Table of Contents

Tiny C Compiler Reference Documentation

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

This manual documents version 0.9.27 of the Tiny C Compiler.

---

1 Introduction

TinyCC (aka TCC) is a small but hyper fast C compiler. Unlike other C compilers, it is meant to be self-relying: you do not need an external assembler or linker because TCC does that for you.

TCC compiles so *fast* that even for big projects `Makefile`s may not be necessary.

TCC not only supports ANSI C, but also most of the new ISO C99 standard and many GNUC extensions including inline assembly.

TCC can also be used to make *C scripts*, i.e. pieces of C source that you run as a Perl or Python script. Compilation is so fast that your script will be as fast as if it was an executable.

TCC can also automatically generate memory and bound checks (see Bounds) while allowing all C pointers operations. TCC can do these checks even if non patched libraries are used.

With `libtcc`, you can use TCC as a backend for dynamic code generation (see Libtcc).

TCC mainly supports the i386 target on Linux and Windows. There are alpha ports for the ARM (`arm-tcc`) and the TMS320C67xx targets (`c67-tcc`). More information about the ARM port is available at http://lists.gnu.org/archive/html/tinycc-devel/2003-10/msg00044.html.

For usage on Windows, see also tcc-win32.txt.

---

2 Command line invocation

2.1 Quick start

```
usage: tcc [options] [infile1 infile2…] [-run infile args…]
```

TCC options are a very much like gcc options. The main difference is that TCC can also execute directly the resulting program and give it runtime arguments.

Here are some examples to understand the logic:

`'tcc -run a.c'`

Compile `a.c` and execute it directly

`'tcc -run a.c arg1'`

Compile a.c and execute it directly. arg1 is given as first argument to the `main()` of a.c.

`'tcc a.c -run b.c arg1'`

Compile `a.c` and `b.c`, link them together and execute them. arg1 is given as first argument to the `main()` of the resulting program.

`'tcc -o myprog a.c b.c'`

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Compile `a.c` and `b.c`, link them and generate the executable `myprog`.

`'tcc -o myprog a.o b.o'`

link `a.o` and `b.o` together and generate the executable `myprog`.

`'tcc -c a.c'`

Compile `a.c` and generate object file `a.o`.

`'tcc -c asmfile.S'`

Preprocess with C preprocess and assemble `asmfile.S` and generate object file `asmfile.o`.

`'tcc -c asmfile.s'`

Assemble (but not preprocess) `asmfile.s` and generate object file `asmfile.o`.

`'tcc -r -o ab.o a.c b.c'`

Compile `a.c` and `b.c`, link them together and generate the object file `ab.o`.

Scripting:

TCC can be invoked from *scripts*, just as shell scripts. You just need to add `#!/usr/local/bin/tcc -run` at the start of your C source:

```
#!/usr/local/bin/tcc -run
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

TCC can read C source code from *standard input* when - is used in place of `infile`. Example:

```
echo 'main(){puts("hello");}' | tcc -run -
```

2.2 Option summary

General Options:

`-c`

Generate an object file.

`-o outfile`

Put object file, executable, or dll into output file `outfile`.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

`-run source [args...]`

Compile file *source* and run it with the command line arguments *args*. In order to be able to give more than one argument to a script, several TCC options can be given *after* the `-run` option, separated by spaces:

```
tcc "-run -L/usr/X11R6/lib -lX11" ex4.c
```

In a script, it gives the following header:

```
#!/usr/local/bin/tcc -run -L/usr/X11R6/lib -lX11
```

`-v`

Display TCC version.

`-vv`

Show included files. As sole argument, print search dirs. -vvv shows tries too.

`-bench`

Display compilation statistics.

Preprocessor options:

`-Idir`

Specify an additional include path. Include paths are searched in the order they are specified.

System include paths are always searched after. The default system include paths are: `/usr/local/include`, `/usr/include` and `PREFIX/lib/tcc/include`. (`PREFIX` is usually `/usr` or `/usr/local`).

`-Dsym[=val]`

Define preprocessor symbol '`sym`' to val. If val is not present, its value is '`1`'. Function-like macros can also be defined: `-DF(a)=a+1`

`-Usym`

Undefine preprocessor symbol '`sym`'.

`-E`

Preprocess only, to stdout or file (with -o).

Compilation flags:

Note: each of the following options has a negative form beginning with `-fno-`.

`-funsigned-char`

Let the `char` type be unsigned.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

`-fsigned-char`

Let the `char` type be signed.

`-fno-common`

Do not generate common symbols for uninitialized data.

`-fleading-underscore`

Add a leading underscore at the beginning of each C symbol.

`-fms-extensions`

Allow a MS C compiler extensions to the language. Currently this assumes a nested named structure declaration without an identifier behaves like an unnamed one.

`-fdollars-in-identifiers`

Allow dollar signs in identifiers

Warning options:

`-w`

Disable all warnings.

Note: each of the following warning options has a negative form beginning with `-Wno-`.

`-Wimplicit-function-declaration`

Warn about implicit function declaration.

`-Wunsupported`

Warn about unsupported GCC features that are ignored by TCC.

`-Wwrite-strings`

Make string constants be of type `const char *` instead of `char *`.

`-Werror`

Abort compilation if warnings are issued.

`-Wall`

Activate all warnings, except `-Werror`, `-Wunusupported` and `-Wwrite-strings`.

Linker options:

`-Ldir`

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Specify an additional static library path for the `-l` option. The default library paths are `/usr/local/lib`, `/usr/lib` and `/lib`.

`-lxxx`

Link your program with dynamic library libxxx.so or static library libxxx.a. The library is searched in the paths specified by the `-L` option and `LIBRARY_PATH` variable.

`-Bdir`

Set the path where the tcc internal libraries (and include files) can be found (default is `PREFIX/lib/tcc`).

`-shared`

Generate a shared library instead of an executable.

`-soname name`

set name for shared library to be used at runtime

`-static`

Generate a statically linked executable (default is a shared linked executable).

`-rdynamic`

Export global symbols to the dynamic linker. It is useful when a library opened with `dlopen()` needs to access executable symbols.

`-r`

Generate an object file combining all input files.

`-Wl,-rpath=path`

Put custom search path for dynamic libraries into executable.

`-Wl,--enable-new-dtags`

When putting a custom search path for dynamic libraries into the executable, create the new ELF dynamic tag DT_RUNPATH instead of the old legacy DT_RPATH.

`-Wl,--oformat=fmt`

Use *fmt* as output format. The supported output formats are:

`elf32-i386`

ELF output format (default)

`binary`

Binary image (only for executable output)

`coff`

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

COFF output format (only for executable output for TMS320C67xx target)

`-Wl,-subsystem=console/gui/wince/...`

Set type for PE (Windows) executables.

`-Wl,-[Ttext=# | section-alignment=# | file-alignment=# | image-base=# | stack=#]`

Modify executable layout.

`-Wl,-Bsymbolic`

Set DT_SYMBOLIC tag.

`-Wl,-(no-)whole-archive`

Turn on/off linking of all objects in archives.

Debugger options:

`-g`

Generate run time debug information so that you get clear run time error messages: `test.c:68: in function 'test5()': dereferencing invalid pointer` instead of the laconic `Segmentation fault`.

`-b`

Generate additional support code to check memory allocations and array/pointer bounds. `-g` is implied. Note that the generated code is slower and bigger in this case.

Note: `-b` is only available on i386 when using libtcc for the moment.

`-bt N`

Display N callers in stack traces. This is useful with `-g` or `-b`.

Misc options:

`-MD`

Generate makefile fragment with dependencies.

`-MF depfile`

Use `depfile` as output for -MD.

`-print-search-dirs`

Print the configured installation directory and a list of library and include directories tcc will search.

`-dumpversion`

Print version.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Target specific options:

`-mms-bitfields`

Use an algorithm for bitfield alignment consistent with MSVC. Default is gcc's algorithm.

`-mfloat-abi (ARM only)`

Select the float ABI. Possible values: `softfp` and `hard`

`-mno-sse`

Do not use sse registers on x86_64

`-m32, -m64`

Pass command line to the i386/x86_64 cross compiler.

Note: GCC options `-Ox`, `-fx` and `-mx` are ignored.

Environment variables that affect how tcc operates.

`CPATH`
`C_INCLUDE_PATH`

A colon-separated list of directories searched for include files, directories given with `-I` are searched first.

`LIBRARY_PATH`

A colon-separated list of directories searched for libraries for the `-l` option, directories given with `-L` are searched first.

---

3 C language support

3.1 ANSI C

TCC implements all the ANSI C standard, including structure bit fields and floating point numbers (`long double`, `double`, and `float` fully supported).

3.2 ISOC99 extensions

TCC implements many features of the new C standard: ISO C99. Currently missing items are: complex and imaginary numbers.

Currently implemented ISOC99 features:

- variable length arrays.
- 64 bit `long long` types are fully supported.
- The boolean type `_Bool` is supported.
- `__func__` is a string variable containing the current function name.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

- Variadic macros: `__VA_ARGS__` can be used for function-like macros:

```
#define dprintf(level, __VA_ARGS__) printf(__VA_ARGS__)
```

  `dprintf` can then be used with a variable number of parameters.

- Declarations can appear anywhere in a block (as in C++).
- Array and struct/union elements can be initialized in any order by using designators:

```
struct { int x, y; } st[10] = { [0].x = 1, [0].y = 2 };

int tab[10] = { 1, 2, [5] = 5, [9] = 9};
```

- Compound initializers are supported:

```
int *p = (int []){ 1, 2, 3 };
```

  to initialize a pointer pointing to an initialized array. The same works for structures and strings.

- Hexadecimal floating point constants are supported:

```
double d = 0x1234p10;
```

  is the same as writing

```
double d = 4771840.0;
```

- `inline` keyword is ignored.
- `restrict` keyword is ignored.

3.3 GNU C extensions

TCC implements some GNU C extensions:

- array designators can be used without '=':

```
int a[10] = { [0] 1, [5] 2, 3, 4 };
```

- Structure field designators can be a label:

```
struct { int x, y; } st = { x: 1, y: 1};
```

  instead of

```
struct { int x, y; } st = { .x = 1, .y = 1};
```

- `\e` is ASCII character 27.
- case ranges : ranges can be used in `case`s:

```
switch(a) {
case 1 … 9:
      printf("range 1 to 9\n");
```

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

```
        break;
    default:
        printf("unexpected\n");
        break;
    }
```

- The keyword `__attribute__` is handled to specify variable or function attributes. The following attributes are supported:
  - `aligned(n)`: align a variable or a structure field to n bytes (must be a power of two).
  - `packed`: force alignment of a variable or a structure field to 1.
  - `section(name)`: generate function or data in assembly section name (name is a string containing the section name) instead of the default section.
  - `unused`: specify that the variable or the function is unused.
  - `cdecl`: use standard C calling convention (default).
  - `stdcall`: use Pascal-like calling convention.
  - `regparm(n)`: use fast i386 calling convention. *n* must be between 1 and 3. The first *n* function parameters are respectively put in registers `%eax`, `%edx` and `%ecx`.
  - `dllexport`: export function from dll/executable (win32 only)

  Here are some examples:

```
int a __attribute__ ((aligned(8), section(".mysection")));
```

  align variable `a` to 8 bytes and put it in section `.mysection`.

```
int my_add(int a, int b) __attribute__ ((section(".mycodesection")))
{
    return a + b;
}
```

  generate function `my_add` in section `.mycodesection`.

- GNU style variadic macros:

```
#define dprintf(fmt, args…) printf(fmt, ## args)

dprintf("no arg\n");
dprintf("one arg %d\n", 1);
```

- `__FUNCTION__` is interpreted as C99 `__func__` (so it has not exactly the same semantics as string literal GNUC where it is a string literal).
- The `__alignof__` keyword can be used as `sizeof` to get the alignment of a type or an expression.
- The `typeof(x)` returns the type of `x`. `x` is an expression or a type.
- Computed gotos: `&&label` returns a pointer of type `void *` on the goto label `label`. `goto *expr` can be used to jump on the pointer resulting from `expr`.
- Inline assembly with asm instruction:

```
static inline void * my_memcpy(void * to, const void * from, size_t n)
{
int d0, d1, d2;
__asm__ __volatile__(
        "rep ; movsl\n\t"
        "testb $2,%b4\n\t"
        "je 1f\n\t"
        "movsw\n"
        "1:\ttestb $1,%b4\n\t"
        "je 2f\n\t"
        "movsb\n"
```

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

```
            "2:"
            : "=&c" (d0), "=&D" (d1), "=&S" (d2)
            :"0" (n/4), "q" (n),"1" ((long) to),"2" ((long) from)
            : "memory");
    return (to);
    }
```

TCC includes its own x86 inline assembler with a `gas`-like (GNU assembler) syntax. No intermediate files are generated. GCC 3.x named operands are supported.

- `__builtin_types_compatible_p()` and `__builtin_constant_p()` are supported.
- `#pragma pack` is supported for win32 compatibility.

3.4 TinyCC extensions

- `__TINYC__` is a predefined macro to indicate that you use TCC.
- `#!` at the start of a line is ignored to allow scripting.
- Binary digits can be entered (`0b101` instead of `5`).
- `__BOUNDS_CHECKING_ON` is defined if bound checking is activated.

---

4 TinyCC Assembler

Since version 0.9.16, TinyCC integrates its own assembler. TinyCC assembler supports a gas-like syntax (GNU assembler). You can deactivate assembler support if you want a smaller TinyCC executable (the C compiler does not rely on the assembler).

TinyCC Assembler is used to handle files with `.S` (C preprocessed assembler) and `.s` extensions. It is also used to handle the GNU inline assembler with the `asm` keyword.

4.1 Syntax

TinyCC Assembler supports most of the gas syntax. The tokens are the same as C.

- C and C++ comments are supported.
- Identifiers are the same as C, so you cannot use '.' or '$'.
- Only 32 bit integer numbers are supported.

4.2 Expressions

- Integers in decimal, octal and hexa are supported.
- Unary operators: +, -, ~.
- Binary operators in decreasing priority order:
    1. *, /, %
    2. &, |, ^
    3. +, -
- A value is either an absolute number or a label plus an offset. All operators accept absolute values except '+' and '-'. '+' or '-' can be used to add an offset to a label. '-' supports two labels only if they are the same or if they are both defined and in the same section.

4.3 Labels

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

- All labels are considered as local, except undefined ones.
- Numeric labels can be used as local `gas`-like labels. They can be defined several times in the same source. Use 'b' (backward) or 'f' (forward) as suffix to reference them:

```
1:
     jmp 1b /* jump to '1' label before */
     jmp 1f /* jump to '1' label after */
1:
```

4.4 Directives

All directives are preceded by a '.'. The following directives are supported:

- .align n[,value]
- .skip n[,value]
- .space n[,value]
- .byte value1[,...]
- .word value1[,...]
- .short value1[,...]
- .int value1[,...]
- .long value1[,...]
- .quad immediate_value1[,...]
- .globl symbol
- .global symbol
- .section section
- .text
- .data
- .bss
- .fill repeat[,size[,value]]
- .org n
- .previous
- .string string[,...]
- .asciz string[,...]
- .ascii string[,...]

4.5 X86 Assembler

All X86 opcodes are supported. Only ATT syntax is supported (source then destination operand order). If no size suffix is given, TinyCC tries to guess it from the operand sizes.

Currently, MMX opcodes are supported but not SSE ones.

---

5 TinyCC Linker

5.1 ELF file generation

TCC can directly output relocatable ELF files (object files), executable ELF files and dynamic ELF libraries without relying on an external linker.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Dynamic ELF libraries can be output but the C compiler does not generate position independent code (PIC). It means that the dynamic library code generated by TCC cannot be factorized among processes yet.

TCC linker eliminates unreferenced object code in libraries. A single pass is done on the object and library list, so the order in which object files and libraries are specified is important (same constraint as GNU ld). No grouping options (`--start-group` and `--end-group`) are supported.

5.2 ELF file loader

TCC can load ELF object files, archives (.a files) and dynamic libraries (.so).

5.3 PE-i386 file generation

TCC for Windows supports the native Win32 executable file format (PE-i386). It generates EXE files (console and gui) and DLL files.

For usage on Windows, see also tcc-win32.txt.

5.4 GNU Linker Scripts

Because on many Linux systems some dynamic libraries (such as `/usr/lib/libc.so`) are in fact GNU ld link scripts (horrible!), the TCC linker also supports a subset of GNU ld scripts.

The `GROUP` and `FILE` commands are supported. `OUTPUT_FORMAT` and `TARGET` are ignored.

Example from `/usr/lib/libc.so`:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

---

6 TinyCC Memory and Bound checks

This feature is activated with the `-b` (see Invoke).

Note that pointer size is *unchanged* and that code generated with bound checks is *fully compatible* with unchecked code. When a pointer comes from unchecked code, it is assumed to be valid. Even very obscure C code with casts should work correctly.

For more information about the ideas behind this method, see http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html.

Here are some examples of caught errors:

Invalid range with standard string function:

```
{
    char tab[10];
    memset(tab, 0, 11);
}
```

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Out of bounds-error in global or local arrays:

```
{
    int tab[10];
    for(i=0;i<11;i++) {
        sum += tab[i];
    }
}
```

Out of bounds-error in malloc’ed data:

```
{
    int *tab;
    tab = malloc(20 * sizeof(int));
    for(i=0;i<21;i++) {
        sum += tab4[i];
    }
    free(tab);
}
```

Access of freed memory:

```
{
    int *tab;
    tab = malloc(20 * sizeof(int));
    free(tab);
    for(i=0;i<20;i++) {
        sum += tab4[i];
    }
}
```

Double free:

```
{
    int *tab;
    tab = malloc(20 * sizeof(int));
    free(tab);
    free(tab);
}
```

---

7 The `libtcc` library

The `libtcc` library enables you to use TCC as a backend for dynamic code generation.

Read the `libtcc.h` to have an overview of the API. Read `libtcc_test.c` to have a very simple example.

The idea consists in giving a C string containing the program you want to compile directly to `libtcc`. Then you can access to any global symbol (function or variable) defined.

---

8 Developer’s guide

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

This chapter gives some hints to understand how TCC works. You can skip it if you do not intend to modify the TCC code.

8.1 File reading

The `BufferedFile` structure contains the context needed to read a file, including the current line number. `tcc_open()` opens a new file and `tcc_close()` closes it. `inp()` returns the next character.

8.2 Lexer

`next()` reads the next token in the current file. `next_nomacro()` reads the next token without macro expansion.

`tok` contains the current token (see `TOK_xxx`) constants. Identifiers and keywords are also keywords. `tokc` contains additional infos about the token (for example a constant value if number or string token).

8.3 Parser

The parser is hardcoded (yacc is not necessary). It does only one pass, except:

- For initialized arrays with unknown size, a first pass is done to count the number of elements.
- For architectures where arguments are evaluated in reverse order, a first pass is done to reverse the argument order.

8.4 Types

The types are stored in a single 'int' variable. It was chosen in the first stages of development when tcc was much simpler. Now, it may not be the best solution.

```
#define VT_INT        0  /* integer type */
#define VT_BYTE       1  /* signed byte type */
#define VT_SHORT      2  /* short type */
#define VT_VOID       3  /* void type */
#define VT_PTR        4  /* pointer */
#define VT_ENUM       5  /* enum definition */
#define VT_FUNC       6  /* function type */
#define VT_STRUCT     7  /* struct/union definition */
#define VT_FLOAT      8  /* IEEE float */
#define VT_DOUBLE     9  /* IEEE double */
#define VT_LDOUBLE   10  /* IEEE long double */
#define VT_BOOL      11  /* ISOC99 boolean type */
#define VT_LLONG     12  /* 64 bit integer */
#define VT_LONG      13  /* long integer (NEVER USED as type, only
                            during parsing) */
#define VT_BTYPE      0x000f /* mask for basic type */
#define VT_UNSIGNED   0x0010  /* unsigned type */
#define VT_ARRAY      0x0020  /* array type (also has VT_PTR) */
#define VT_VLA        0x20000 /* VLA type (also has VT_PTR and VT_ARRAY) */
#define VT_BITFIELD   0x0040  /* bitfield modifier */
#define VT_CONSTANT   0x0800  /* const modifier */
#define VT_VOLATILE   0x1000  /* volatile modifier */
#define VT_DEFSIGN    0x2000  /* signed type */

#define VT_STRUCT_SHIFT 18   /* structure/enum name shift (14 bits left) */
```

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

When a reference to another type is needed (for pointers, functions and structures), the `32 - VT_STRUCT_SHIFT` high order bits are used to store an identifier reference.

The `VT_UNSIGNED` flag can be set for chars, shorts, ints and long longs.

Arrays are considered as pointers `VT_PTR` with the flag `VT_ARRAY` set. Variable length arrays are considered as special arrays and have flag `VT_VLA` set instead of `VT_ARRAY`.

The `VT_BITFIELD` flag can be set for chars, shorts, ints and long longs. If it is set, then the bitfield position is stored from bits VT_STRUCT_SHIFT to VT_STRUCT_SHIFT + 5 and the bit field size is stored from bits VT_STRUCT_SHIFT + 6 to VT_STRUCT_SHIFT + 11.

`VT_LONG` is never used except during parsing.

During parsing, the storage of an object is also stored in the type integer:

```
#define VT_EXTERN  0x00000080  /* extern definition */
#define VT_STATIC  0x00000100  /* static variable */
#define VT_TYPEDEF 0x00000200  /* typedef definition */
#define VT_INLINE  0x00000400  /* inline definition */
#define VT_IMPORT  0x00004000  /* win32: extern data imported from dll */
#define VT_EXPORT  0x00008000  /* win32: data exported from dll */
#define VT_WEAK    0x00010000  /* win32: data exported from dll */
```

8.5 Symbols

All symbols are stored in hashed symbol stacks. Each symbol stack contains `Sym` structures.

`Sym.v` contains the symbol name (remember an identifier is also a token, so a string is never necessary to store it). `Sym.t` gives the type of the symbol. `Sym.r` is usually the register in which the corresponding variable is stored. `Sym.c` is usually a constant associated to the symbol like its address for normal symbols, and the number of entries for symbols representing arrays. Variable length array types use `Sym.c` as a location on the stack which holds the runtime sizeof for the type.

Four main symbol stacks are defined:

`define_stack`

: for the macros (`#define`s).

`global_stack`

: for the global variables, functions and types.

`local_stack`

: for the local variables, functions and types.

`global_label_stack`

: for the local labels (for `goto`).

`label_stack`

: for GCC block local labels (see the `__label__` keyword).

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

`sym_push()` is used to add a new symbol in the local symbol stack. If no local symbol stack is active, it is added in the global symbol stack.

`sym_pop(st,b)` pops symbols from the symbol stack *st* until the symbol *b* is on the top of stack. If *b* is NULL, the stack is emptied.

`sym_find(v)` return the symbol associated to the identifier *v*. The local stack is searched first from top to bottom, then the global stack.

8.6 Sections

The generated code and data are written in sections. The structure `Section` contains all the necessary information for a given section. `new_section()` creates a new section. ELF file semantics is assumed for each section.

The following sections are predefined:

`text_section`

is the section containing the generated code. *ind* contains the current position in the code section.

`data_section`

contains initialized data

`bss_section`

contains uninitialized data

`bounds_section`
`lbounds_section`

are used when bound checking is activated

`stab_section`
`stabstr_section`

are used when debugging is active to store debug information

`symtab_section`
`strtab_section`

contain the exported symbols (currently only used for debugging).

8.7 Code generation

8.7.1 Introduction

The TCC code generator directly generates linked binary code in one pass. It is rather unusual these days (see gcc for example which generates text assembly), but it can be very fast and surprisingly little complicated.

The TCC code generator is register based. Optimization is only done at the expression level. No intermediate representation of expression is kept except the current values stored in the *value stack*.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

On x86, three temporary registers are used. When more registers are needed, one register is spilled into a new temporary variable on the stack.

8.7.2 The value stack

When an expression is parsed, its value is pushed on the value stack (*vstack*). The top of the value stack is *vtop*. Each value stack entry is the structure `SValue`.

`SValue.t` is the type. `SValue.r` indicates how the value is currently stored in the generated code. It is usually a CPU register index (`REG_xxx` constants), but additional values and flags are defined:

```
#define VT_CONST     0x00f0
#define VT_LLOCAL    0x00f1
#define VT_LOCAL     0x00f2
#define VT_CMP       0x00f3
#define VT_JMP       0x00f4
#define VT_JMPI      0x00f5
#define VT_LVAL      0x0100
#define VT_SYM       0x0200
#define VT_MUSTCAST  0x0400
#define VT_MUSTBOUND 0x0800
#define VT_BOUNDED   0x8000
#define VT_LVAL_BYTE     0x1000
#define VT_LVAL_SHORT    0x2000
#define VT_LVAL_UNSIGNED 0x4000
#define VT_LVAL_TYPE     (VT_LVAL_BYTE | VT_LVAL_SHORT | VT_LVAL_UNSIGNED)
```

`VT_CONST`

indicates that the value is a constant. It is stored in the union `SValue.c`, depending on its type.

`VT_LOCAL`

indicates a local variable pointer at offset `SValue.c.i` in the stack.

`VT_CMP`

indicates that the value is actually stored in the CPU flags (i.e. the value is the consequence of a test). The value is either 0 or 1. The actual CPU flags used is indicated in `SValue.c.i`.

If any code is generated which destroys the CPU flags, this value MUST be put in a normal register.

`VT_JMP`
`VT_JMPI`

indicates that the value is the consequence of a conditional jump. For VT_JMP, it is 1 if the jump is taken, 0 otherwise. For VT_JMPI it is inverted.

These values are used to compile the `||` and `&&` logical operators.

If any code is generated, this value MUST be put in a normal register. Otherwise, the generated code won't be executed if the jump is taken.

`VT_LVAL`

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

is a flag indicating that the value is actually an lvalue (left value of an assignment). It means that the value stored is actually a pointer to the wanted value.

Understanding the use `VT_LVAL` is very important if you want to understand how TCC works.

`VT_LVAL_BYTE`
`VT_LVAL_SHORT`
`VT_LVAL_UNSIGNED`

if the lvalue has an integer type, then these flags give its real type. The type alone is not enough in case of cast optimisations.

`VT_LLOCAL`

is a saved lvalue on the stack. `VT_LVAL` must also be set with `VT_LLOCAL`. `VT_LLOCAL` can arise when a `VT_LVAL` in a register has to be saved to the stack, or it can come from an architecture-specific calling convention.

`VT_MUSTCAST`

indicates that a cast to the value type must be performed if the value is used (lazy casting).

`VT_SYM`

indicates that the symbol `SValue.sym` must be added to the constant.

`VT_MUSTBOUND`
`VT_BOUNDED`

are only used for optional bound checking.

8.7.3 Manipulating the value stack

`vsetc()` and `vset()` pushes a new value on the value stack. If the previous *vtop* was stored in a very unsafe place(for example in the CPU flags), then some code is generated to put the previous *vtop* in a safe storage.

`vpop()` pops *vtop*. In some cases, it also generates cleanup code (for example if stacked floating point registers are used as on x86).

The `gv(rc)` function generates code to evaluate *vtop* (the top value of the stack) into registers. *rc* selects in which register class the value should be put. `gv()` is the *most important function* of the code generator.

`gv2()` is the same as `gv()` but for the top two stack entries.

8.7.4 CPU dependent code generation

See the `i386-gen.c` file to have an example.

`load()`

must generate the code needed to load a stack value into a register.

`store()`

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

must generate the code needed to store a register into a stack value lvalue.

```
gfunc_start()
gfunc_param()
gfunc_call()
```

should generate a function call

```
gfunc_prolog()
gfunc_epilog()
```

should generate a function prolog/epilog.

```
gen_opi(op)
```

must generate the binary integer operation *op* on the two top entries of the stack which are guaranteed to contain integer types.

The result value should be put on the stack.

```
gen_opf(op)
```

same as `gen_opi()` for floating point operations. The two top entries of the stack are guaranteed to contain floating point values of same types.

```
gen_cvt_itof()
```

integer to floating point conversion.

```
gen_cvt_ftoi()
```

floating point to integer conversion.

```
gen_cvt_ftof()
```

floating point to floating point of different size conversion.

```
gen_bounded_ptr_add()
gen_bounded_ptr_deref()
```

are only used for bounds checking.

8.8 Optimizations done

Constant propagation is done for all operations. Multiplications and divisions are optimized to shifts when appropriate. Comparison operators are optimized by maintaining a special cache for the processor flags. &&, || and ! are optimized by maintaining a special 'jump target' value. No other jump optimization is currently performed because it would require to store the code in a more abstract fashion.

Concept Index

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

Jump to: _
A B C D E F G I J L M O P Q R S T U V W

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

PDFmyURL