KickC

Reference Manual

Version 0.1

by Jesper Gravgaard / Rex of Camelot

Contents

1 What is KickC?

KickC is a compiler for a C-family language creating optimized and readable 6502 assembler code.

The KickC Language

The KickC language is classic C with some limitations, some modifications and some extensions to ensure an optimal fit for creating 6502 assembler code.

The language has the familiar C syntax and supports many of the basic features of C, so it should be quite easy to get started with if you have programmed in C or any similar language. Here is a simple “hello world” program, that prints “hello world!” at the top of the screen.

```
import “print”
void main() {
      print_str(“hello world!@”);
}
```

The language has a number limitations compared to standard C, for example no support for structs, enums, unions, pointers to functions or pointers to pointers. Some features were omitted because they cannot be realized in a way that creates optimized 6502 assembler code. Others were omitted simply because they have not yet been implemented in the current version.

The language also have some modifications and extensions to standard C. The modifications and extensions were included either to allow creation of better 6502 assembler code or for convenience. A significant modification is that the language does not support normal C types (char, int, long), but instead has its own types specifically aimed at the 6502 (byte, word, dword).

All limitations, modifications and extensions are described in the following sections.

Optimized and Readable 6502 Assembler Code

The KickC Compiler produces assembler code for the MOS Technology 6502 processor. The assembler code is produced as source code that can be assembled to binary by the Kick Assembler (http://theweb.dk/KickAssembler).

The compiler uses a number of modern optimization methods to create 6502 assembler code that executes as fast as possible and does not contain unnecessary boilerplate. The optimization techniques include

- Detection of constant values and expressions
- Optimized allocation of registers to variables
- Optimized parameter and return value passing to/from functions
- Minimizing the number of zero-page addresses used for storing variables
- Choosing optimal assembler instructions to represent each statement
- Removing unused functions, variables and code
- Peephole optimization of the generated assembler code

The optimization techniques are also explained in more detail in later sections.

Below a slightly more complex version of hello world, which prints “hello world!” with an added space between each letter on the first and third line of the C64 default screen at $400. This example illustrates how the KickC compiler creates optimized readable 6502 assembler.

helloworld2.kc

```
byte* screen = $400;

void main() {
    byte* hello = "hello world!@";
    print2(screen, hello);
    print2(screen+2*40, hello);
}

void print2(byte* at, byte* msg) {
    byte j=0;
    for(byte i=0; msg[i]!='@'; i++) {
        at[j] = msg[i];
        j += 2;
    }
}
```

helloworld2.asm

```
.label screen = $400
main: {
    lda #<screen
    sta print2.at
    lda #>screen
    sta print2.at+1
    jsr print2
    lda #<screen+2*$28
    sta print2.at
    lda #>screen+2*$28
    sta print2.at+1
    jsr print2
    rts
    hello: .text "hello world!@"
}
print2: {
    .label at = 2
    ldy #0
    ldx #0
  b1:
    lda main.hello,x
    sta (at),y
    iny
    iny
    inx
    lda main.hello,x
    cmp #'@'
    bne b1
    rts
}
```

The KickC compiler uses the following insights to optimize the helloworld2 program:

- The screen pointer is never modified and is therefore a constant location in memory.
- The second parameter to the *print2*-function (*msg)* always has the same value *main.hello,* so that can be hardcoded inside the method body instead of parsing it.

- The contents of *msg[i]* (ie. the hardcoded *main.hello* string) can be addressed using simple indexing.
- The *at*-parameter in *print2* is truly variable, so it is placed on zero-page ($2 and $3) and indirect indexing can be used for addressing the contents of *at[j]*.
- The X-register is optimal for the *i* variable in the *print2* function as it is good at indexing and incrementing.
- The Y-register is optimal for the *j* variable in the *print2* function as it is good at indirect indexing and incrementing.
- When 2 is added to the *j*-variable it is better to do INY twice than to use addition.
- The A-register is optimal for holding the current character *c* of the message being moved to the screen as LDA and STA can be efficiently indexed by X and indirect indexed by Y.

When generating Kick Assembler code the KickC compiler tries to ensure that the assembler code is readable and corresponds to the source C program as much as possible. This includes using the same names, using scopes and recreating constant calculations in assembler, when possible.

The KickC Compiler creates readable 6502 assembler code for the helloworld2 program by:

- Using the variable and function-names *screen*, *hello*, *main, print2, at* in the generated code
- Recreating the calculation of the constant *screen+2*40* in the assembler code as *screen+2*$28*
- Creating a local named scope in the assembler-code for the methods *main* and *print2* by enclosing them in curly braces.
- Placing method-local data and labels inside the method scope.This allows other assembler code to access the local data/labels using dot-syntax eg. *main.hello* or *print2.at*

Getting Started

- Downloading
- Installing
- Compiling

2 KickC Language Reference

KickC is a C-family language, and much of the syntax and semantics is the same as C99. In the following the different parts of the language is explained. Finally the differences between KickC and standard C are listed.

Variables

Variables are declared like in regular C by *`type name`* and can include an optional initialization assignment. The following declares a byte variable with the name *size* and the initial value 12.

```
byte size = 12;
```

In KickC variables can be declared at any point in the program, outside functions or inside function declarations.

Data Types

Integers

KickC does not support the standard C integer data types. Instead KickC has the following fixed size integer types, that have names more familiar on the 6502 platform.

Type Name	**Description**
`byte` `unsigned byte`	An unsigned 8 bit (1 byte) integer. Range is [0;255].
`signed byte`	A signed 8 bit (1 byte) integer in two's complement. Range is [-128;127].
`word` `unsigned word`	An unsigned 16 bit (2 byte) integer Range is [0;65,535].
`signed word`	A signed 16 bit (2 byte) integer in two's complement. Range is [−32,767; +32,767].
`dword` `unsigned dword`	An unsigned 32 bit (4 byte) integer. Range is [0, 4,294,967,295].
`signed dword`	A signed 32 bit (4 byte) integer in two's complement. Range is [−2,147,483,647, 2,147,483,647].

If the integer types are declared without a unsigned/signed prefix they default to unsigned.

Integer literals can be either decimal, hexadecimal or binary. The syntax for hexadecimal integer literals support both C syntax (prefixing with `0x` for) 6502 assembler syntax (prefixing with $). Similarly the syntax for binary supports both prefixing with `0b` and `%`.

Prefix	**Format**	**Examples**
	Decimal	`12` `53280`
`0x` `$`	Hexadecimal	`0x40` `$dc01`
`0b` `%`	Binary	`0b101` `%1100110011001100`

Character literals are unsigned bytes. The syntax for a character literal is the character enclosed in single quotes. Numerically the character is represented by the C64 screen code. The following initializes the variable *c* to the character 'c'.

```
byte c = 'c';
```

Booleans

KickC also has a boolean type called `bool`. The boolean literals are `true` and `false`. Underlying the boolean type is a byte containing either 0 (if false) or 1 (if true). The following is an example of a boolean variable called *enabled*.

```
bool enabled = true;
```

Pointers

Pointers to all integer types and booleans are supported and declared using the syntax *`type*`*. The following is an example, where *screen* is a pointer to a byte and *pos* is a pointer to a signed word.

```
byte* screen;
signed word* pos;
```

Pointers to pointers and pointers to functions are not supported.

Arrays

Arrays of integer and boolean types are supported using the syntax *`type[]`* or *`type[size]`*. For all practical purposes array variables are treated exactly like pointers.

Arrays can be initialized by an array literal written as comma-separated values inside curly braces eg. `{ 1, 2, 3 }`. They can also be initialized with all zero values just by declaring the array to have a specific size.

String literals can also be used to initialize an array of unsigned bytes. The syntax for a string literal is the string enclosed in double quotes.

Arrays that are initialized will allocate the memory needed for the size.

In the following example, *sums* is array of 3 signed bytes initialized with zeros, *fibs* is an array of 6 unsigned words containing the first fibonacci numbers and *msg* is an unsigned byte array containing the numeric value of the 5 characters ‘h’, ‘e’, ‘l’, ‘l’ and ‘o’. Finally *bs* is simply a pointer to a boolean.

```
signed byte[3] sums;
word[] fibs = { 1, 1, 2, 3, 5, 8 };
byte[] msg = "hello";
bool[] bs;
```

Arrays of arrays and arrays of pointers are not supported.

Constants

Constants can be declared by using the `const` keyword.

```
const byte SIZE = 42;
```

The compiler is quite good at detecting constants automatically, so it is not strictly necessary to declare any constants. However declaring a constant can help make the code more readable and will generate an error if any code tries to modify the value.

When an array is declared constant only the pointer to the array is constant. The contents of the array can still be modified.

No Floating Point Types

KickC has no floating point types, as the 6502 has no instructions for handling these.

Expressions

Expressions in KickC consist of operands and operators. Operands are either data type literals or names of variables or constants. All well known expression operators from C and similar languages also exist in KickC. An example of an expression is `(bits & $80) != 0`

Arithmetic Operators

The arithmetic operators support performing simple numeric calculations

- `a + b` Addition
- `a - b` Subtraction
- `- a` Negation
- `+ a` Positive
- `a * b` Multiplication
- `a / b` Division
- `a % b` Modulo

Multiplication, division and remainder are allowed, however there is no run-time support for these operators as the 6502 has no instructions supporting them. Therefore they can only be used for calculating values that end up being constant. If they are used in a way that requires runtime support the compiler will fail with an error.

For convenience plus (addition) can also be used for concatenating strings initializers with other strings or characters. The following initializes the variable *msg* with the text “hello world”.

```
byte[] msg = "hello"+' '+"world";
```

Bitwise Operators

The bitwise operators operate on the individual bits of the numeric operands.

- `a & b` Bitwise and
- `a | b` Bitwise or
- `a ^ b` Bitwise exclusive or
- `~ a` Bitwise not
- `a << n` Bitwise shift left n bits
- `a >> n` Bitwise shift right n bits

Relational Operators

The relational operators compare values and has a boolean value result.

- `a == b` Equal to
- `a != b` Not equal to
- `a < b` Less than
- `a <= b` Less than or equal to
- `a > b` Greater than
- `a >= b` Greater than or equal to

Logical Operators

The logical operators operate on boolean operands and has a boolean result.

- `a && b` Logical and
- `a || b` Logical or
- `! a` Logical not

When `&&` and `||` are used in *if, while, do-while* or similar statements they are short circuit-evaluated meaning that if *a* evaluates to *true* in `if(a || b) { ... }` then *b* is never evaluated. Similarly if *a* evaluates to *false* in `if(a && b) { ... }` then *b* is never evaluated.

Assignment Operators

Assigning a value to a variable is also an expression operator that returns the value assigned. This means that assignments can be nested inside expressions, and that they can be chained if multiple variables should be assigned the same value like `a = b = 0`. An assignment of course has side effects, as it modifies the assigned variable.

Compound assignment operators, such as `a += b` is a convenient shorthand for updating the value of a variable. It works exactly like `a = a + b`.

- `a = b` Assignment
- `a += b` Addition assignment
- `a -= b` Subtract assignment
- `a *= b` Multiply assignment
- `a /= b` Divide assignment
- `a %= b` Modulo assignment
- `a <<=b` Left shift assignment
- `a >>=b` Right shift assignment
- `a &= b` Bitwise and assignment
- `a |= b` Bitwise or assignment
- `a ^= b` Bitwise exclusive or assignment

Increment/decrement Operators

The pre-increment/decrement and post-increment/decrement operators is a convenient way of incrementing/decrementing the value of a numeric variable just before or just after the value is used in an expression.

- `++a` Pre-increment
- `--a` Pre-decrement
- `a++` Post-increment
- `a--` Post-decrement

Pre-incrementing works just like incrementing the value of c before the statement, meaning that `a = b + ++c;` is the same as `c += 1; a = b + c;` Similarly post-incrementing works just like incrementing the value after the statement, meaning that `a = b + c--;` is the same as `a = b + c; c -= 1;`

Pointer and Array Operators

The two basic pointer operators are the `&a` address-of operator, which creates a pointer to a variable and the `*a` pointer dereference operator, which supports reading and writing the value pointed to by the pointer.

The array indexing operator a[b], which supports reading and writing of array elements, is in fact also a pointer operator. Because an array variable is actually a pointer to the start of the array the array dereference operator a[b], which is actually shorthand for *(a+b).

- `&a` Address of
- `*a` Pointer dereference
- `a[b]` Array indexing

Unlike in standard C, KickC performs both pointer addition and array indexing by number of bytes. Where standard C automatically handles the size of the element type, the programmer has to take these into account when using pointers and arrays in KickC.

Expression	**Standard C**	**KickC**
`ptr++`	Increases the pointer so it points to the next element in memory. If the pointer points to a byte it is increased by 1, if it points to a word it is increased by 2, etc.	Increases the pointer so it points to the next byte in memory. The pointer is increased by 1 byte irregardless of the size of the element type pointed to by *ptr*.
`ptr+n`	Adds *n* element sizes to the pointer. If elements are placed consecutively in memory this creates a pointer that points to the *n*'th element after the element pointed to by *ptr*.	Adds *n* bytes to the pointer, creating a pointer that points *n* bytes after *ptr*.
`a[i]`	Addresses the i'th element of the array a - equivalent to **(a+i)*.	Addresses an element *i* bytes into the memory of the array *a* - equivalent to **(a+i)*.

KickC only supports array indexing up to 255 bytes after the start of the array. This is to allow the compiler to always convert array indexing into assembler register X/Y-indexing.

Low/High Operators

An extension in KickC is the inclusion of operators that allow addressing the low/high byte of a word, and the low/high words of a dword. These are well known from 6502 assemblers.

The low-operator `<a` addresses the low-byte of the word *a*, or the low-word if *a* is a dword. Similarly the high-operator `>a` addresses the high-byte of a word or high-word of a dword.

The low/high operator can also be used on the left side of assignments to modify only the low/high byte of a word or low/high word of a dword.

- `<a` Low part of
- `>a` High part of

The following example sets the low part of the word in *a* to 0. If *a* was $0428 before the assignment it will be $0400 afterwards.

```
<a = 0;
```

Function Calls

Function that returns a value can be called as part of an expression. Functions are called by the normal parenthesis-syntax with parameter values separated by commas. Coding functions is described in the section *Functions*.

- `f(a,b)` Function Call

Automatic Type Conversion and Type Casting

KickC handles automatic type conversions a differently than standard C. Where standard C converts all small integers to int before evaluating expressions KickC supports evaluating operators for all types and only performs conversions if they are strictly necessary. Limiting the number of automatic type conversions helps creating better optimized 6502 assembler. In a many cases KickC can even handle operators for two values of different type directly. For instance adding a byte to a word can be done in more optimal 6502 code than first converting the byte to a word and then adding the two words. In practice the difference to standard C rarely has any consequences.

Like standard C, KickC will ensure automatic type conversion (if necessary) as long as the type of one value can contain all values of the type of the other. For instance an unsigned byte can be automatically converted to a signed word as signed words can hold all possible unsigned byte values. An unsigned byte can not be automatically converted to an signed byte, since a signed byte cannot hold all possible byte values.

The automatic conversions that Kick can perform for each type are the following

Type	**Can be automatically converted to**
unsigned byte	unsigned word, signed word, unsigned dword, signed dword
signed byte	signed word, signed dword
unsigned word	unsigned dword, signed dword
signed word	signed dword
unsigned dword	-
signed dword	-

The cast operator can be used to perform explicit type conversion. Casting also allows conversions to a type that cannot hold all possible values, and where some information may be lost in the conversion, for example casting a signed byte to an unsigned byte.

- `(type)a` Casting

For sub-expressions containing only constants the KickC compiler tries to infer the type of the sub-expression. This is done by performing the calculation and then checking which types can hold the calculated value. For instance the calculation *$4000/$80* is inferred to match any integer type except signed byte (since a signed byte cannot hold 128) . This also differs from standard C, where all constant integer numbers are ints unless specified otherwise.

Operator Precedence and Parenthesis

Operators precedence decides which operators are applied first when multiple operators are combined in an expression. For instance multiplication is performed before addition in `a*b+c`. In KickC operators generally have the same precedence as in standard C.

Precedence rules can be overridden by using explicit parentheses in expressions. For instance to perform addition before multiplication `(a+b)*c`.

- `( a )` Parenthesis

The following table shows KickC operator precedences. Operators at the top of the table binds most tightly.

Precedence	**Operators**	**Associativity**

1	`a++  a--  f( )  a[b]`	Left-to-right
2	`++a  --a  +a  ~a` `!a  (t)a  *a  &a`	Right-to-left
3	`a*b  a/b  a%b`	Left-to-right
4	`a+b  a-b`	Left-to-right
5	`a<<b  a>>b`	Left-to-right
6	`<a  >a`	Left-to-right
7	`a<b  a<=b  a>b  a>=b`	Left-to-right
8	`a==b  a!=b`	Left-to-right
9	`a&b`	Left-to-right
10	`a^b`	Left-to-right
11	`a|b`	Left-to-right
12	`a&&b`	Left-to-right
13	`a||b`	Left-to-right

14	`a=b  a+=b  a-=b  a*=b`
	`a/=b  a%=b  a<<=b  a>>=b`
	`a&=b  a^=b  a|=b`
		Right-to-left

Statements

The statements of a KickC program control the flow of execution. KickC supports most statements supported by standard C.

Statements are separated by semicolons `stmt; stmt;` and can be grouped together in blocks using curly braces `{ stmt; stmt; }`.

Expressions and Assignments

All expressions are valid statements.There are two typical ways of using expressions as statements. The first is an assignment, which is an expression, modifying the value of a variable.

```
a += 2;
```

The second is calling a function that has a side effect.

```
print(“hello”);
```

If

The body of an if-statement is only executed if the condition is `true`. The if-statement can have an else-body, that is executed if the condition is not true.

The following if-statement prints “even” to the screen if `a` is even.

```
if((a&1)==0) { print(“even”); }
```

The following if-statement increases `b` if a is less than 10 and decreases `b` otherwise.

```
if(a<10) { b++; } else { b--; }
```

While

The body of a while-loop is executed repeatedly as long as the condition is still true. The while-loop is executed by first evaluating the condition. If the condition is true the body is executed and the loop starts over. If the condition is not true execution continues after the loop. If the condition is not true the first time the loop is encountered then the body is never executed.

The following while-loop prints `i` dots on the screen, while counting i down to zero.

```
while(i!=0) { print(“.”); i--; }
```

Do-While

The do-while-loop is very similar to the while-loop. The body of a do-while-loop is executed repeatedly as long as the condition is true. In the do-while-loop the body is executed first and then the condition is evaluated. If the condition is true the loop starts over. If the condition is not true execution continues after the loop.The body of the loop is always executed at least once.

The following do-while-loop keeps scanning the keyboard until the space key is pressed.

```
do {
  keyboard_event_scan();
} while (keyboard_event_get()!=KEY_SPACE)
```

For

The for-loop is a convenient way of creating a loop, where a loop-variable is initialized, the body is executed, the loop-variable is incremented and finally the condition is evaluated to determine whether to repeat the loop again.

A for-loop has the following syntax

```
for(init; condition; increment) { body }
```

and is equivalent to the following KickC code

```
init;
do {
  body;
  increment;
} while(condition)
```

The body and increment is always executed at least once in a KickC for-loop. This differs from standard C, where the condition is evaluated before the body and increment, enabling for-loops where the body is never executed. The KickC behavior was chosen because it can create more efficient 6502 ASM code than the standard C behavior.

KickC has an additional convenience syntax for creating simple for-loops that loop over an integer range. The following for-loop executes the body 128 times with `i` having values 0,1,...,127

```
for(byte i : 0..127) { body }
```

And is equivalent to

```
for(byte i=0; i!=127+1; i++) { body }
```

This convenience syntax only accepts constants or expressions evaluating to constants as the ends of the integer interval. It can loop both back wards and forwards.

Functions

- Declarations & parameters
- Returns
- Calls
- The main() function

Comments

- Oneline // …
- Block /* … */

Imports

- Importing
- Forward referencing variables

Directives

  - Align
  - Register
  - Inline

Inline Assembler

- Asm { ...}
- Assembler Language
  - Memory Directives (bytes)
  - Labels
  - Instructions and addressing modes
  - Expressions

Comparison with standard C

Supported

Not supported/implemented

- C integer types *char, short, int, long*
- Floating point types *float, double*
- Runtime multiplication *a * b*
- Runtime division *a / b*
- Runtime modulo *a % b*
- Enum *enum Status { on, off } status;*
- Struct *struct Point { byte x; byte y; } p;*
- Union *union { byte b; word w; } u;*
- Pointer to pointer *byte **pptr;*
- Array of arrays *byte[4][4] baa;*
- Function pointers *byte (*fptr)(byte);*
- Comma operator *byte a=0, b=1;*
- Ternary operator *(b<4) ? 'a' : 'b';*
- Heap Allocation *dword* dw = malloc(4);*

- Recursive functions *byte fib(byte n) { return fib(n-1)+fib(n-2)}*
- Sizeof operator *byte s = sizeof (word);*
- Alignof operator *byte s = alignof (word);*
- Variadic functions *printf(const byte* format, ...)*
- Volatile variables *volatile byte b;*
- C preprocessor (see imports)
- C standard library

Limitations / Modifications

- Types without a signed/unsigned prefix are assumed to be unsigned.
- Array indexing and pointer incrementing always done by number of bytes. Pointer arithmetic does not take into account the size of the element type.
- Array indexing can only index up to 256 bytes of memory.
- For-loops always execute the body at least once.
- Multiplication and division only supported for constants (no runtime support for multiplication/division without using a library and a function call.)
- Arrays and strings are always statically allocated (as data bytes in the resulting assembler).
- Conditions in *if, while, do-while* and similar statements only accept boolean values. Numeric values are not accepted without casting.
- Alignment directive *align($100)*
- Register directive *register(X)*
- Inline assembler *asm { SEI CLD };*
- Main-function *void main() { … }*

Extensions

- 6502 types *byte, word, dword*
- Imports *import “print”*
- Forward referencing variables in the outer scope
- Ranged for-loops *for(byte i: 0.. 10) { }*
- Word operator *word w = { hi, lo };*
- Lo/hi-byte operator *byte lo = <w; <w = 12;*

The KickC Libraries

No runtime library!

Some libraries

- print

- c64
- c64dtv
- keyboard
- division
- Multiply
- Fast multiply
- sinus
- basic-floats

3 Working with KickC

The Coding Workflow / Related Tools

- Assembling
- Executing (Emulators or The Real Thing)
- Editing
- Reporting Issues
- The Source Code
- Contributing

Combining KickC and KickAssembler

Optimizing KickC Code

- Use do {} while() instead of while() {}
- Unsigned types are more optimal than signed types.
- Use array indexing instead of incrementing pointers.
- Booleans are not always very efficient. Often bytes are better.
- Use inline functions
- Use (experimental) inline loops
- Use normal assembler optimization techniques (putting a calculated result that is used multiple times into a variable instead of repeating the calculation, create arrays for lookup instead of repeating a calculation many times, loop unrolling,)

Handling Compiler Errors

Missing Assembler Fragments

4 The Compiler Architecture

1. Loading and Parsing
2. Creating The Single-Static Assignment Control Flow Graph and Symbol Table
3. Optimizing the SSA Control Flow Graph
4. Control Flow Graph Analysis
5. Register Allocation
6. Assembler Code Generation using Assembler Fragments
7. Assembler Code Optimization

Assembler Fragment Sub System

Adding missing assembler fragments