KickC

Reference Manual

Version 0.8 (BETA)

by Jesper Gravgaard / Rex of Camelot

Contents

BETA

KickC is currently in beta, and occasionally crash with a cryptic error or create ASM code that does not work properly. Feel free to test it and report any problems or errors you encounter. Also, be prepared that breaking changes (to syntax, to semantics, etc.) may be implemented in the next versions.

1 What is KickC?

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

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

The KickC Language

The language is basically C and it 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 some limitations compared to standard C, for example no support for unions or reentrant functions. 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 has a few extensions to standard C. The modifications and extensions were included either to allow creation of better 6502 assembler code or for convenience.

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 0x400. This example illustrates how the KickC compiler creates optimized readable 6502 assembler.

helloworld2.c

```
char* screen = 0x400;

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

void print2(char* at, char* msg) {
    char j=0;
    for(char 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 #0
    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

The KickC development is hosted on gitlab https://gitlab.com/camelot/kickc. Here it is possible to follow the development and to download the latest binary release.

You install KickC on your own computer by:
1. Download the newest KickC release from https://gitlab.com/camelot/kickc/releases
2. Unpack the zip-file to a folder of your own choice.

The zip-file contains the following
    - *bin* Folder containing bat/sh-files for running KickC.
    - *examples* Folder containing some example KickC programs.
    - *include* Folder containing header-files for useful library functions usable in your own program.
    - *lib* Folder containing C-files implementing the useful library functions usable in your own program.

- *jar* Folder containing the KickC JAR-file plus a few other JARs needed for running KickC (antlr4-runtime, picocli and KickAssembler).
- This manual in PDF-format and some files with license-information.

KickC is written in Java. To use KickC you need to download and install a Java runtime from https://www.java.com. Java must be added to your PATH, or the environment variable JAVA_HOME must point to the folder containing the Java installation.

NOTE: KickC runs a lot faster on 64bit Java than on 32bit Java. You are therefore encouraged to ensure that your Java is a 64bit version. You can check by executing `java -version` in a Terminal/Command Prompt.

After installing KickC and you can compile a simple sample KickC program by doing the following:

MacOS

1. Start a Terminal
2. `cd` to the folder containing KickC
3. Enter the command
   `bin/kickc.sh examples/helloworld/helloworld.c`

Windows

1. Start a Command Prompt
2. `cd` to the folder containing KickC
3. Enter the command
   `bin\kickc.bat examples\helloworld\helloworld.c`

This compiles the helloworld KickC program `examples/helloworld/helloworld.c` and produces assembler code in `examples/helloworld/helloworld.asm`. The resulting ASM-file can then be assembled using KickAssembler, producing a runnable program, that can be executed using an emulator or transfered to the real 8-bit hardware.

To make the workflow convenient KickC has a command line option (`-a`) to compile the resulting ASM code with KickAssembler. This is one of the reasons that the KickAssembler JAR is bundled with the release of KickC.

For even more convenience KickC also has a command line option (`-e`) that both assembles the ASM code and executes the resulting binary program in the VICE Commodore 64 emulator (http://vice-emu.sourceforge.net/). The option requires VICE `x64` to be available in the PATH.

To compile, assemble and execute the example program `simple-multiplexer.c` (a sprite multiplexer moving 32 balloons in a sinus on the screen) use the following command in the kickc-folder (assuming MacOs)

```
bin/kickc.sh -e examples/multiplexer/simple-multiplexer.c
```

To create your own KickC-programs use any text-editor to create a source file containing a `void main()` function, save it to the file system (using the `.c` extension is recommended) and compile it by passing it to `kickc.sh` (on MacOs) or `kickc.bat` (on Windows).

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 signed char variable with the name *size* and the initial value 12.

```
char size = 12;
```

If variables do not have an initial assignment they will be initialized with the default value zero.

It is possible to declare multiple variables with the same type by separating the names and optional initializations with commas. Here two `unsigned int` variables `a` and `b` are declared, `a` is initialized to 4 while `b` is initialized to the default value (zero).

```
unsigned int a = 4, b;
```

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

Data Types

Integers

KickC supports the standard C integer data types (char, short, int, long), but also adds some fixed size integer types, that have names more familiar on the 6502 platform (byte, word, dword).

Type Name	**Description**

`char`
`unsigned char`
`byte`
`unsigned byte`
	An unsigned 8 bit (1 byte) integer.
	Range is [0;255].

`signed char` `signed byte`	A signed 8 bit (1 byte) integer in two's complement. Range is [-128;127].

`unsigned short`
`unsigned int`
`unsigned`
`word`
`unsigned word`
	An unsigned 16 bit (2 byte) integer
	Range is [0;65,535].

`short`
`signed short`
`int`
`signed int`
`signed`
`signed word`
	A signed 16 bit (2 byte) integer in two's complement.
	Range is [−32,767; +32,767].

`unsigned long`
`dword`
`unsigned dword`
	An unsigned 32 bit (4 byte) integer.
	Range is [0, 4,294,967,295].

`long`
`signed long`
`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 standard C integer types are declared without a unsigned/signed prefix they default to signed, except char which defaults to unsigned. If the special 6502-friendly 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`) and 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 / chars. 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'.

```
char c = 'c';
```

Escape sequences can be used to represent special characters such as newline. The following character escape sequences are supported

- `'\n'` newline
- `'\r'` carriage return
- `'\f'` form feed
- `'\''` single quote
- `'\"'` double quote
- `'\\'` backslash
- `'\\'` backslash

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 char and *pos* is a pointer to a signed integer.

```
char* screen;
int* pos;
```

Pointers to pointers are also supported. Here an example of a pointer to a pointer to an unsigned char.

```
unsigned char** screenptr = &screen;
```

For functions it is only possible to use pointers to functions that has no return value and take no parameters.

Arrays

Arrays are supported using the syntax *`type a[]`* or *`type a[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 a string enclosed in double quotes. Strings can use escape sequences to represent special characters such as newline. See character literals above for a list of the escape sequences. Stings are per default zero terminated. It is possible to create a string that is not zero terminated by adding the special suffix z after the last double quote.

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

In the following example, *sums* is array of 3 signed chars initialized with zeros, *fibs* is an array of 6 signed integers containing the first Fibonacci numbers. *msg* is an unsigned char array containing the numeric value of the 5 characters ‘h’, ‘e’, ‘l’, ‘l’ and ‘o’ plus a sixth value that is zero (because strings are zero terminated) while *msg2* only has the 5 characters without the final zero. Finally *bs* is simply a pointer to a boolean.

```
signed char sums[3];
int fibs[] = { 1, 1, 2, 3, 5, 8 };
char msg[] = "hello";
char msg2[] = "hello"z;
bool bs[];
```

Arrays of arrays are not supported.

Constants

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

```
const char 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 processor 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 limited run-time support for these operators as the 6502 has no instructions supporting them. Any multiplication/division/remainder that is a part of a calculation of a constant value is allowed. There is also runtime-support for unsigned multiplication of a variable by a constant.

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.

Conditional Operator

The conditional operator is also called ternary operator because it is the only operator using three operands. It is used to choose between two different values.

- `a ? b : c` Conditional

It evaluates the first operand, which should be boolean. If the first operand evaluates to true it returns the value of the second operator. If the first operand evaluates to true it returns the value of the third operator. It uses short-circuit-evaluation meaning that only one of the two last operands are evaluated depending on the value of the first operand.

In this example d will be set to the value of c if c is positive and to -c if c is negative.

```
char d = c>0 ? c : -c;
```

Comma Operator

The comma operator can be used to evaluate multiple values in place of an expression. It evaluates the first operand and discards the results. Then it evaluates the second operand and returns the results.

- `a , b` Comma

Using it can produce code that is hard to read, however it can come to good use in for()-statements to increment multiple variables. Here an example of a for()-loop with two loop-variables `i` and `j`.

```
for(unsigned char i=0, j=0; i<32; i++, j+=2) { ... }
```

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

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 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 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 char can be automatically converted to a signed int as signed ints can hold all possible unsigned char values. An unsigned char can not be automatically converted to a signed char, since a signed char cannot hold all possible unsigned char values.

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

Type	**Can be automatically converted to**
unsigned char	unsigned int, signed int, unsigned long, signed long
signed char	signed long, signed long
unsigned int	unsigned long, signed long
signed int	signed long
unsigned long	-
signed long	-

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 char to an unsigned char.

- `(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 char (since a signed char 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:c`	Right-to-left

15	`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

16	`a,b`	Left-to-right

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--; }
```

If the condition is an integer or pointer expression then a value of zero is considered `false` and a non-zero value is considered `true`. The following if-statement decrements `i` if it is non-zero

```
if(i) i--;
```

Switch

The switch-statement is used to choose between a number of cases. It is usable when you would otherwise use several consecutive if-statements. Switch works by examining a single expression and comparing the value to a number of cases labelled with a constant value. The execution starts at the case where the label matches the value. Execution then continues forward until through the statements of the following cases. A break-statement can be used to break out of the switch. At the end a special default-case can be used for catching values that were not matched by any other case.

The following switch-statement examines the char `c` and chooses what to do: if `c` is a zero it exits the function, if `c` is a TAB or newline it prints a space - and for all other values of `c` it prints `c` itself.

```
switch(c) {
case 0:
	return;
case '\t':
case '\n':
	print(' ');
	break;
default
	print(c);
}
```

Notice how the TAB case is empty, but since switch continues to execute statements from following cases the execution will start at the print(' '). The return-statement in case 0 also breaks the execution of the switch.

Switch has the following syntax

```
switch(expr) {
case const1:
  body1
case const2:
  body2
…
default:
  body3
}
```

Where `expr` is an expression giving the value to examine and `const1, const2`, … are expressions that must evaluate to constants. Finally `body1, body2`, … are sequences of statements.

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

The condition is evaluated before the body and increment, enabling for-loops where the body and increment is never executed.

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(char i : 0..127) { body }
```

And is equivalent to

```
for(char 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 backwards and forwards.

Break and Continue

The `break` statement terminates a loop or a switch, whereas `continue` statement forces the next iteration of a loop. These statements are very useful when creating complex loop logic.

The following loop prints a string on the screen, skipping all spaces by using the continue statement. When it encounters a zero char in the string it stops printing using the break `statement`.

```
char* screen = $400;
char str[] = "hello brave new world!";
for( char i: 0..255) {
      if(str[i]==0) break;
      if(str[i]==' ') continue;
      *screen++ = str[i];
}
```

Functions

Functions are named pieces of code that can be reused by calling them and passing different parameters.

Calling functions

In the following code a function called `max` is called 3 times. The max-function takes 2 chars parameters and returns the maximal value of the two. After this code `m` will have the value 31 and `n` will have the value 47.

```
char m = max(31, 9 );
char n = max(m, max(47, 7));
```

In general, a function is called using the syntax:

name(*param1, param2,* ...)

Where

- *name* is the name if the function
- *param1* is the value of the first parameter
- *param2* is the value of the second parameter

The number of parameters passed must exactly match the number of parameters the function expects.

If the function returns a value then the function call can be used as part of any expression. An example of this can be seen in the nested call to max above.

Creating functions

Functions are created by adding function declarations to your program. The following is a declaration of the max-function used above. It expects two char value parameters, finds the largest one and returns it.

```
char max(char a, char b) {
  if(a>b) {
    return a;
  } else {
    return b;
  }
}
```

In general a function declaration has the following syntax:

```
return-type name(param-type1 param-name1, param-type1 param-name1, ...) { body }
```

Where

- *return-type* is the type of value that the function returns
- *name* is the name if the function
- *param-type1* is the type of the first parameter
- *param-name*1 is the name of the first parameter
- *param-type*2 is the type of the second parameter
- *param-name*2 is the name of the second parameter
- *body* is the code performing the task of the function

If a function does not return a value it must declare the return type as `void`.

The parameter declaration inside the parenthesis describes how many parameters must be passed when calling the function, the types of the parameters to be passed and names the parameters have inside the functions body code. The max-function above takes 2 chars parameters, named `a` and `b`.

The function `body` is the code executed when calling the function. In the body code the declared parameters can be used as variables and will have the values passed by the call.

The special statement `return` is used to return a value to the caller. The `return` statement exits the function immediately.

The following statement exits the function and returns the sum of values a and b.

```
return a+b;
```

The main() function

All KickC programs must have exactly one function called `main`. The `main` function is the starting point of the program. In KickC the main() function takes no parameters and returns no value.

The following is a very simple KickC program with a `main` function that turns the screen background color black and exits.

```
import "c64"

void main() {
      *BGCOL = BLACK;
}
```

When compiling the main-function generates a C64 BASIC program containing a single SYS command which starts the execution of the compiled KickC program.

The following is the KickAssembler code resulting from compiling the KickC program above. BasicUpstart is KickAssemblers way of creating a BASIC-program with a single SYS command.

```
.pc = $801 "Basic"
:BasicUpstart(main)
.pc = $80d "Program"
  .label BGCOL = $d021
  .const BLACK = 0
main: {
    lda #BLACK
    sta BGCOL
    rts
}
```

Comments

Two types of comments are supported in KickC. Anything inside a comment has no effect on the generated code.

Block comments, started with /* and ended with */ can span multiple lines. They can sometimes be useful for commenting out large parts of programs.

```
/* A multi-line
block comment */
```

Single line comments are started with double-slash // and ends at the next newline

```
// a single-line comment
```

Splitting code into multiple files

Like in C you can split code into multiple files by creating header-files (extension `.h`) and code-files (extension `.c`). The header-files typically contain only the interface of the module, ie. function declarations without body, `extern` variable declarations and `#defines`.

For one C-file to be able to use the functions/variables from another file you can include the header using

```
#include “other.h”
```

When including files the compiler first searches through the current folder where the file that has the `#include` statement is located, then it searches each library folder added by the `-I` / `-includedir` option to the compiler.

Include can be told to look in subfolders by prefixing the filename with a slash-separated path. Here is an example where graphics code files are located in a subfolder:

```
#include “graphics/character.h”
```

Using `#include <file.h>` instead of `#include “file.h”` will cause the compiler to not look for the file in the current directory, only looking in the search folders. This is useful when including libraries.

Unlike a normal C-compiler KickC does not support compiling each C-file individually and linking them later. Instead KickC insists on compiling all needed C-files in one go to be able to optimize better.

When you include a header-file using `#include` the compiler will try to automatically find the matching C-file. This is done by looking for a file with the same name, but extension .c in the current folder and each library folder added by the `-L` / `-libdir` option to the compiler.

You can also choose to pass multiple C-files directly to the compiler

```
kickc.sh main.c other.c
```

KickC also differs from standard C by automatically ensuring that any included file is only included once by automatically keeping track of which files have been included. This, combined with compiling all files in one go, means that with KickC you can skip the header-files and include C-files directly instead. This may be more convenient for you if you do not plan to port your code to other C-compilers.

Variables Directives

KickC has a number of directives can be used for controlling how a variable works. Variable directives are can be added before or after the type of the variable.

Const

The `const` directive is used to declare that a variable cannot be modified by the program. The declaration must also contain an assignment and the compiler will issue an error if the variable is assigned anywhere else.

```
const char SPRITES = 8;
```

For pointers the `const` directive can either be used to declare a constant pointer to a value or a pointer to a constant value. Which you declare is determined by the location of the `const` keyword relative to the `*`.

A constant pointer to a value means that the pointer cannot be modified, but the value that it points to can be modified. It is declared by placing the keyword after the *

```
char * const SCREEN = 0x0400 ;
```

A pointer to a const value means that the program is not allowed to modify whatever the pointer points to, but that the pointer itself can be modified. It is declared by placing the keyword before the *

```
const char * ROM = 0xA000 ;
```

Register

The `register` directive is used to instruct the compiler to optimize a variable, putting it into a CPU-register if possible. The compiler is quite good at optimizing register usage, so using this directive should not be needed.

```
char register i;
```

It is also possible to add a specific register in parenthesis to force the variable into a specific CPU-register. Using this directive can cause the compiler to fail if it is impossible to compile the program with the variable assigned to the register.

```
char register(X) i;
```

Align

The `align` directive is used to control the placement of arrays and strings in memory. For instance `align(0x40)` will ensure that the memory address where the data is placed is a multiple of 0x40 bytes. This can be useful when trying to optimize the performance of your program.

```
char align(0x100) sine[0x100];
```

Volatile

The `volatile` directive tells the compiler that the value of the variable might change at any time. The `volatile` keyword must be used for variables that are shared between code running "simultaneously". An example is when coding with interrupts (see the `interrupt` directive). The directive prevents the compiler from using all optimizations, where it assumes it can guess the value of the variable from the surrounding code. It will also ensure that the compiler does not produce code where the value of the variable is held in a register.

```
volatile char sprite_ypos;
```

For pointers the location of the `volatile` keyword relative to the `*` is used to distinguish between a volatile pointer and a pointer to volatile data. See `const` for a more thorough explanation.

Extern

The extern keyword on a variable specifies that this variable is defined somewhere else. The extern keyword is typically used in header-files, when a variable defined in the C-file should be usable by other C-files.

```
extern char cursor_onoff;
```

Export

The `export` directive tells the compiler that the value of a data-variable must be included in the resulting ASM even if it is never used anywhere and would normally be deleted by the optimizer. The `export` keyword is only usable for global variables containing data, typically arrays.
The directive prevent the compiler from deleting the variable during optimization.

```
export char MESSAGE[] = “Hello World!”;
```

Function Directives

KickC also has a few directives that instruct the compiler to treat functions in a specific way.

Inline

The `inline` function directive instructs the compiler to inline the whole function body everywhere the function is called. This can be used for optimizing your code since it allows the compiler to optimize the code of each function call independently, for instance by identifying constants in each call. It also saves the CPU cycles normally needed to call the function and return from it. The trade-off is that your program will compile into more bytes of code.

```
inline char sum( char a, char b) {
    return a+b;
}
```

Interrupt

The `interrupt` function directive is used for creating interrupt handler functions.

```
interrupt(kernel_keyboard) void irq() {
    *BGCOL = WHITE;
    *BGCOL = BLACK;
}
```

Setting up an interrupt on the C64 is done by assigning a pointer to an interrupt handler function to one of the interrupt vectors placed a specific address in the memory. Below is an example of setting up the kernal IRQ vector (at $314 in memory) to run the `irq()` function declared above. When setting up interrupts it is good practice to surround the code with the SEI/CLI instructions to prevent any interrupt from occuring during the setup itself.

```
#include <c64.h>

void main() {
```

```
    asm { sei }
    *KERNEL_IRQ = &irq;
    asm { cli }
}
```

Inside the parentheses of the interrupt directive the type of interrupt handler function is specified. This controls what kind of code is generated for saving/restoring register values and how the interrupt is exited:

- `kernel_keyboard` Interrupt served by the kernal called through `0x0314-5`. Will exit through the kernal using `0xea31`, which runs the normal kernal service routine that includes checking and handling keyboard input .
- `kernel_min` Interrupt served by the kernal called through `0x0314-5`. Will exit through the kernal using `0xea81`, which restores the registers and exits.
- `hardware_all` Interrupt served directly from hardware through `0xfffe-f` or `0xfffa-b`. Will exit through `RTI` and will save/restore all registers
- `hardware_stack` Interrupt served directly from hardware through `0xfffe-f` or `0xfffa-b`. Will exit through `RTI` and will save/restore all registers using the stack.
- `hardware_clobber` Interrupt served directly from hardware through `0xfffe-f` or `0xfffa-b`. Will exit through `RTI` and will save necessary registers based on a clobber analysis of the interrupt handler code.
- `hardware_none` Interrupt served directly from hardware through `0xfffe-f` or `0xfffa-b`. Will exit through `RTI` and will save/restore NO registers.

If your interrupt code needs to utilize global variables to communicate with other parts of the program or to store state between interrupt calls these variables should be declared as `volatile`.

Inline Assembler Code

Programs can include inline assembler code inside a function body. This can for instance be useful for interfacing to machine code such as the BASIC/KERNAL or for modifying processor flags (such as the interrupt flag or decimal flag).

Inline assembler is created using the `asm` statement with a body containing the assembler code. The following is an example setting the processor interrupt flag.

```
asm {
  sei
}
```

The assembler language usable within the curly braces is pretty limited standard syntax 6502 assembler using the same syntax as KickAssembler (and most classical 6502 assemblers). The following is supported:

- All normal 6502/6510 instructions and addressing modes
  - Immediate `lda #%10101010`
  - Absolute `eor 1024`
  - Zeropage `rol 2`
  - Relative `bne nxt`
  - Absolute indexed X `adc $2000,x`
  - Absolute indexed Y `cmp sintab,y`
  - Zeropage indexed X `sbc 2,x`
  - Zeropage indexed Y `stx $fe,y`
  - Zeropage indexed indirect X `lda ($20,x)`
  - Zeropage indirect indexed Y `ora (14),y`
  - Indirect `jmp ($1000)`
  - Implied (no operand) `tax`
- Labels
  - Normal labels `next:`
  - Multi labels `next!:`
- Data
  - Bytes `.byte $10, $20`

The parameters for instructions and the data bytes can be written as expressions supporting

- Literal numbers in decimal, binary or hexadecimal using the same syntax as KickC literal numbers eg. `1024, $3fff, %10101010`
- Literal characters eg. `'q'`
- Constant variables declared in the C-code eg. `SCREEN`
- Labels declared within the assembler code
- Math operators `+ - * / < > << >>`
- Parenthesis using `[` and `]` to avoid the assembler interpreting them as indirect addressing mode.

The ability to reference constant variables declared outside the assembler code allows the inline assembler to interact with data in the C-part of the program. The following is an example referencing the constant variable `SCREEN`.

```
const char* SCREEN = $400;

void main() {
      asm {
            lda #'c'
            sta SCREEN+40
      }
}
```

There is also support for referencing labels declared inside inline ASM in other functions using the special `.` operator (eg. `clearscreen.fillchar`). This makes it possible for ASM in one function to modify the code of ASM inside other scopes.

The KickC compiler understands the inline assembler code and attempts to optimize it during the compilation process. For instance it can analyze which registers are clobbered by the inline assembler, and optimize surrounding KickC-code register usage.

If your inline ASM contains a JSR call the compiler assumes that all registers are clobbered. However, it is possible to add a `clobbers` directive in parenthesis specifying exactly which registers are clobbered by your inline assembler code. Here is an example of inline assembler where the directive is used to specify that the JSR only clobbers the A- and X-registers.

```
void playMusic() {
      asm(clobbers “AX”) {
            jsr $1000
      }
}
```

Since the compiler understands the inline assembler it will also modify the assembler code if this leads to faster execution, for instance removing an immediate load-instruction that loads a value that the register is already guaranteed to contain.

Inline KickAssembler Code

The inline assembler code described above can be very useful, but only supports very rudimentary assembler features. The limitations allow the compiler to understand the assembler code and include it in optimizations.

If you need advanced assembler features in your code such as macros, loops or importing binary files or images it is possible to include inline KickAssembler code in your KickC program using the `kickasm` statement. The body of the `kickasm` statement must be enclosed in double curly braces and is passed directly to KickAssembler. The KickC compiler does not make any attempt to parse or understand the KickAssembler code. All advanced KickAssembler features are described in the manual here http://theweb.dk/KickAssembler.

The following is an example of inline KickAssembler code creating assembler for really fast screen clearing (1000 STA operations with no looping).

```
void clearscreen() {
      kickasm(uses screen) {{
            lda #0
            .for (var i = 0; i < 1000; i++) {
```

```
                sta screen+i
            }
        }}
}
```

Inline KickAssembler can reference constant variables declared in the surrounding C-code. To ensure that the KickC compiler knows that the inline KickAssembler uses a constant you should add a `uses` directive in parenthesis. This will ensure that KickC knows that the symbol is used, and for instance prevent the compiler from removing the symbol entirely if it is not used anywhere else.

Inline KickAssembler is also allowed outside function bodies. Here it allows utilizing KickAssemblers powerful macro language to initialize data tables.

```
const char sintab[] = kickasm(pc sintab) {{
      .fill $100, 127.5 + 127.5*sin(i*2*PI/256)
}};
```

To specify exactly where the resulting data bytes ends up in memory the `pc` directive can be specified in parenthesis. Here an example of generating a table with sinus values at a specific location.

```
const char* sintab = $1000;
kickasm(pc sintab) {{
      .fill $100, 127.5 + 127.5*sin(toRadians(i*360/256))
}}
```

It is also possible to use inline KickAssembler for loading pictures, music or other binary files and generating data bytes from these. When loading binary files in the inline KickAssembler code it is necessary to inform the KickC compiler using a `resource` directive within parentheses. This is needed because KickC may have to copy the used resource files to the output directory where the compiled assembler code is written. Here an example of including a sprite from a PNG image file and placing it at a specific memory address.

```
const char* SPRITE = $0c00;
kickasm(pc SPRITE, resource "balloon.png") {{
    .var pic = LoadPicture("balloon.png", List().add($000000, $ffffff))
    .for (var y=0; y<21; y++)
        .for (var x=0;x<3; x++)
            .byte pic.getSinglecolorByte(x,y)
}}
```

Comparison with standard C

Not supported/implemented

- Floating point types *float, double*
- Runtime multiplication (except constants) *a * b*
- Runtime division (except powers of 2) *a / b*
- Runtime modulo *a % b*
- Union *union { char b; int w; } u;*
- Array of arrays *char baa[4][4];*
- Function pointers w. param *void(char)*;*
- Function pointers w. return *char()*;*
- Recursive functions *char fib(byte n) { return fib(n-1)+fib(n-2)}*
- Alignof operator *char s = alignof(word);*
- Variadic functions *printf(const char* format, ...)*
- C preprocessor (see imports)
- C standard library (some bits are included, look in stdlib)

Limitations / Modifications

- Multiplication and division has limited support.
    - Multiplication and division is supported for constant values
    - Multiplication of a variable with an unsigned constant is supported and converted to shifts/additions.
    - There is no general 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).
- Alignment directive *align($100)*
- Register directive *register(X)*
- Inline assembler *asm { SEI CLD };*
- Main-function *void main() { … }*

Extensions

- Imports *import “print”*
- Forward referencing variables in the outer scope
- Ranged for-loops *for(char i: 0.. 10) { }*
- Word operator *unsigned int w = { hi, lo };*

- Lo/hi-byte operator *char lo = <w; <w = 12;*

The KickC Libraries

A limited C standard library:

- stdlib.h
- string.h
- time.h

Some libraries

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

3 Working with KickC

KickC Command Line Reference

Usage

kickc [-adehSvV] [-Ocoalesce] [-Si] [-Sl] [-vasmoptimize] [-vcreate] [-vfragment] [-vliverange] [-vloop] [-vnonoptimize] [-voptimize] [-vparse] [-vsequence] [-vunroll] [-vuplift] [-F=<*fragmentDir*>] [-fragment=<*fragment*>] [-o=<*asmFileName*>] [-odir=<*outputDir*>] [-Ouplift=<*optimizeUpliftCombinations*>] [-I=<*libDir*>]... [<kcFile>]

Description

Compiles a KickC source file, creating a KickAssembler source file. KickC is a compiler for a C-family language creating optimized and readable 6502 assembler code.

See https://gitlab.com/camelot/kickc for detailed information about KickC.

Parameters

[<kcFile>] The KickC source file to compile.

Options

Options:

-a Assemble the output file using KickAssembler. Produces a .prg file.

-calling=<*calling*> Configure calling convention. Default is __phicall. See #pragma calling

-cpu=<*cpu*> The target CPU. Default is 6502 with illegal opcodes. See #pragma cpu

-d Debug the assembled prg file using C64Debugger. Implicitly assembles the output.

-e Execute the assembled prg file using VICE. Implicitly assembles the output.

-E Only run the preprocessor. Output is sent to standard out.

-F, -fragmentdir=<*fragmentDir*> Path to the ASM fragment folder, where the compiler looks for ASM fragments.

-fragment=<*fragment*> Print the ASM code for a named fragment. The fragment is loaded/synthesized and the ASM variations are written to the output.

-h, --help Show this help message and exit.

-I, -includedir=<*includeDir*> Path to an include folder, where the compiler looks for included files. This option can be repeated to add multiple include folders.

-L, -libdir=<*libDir*> Path to a library folder, where the compiler looks for library files. This option can be repeated to add multiple library folders.

-o, -output=<*outputFileName*> Name of the output file. By default it is the same as the first input file with the proper extension.

-Ocache Optimization Option. Enables a fragment cache file.

-Ocoalesce Optimization Option. Enables zero-page coalesce pass which limits zero-page usage significantly, but takes a lot of compile time.

-odir=<*outputDir*> Path to the output folder, where the compiler places all generated files. By default the folder of the output file is used.

-Oliverangecallpath Optimization Option. Enables live ranges per call path optimization, which limits memory usage and code, but takes a lot of compile time.

-Oloophead Optimization Option. Enabled experimental loop-head constant pass which identifies loops where the condition is constant on the first iteration.

-Onoloophead Optimization Option. Disabled experimental loop-head constant pass which identifies loops where the condition is constant on the first iteration.

```
   -Onouplift          Optimization Option. Disable the register uplift
                       allocation phase. This will be much faster but
                       produce significantly slower ASM.
   -Ouplift=<optimizeUpliftCombinations>
                      Optimization Option. Number of combinations to test
                       when uplifting variables to registers in a scope. By
                       default 100 combinations are tested.
-S, -Sc                 Interleave comments with C source code in the generated
                       ASM.
   -Si                 Interleave comments with intermediate language code and
                       ASM fragment names in the generated ASM.
   -Sl                 Interleave comments with C source file name and line
                       number in the generated ASM.
-t, -target=<target>       The target system. Default is C64 with BASIC upstart.
                       See #pragma target
-T, -link=<linkScript>     Link using a linker script in KickAss segment format.
                       See #pragma link
-v                     Verbose output describing the compilation process
-V, --version           Print version information and exit.
   -var_model=<varModel>  Configure variable optimization/memory area. Default is
                       ssa_zp. See #pragma var_model
   -vasmoptimize         Verbosity Option. Assembler optimization.
   -vasmout             Verbosity Option. Show KickAssembler standard output
                       during compilation.
   -vcreate            Verbosity Option. Creation of the Single Static
                       Assignment Control Flow Graph.
   -vfragment           Verbosity Option. Synthesis of Assembler fragments.
   -vliverange          Verbosity Option. Variable Live Range Analysis.
   -vloop              Verbosity Option. Loop Analysis.
   -vnonoptimize         Verbosity Option. Choices not to optimize.
   -voptimize           Verbosity Option. Control Flow Graph Optimization.
   -vparse             Verbosity Option. File Parsing.
   -vsequence            Verbosity Option. Sequence Plan.
   -vsizeinfo          Verbosity Option. Compiler Data Structure Size
                       Information.
   -vunroll             Verbosity Option. Loop Unrolling.
   -vuplift            Verbosity Option. Variable Register Uplift Combination
                       Optimization.
   -Warraytype           Warning Option. Non-standard array syntax produces a
                       warning instead of an error.
   -Wfragment            Warning Option. Missing fragments produces a warning
                       instead of an error.
```

The Coding Workflow / Related Tools

- Assembling
- Executing (Emulators or The Real Thing)
- Debugging
- 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,)

4 The Compiler Architecture

The KickC compiler uses the following phases during compilation

1. **Loading and Parsing**
   Loads the main source file and recursively loads all imported source files. Parses the files creating a parse tree representation.
2. **Creating Control Flow Graph and Symbol Table**
   Converts the parse tree to a symbol table containing all variables and functions and a control flow graph in static single assignment form (SSA form) [1]. SSA form consists of a control flow graph which models all possible execution paths. Each control flow block contains a number of statements (mostly assignments). Expressions are broken into a number of simple assignments to intermediate variables ensuring that each SSA statement only handles a single operator. In SSA form variables are also broken into multiple versions to ensure that each variable is assigned a value exactly once in the entire program. Finally transitions between blocks are handled through so called Phi-functions that describe how variable versions are mapped when execution flows from one block to another. SSA form has huge advantages in making a lot of optimizations easier to program.
3. **Optimizing the SSA Control Flow Graph**
   Optimizes the control flow graph by repeatedly calling 20+ micro-passes. Each micro-pass examines the control flow graph and can perform a specific type of

[1] https://en.wikipedia.org/wiki/Static_single_assignment_form

optimization by modifying the graph. The optimizer keeps cycling through the micro-passes until none of them can perform any more optimizations.

4. **Control Flow Graph Analysis**
   Performs several analyses of the program in preparation of register allocation. This includes call graph analysis (which functions call each other), variable live range analysis (where is a variable defined, where is it used and where is it alive, meaning that it will be used at a later point) and loop analysis (which loops exist in the code, which loops nest each other, how deep is each loop).
5. **Register Allocation**
   Allocates registers and memory locations to all variables. Initially in this phase the SSA variables are grouped together into groups that will benefit from having the same register allocation. The technique used for this is called PhiLifting and PhiMemCoalesce[2]. The allocation is then essentially done by trying out a lot of different register combinations and choosing the one that generates the best assembler code (uses fewest cycles). The number of different combinations tested can be controlled by the compiler option `-Ouplift=nnnn`
6. **Assembler Code Generation using Assembler Fragments**
   6502 Assembler code is generated by converting each SSA statement to assembler code using the chosen register allocation. Because SSA statements can only express pretty simple operations ASM generation is essentially done by having a large library of ASM fragment files containing the ASM code needed for a specific SSA statement with a specific register allocations. This library of ASM fragments is stored in the *fragment* folder in the compiler installation. The Fragment sub-system of the compiler loads fragments from this folder, but also uses a bunch of rules for synthesizing more advanced fragments from simpler ones. Even with the current 500+ fragment-files and 150+ synthesis-rules the compiler still occasionally runs into SSA statements it does not know how to create assembler for. If you encounter this problem you can fix it yourself by adding the right fragment-file in the *fragments*-folder. The fragment sub-system is described in more detail below to help you do this in case you need to.
7. **Assembler Code Optimization**
   Finally the compiler performs optimization of the generated assembler code. This is also done by repeatedly calling a bunch of micro-passes that each knows how to perform a simple Assembler Code Optimization. The optimizer keeps cycling through the micro-passes until none of them can perform any more optimizations. The assembler optimizations include eliminating redundant register loads (eg.LDA #0 when A is already zero), replacing double jumps with jumps straight to the destination, removing unused labels, eliminating etc.

[2] http://compilers.cs.ucla.edu/fernando/projects/soc/reports/short_tech.pdf

Assembler Fragment Sub System

Adding missing assembler fragments

The format for the values in the fragment name is:

1. "v" value / "p" pointer
2. "b" byte / "w" word / "d" dword
3. "u" unsigned / "s" signed / "o" boolean
4. "aa" A-register / "xx" X-register / "yy" Y-register / "z1" zeropage {z1} / "c1" constant {c1}.

When {c1} is used for values it is an immediate value, eg. in vbuc1, {c1} is a constant unsigned byte value.

When c1 is used for pointers it is an address in main memory, eg. in pbsc1, {c1} is a constant pointer to a signed byte. This means that {c1} is effectively an address in main memory.

vwuz1_gt_vbsc1_then_la1 for example means [if] variable word unsigned zero-page value {z1} is greater than variable byte signed constant {c1} then [goto] label {la1}.

Fragments can use $ff as temporary storage (and $fe if 2 addresses are needed).

Before adding the fragment try compiling with the -vfragment flag. It will show you all the different fragments that the compiler is considering. You only need to implement one and then the fragment synthesizer can create what it needs from that.

If you are wondering how a specific fragment looks you can ask the compiler using the `-fragment` flag. The following command will show all the different variations of assembler the compiler can use when needing to assign an unsigned byte in a zeropage variable {z1} to the value found in a table of unsigned bytes {c1} indexed by another unsigned byte variable on zeropage {z2}.

```
kickc.sh -fragment vbuz1=pbuc1_derefidx_vbuz2
```

You do not need to restore any register values in fragments. In fact that is part of the optimizers strength.

The compiler analyses the ASM in each fragment and determines which registers are clobbered. When allocating variables to registers it avoids any allocation where a fragment clobbers a register holding a variable value that is needed later in the code. So it will avoid holding a value in A that is needed after any fragment that clobbers A - and will instead look at different options (X, Y or on zeropage).

This produces much better ASM than each fragment restoring register values since it allows the compiler flexibility in choosing the register/zeropage allocation that minimises the number of cycles the code consumes.

This is further improved by the compiler treating each assignment to a variable as a separate variable - meaning it often ends up choosing to hold much used variables in different registers or on zero page for different parts of the code.

Overall KickC has the following compile process:

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

The Control Flow Graph uses Single Static Assignment (where each variable is only assigned once - and where statements are mostly assignments with 1 or 2 arguments and an operator.) See https://en.wikipedia.org/wiki/Static_single_assignment_form

The SSA of KickC has been specifically designed to inline pointer derefs or indexed pointer derefs - because this improves the ASM that it generates significantly.

The SSA Optimization optimizes the control flow graph by repeatedly calling 20+ micro-passes. This is where constants are propagated, unused code is removed and so forth.

Register Allocation decides which variables to put into registers (A/X/Y) and which to store on zeropage. This is done based on variable range live range analysis, using PHI-lifting and register clobber analysis - plus testing a lot of combinations to find the solution using the fewest CPU cycles. See http://compilers.cs.ucla.edu/.../soc/reports/short_tech.pdf

The SSA-statements are then turned into ASM using the fragment system. Each SSA-statement combined with a specific register allocation becomes a fragment.

An example is the fragment
vbuz1=vbuz1_plus_pbuc1_derefidx_vbuxx
which matches a SSA-statement like
digit#1 = digit#3 + *(UTOA10_VAL#0 + i#2)
where digit#1 and digit#3 are unsigned bytes allocated to the same zeropage-address (which is possible because digit#3 is never used again after this assignment so digit#1 can overwrite it), UTOA10_VAL#0 is a table of unsigned byte values stored in memory at a constant address and i#2 is an index into the table allocated to the X-register.

The fragment sub-system has two parts. First around 900 files with specific fragment ASM-code. Second a fragment synthesizer that can make more complex fragments from simpler fragments.

The fragment above is actually synthesized from vbuaa=vbuaa_plus_vbuz1 (which is just adding a value on zeropage to the A-register) using different synthesis rules.

synthesized vbuz1=vbuz1_plus_pbuc1_derefidx_vbuxx < vbuz1=pbuc1_derefidx_vbuxx_plus_vbuz1 < vbuaa=pbuc1_derefidx_vbuxx_plus_vbuz1 < vbuaa=vbuz1_plus_pbuc1_derefidx_vbuxx < vbuaa=vbuz1_plus_vbuaa < vbuaa=vbuaa_plus_vbuz1 - clobber:A cycles:12.5
lda {c1},x
clc
adc {z1}
sta {z1}

Finally the resulting ASM is optimized using peephole optimization.