

Kick Assembler V3.21 Reference Manual



By Mads Nielsen

Index

1	Introduction.....	4
2	Getting Started.....	5
2.1	Running the Assembler using Java.....	5
2.2	An Example Interrupt.....	5
2.3	Configuring the Assembler.....	6
3	Basic Assembler Stuff.....	8
3.1	6510 Commands.....	8
3.2	Addressing Modes / Argument Types.....	10
3.3	Number formats.....	11
3.4	Labels and Multi Labels.....	11
3.5	Memory and Data Directives.....	12
3.6	The Import Directive.....	14
3.7	Comments.....	15
3.8	User Console Output (.print, printnow and .error).....	15
4	Introducing the Script Language.....	17
4.1	Expressions.....	17
4.2	Variables, Constants and User Defined Labels.....	17
4.3	Scoping.....	19
4.4	Numeric Values.....	20
4.5	Parentheses.....	21
4.6	String Values.....	21
4.7	Char Values.....	22
4.8	The Math Library.....	24
4.9	List Values.....	25
4.10	Working with Mutable Values.....	25
4.11	Hashtable Values.....	26
4.12	Vector and Matrix Values.....	27
5	User Defined Structures.....	28
6	Branching and Looping.....	29
6.1	Boolean Values.....	29
6.2	.if.....	29
6.3	Questionmark if's.....	30
6.4	.for.....	31
6.5	Optimization Considerations when using Loops.....	31
7	Macros, Functions and Pseudo Commands.....	33
7.1	Macros.....	33
7.2	Functions.....	34
7.3	Pseudo Commands.....	35
8	Name Spaces.....	37
8.1	The Namespace Directive.....	37
8.2	File Namespaces.....	38
8.3	Label Namespaces (Autonamespacing).....	39
8.4	Accessing Local Labels of Macros and Pseudocommands.....	40
9	Special Features.....	41
9.1	Creating a Basic Upstart Program.....	41
9.2	Opcode Constants.....	41
9.3	Colour Constants.....	42

9.4	Exporting Labels.....	43
9.5	Import of Binary Files	43
9.6	Import of SID Files	45
9.7	Converting Graphics.....	47
9.8	Writing to User Defined Files.....	48
9.9	Passing Command Line Arguments to the Script.....	49
9.10	Making 3D Calculations	50
10	Testing	53
10.1	Asserting expressions	53
10.2	Asserting errors in expressions.....	53
10.3	Asserting code	53
10.4	Asserting errors in code	54
11	Some technical details	55
11.1	The flexible parse algorithm.....	55
11.2	Recording of side effects	55
11.3	Function Mode and Asm Mode	55
11.4	Invalid value calculations	55
12	Going from Kick Assembler 2.x to 3.x.....	57
13	Command line options.....	58

1 Introduction

Welcome to this third version of Kick Assembler. The first version was a normal 6510 cross assembler. The second version combined the assembler with a script language, giving you the opportunity to write programs that generate data for the assembler code. This could be data such as sine waves, coordinates for a vector object, or graphic converters. Writing small data generating programs directly in you assembler source code is much handier than writing them in external languages like Java or C++.

So why version 3? If you take a look at this manual, you will see that the functionality of Kick Assembler 3 is almost similar to that of Kick Assembler 2. The syntax, the functions and the directives are the same as in version 2. However, the code of almost every class in the assembler has been rewritten or modified. Three fundamental changes have been made. Kick assembler 3 uses a new flexible pass algorithm for assembling, and it handles side effects and invalid values in a more flexible and efficient way that prevent errors during assembling. If you want to know more about this then read the chapter 'Some technical details'.

If you have previously used Kick Assembler 2.x you might want to read the chapter 'Going from Kick Assembler 2.x to 3.x'

A lot of people have contributed with valuable comments and suggestions by mail and on CSDB. Thanks guys. Your feedback is greatly appreciated. I would especially like to thank Martin 'Cruzer' Kristensen for proofreading and testing the assembler; Gunni 'Dagnet' Rode and Bastiaan 'Mace' for proofreading; Gerwin Klein for doing JFlex (the lexical analyser used for this assembler); Scott Hudson, Frank Flannery and C. Scott Ananian for doing CUP (The parser generator).

I would like to hear from people that use Kick Assembler so do not hesitate to write your comments to kickassembler@no.spam.theweb.dk (<- Remove no.spam. for real address).

I wish you happy coding..

2 Getting Started

This chapter is written to quickly get you started using Kick Assembler. The details of the assembler's functionalities will be presented later.

2.1 Running the Assembler using Java

Kick Assembler run on any platform with Java5.0 or higher installed. Java can be downloaded for free on Javas website (<http://java.com/en/download/index.jsp>). To assemble the file myCode.asm simply go to a command prompt and write:

```
java -jar kickass.jar myCode.asm
```

And that's it.

Having problems with java? Some Windows users found that java couldn't be reached from the command prompt after installation. If this is the case you have to insert it in your path environment variable. You can test it by writing:

```
java -version
```

Java will now display the java version if it's correctly installed.

2.2 An Example Interrupt

Below is a little sample program to quickly get you started using Kick Assembler. It sets up an interrupt, which play some music. It shows you how to use non-standard features such as the .pc directive, comments, how to use macros and include external files. This should be enough to get you (kick) started.

```

//-----
//-----
//                               Simple IRQ
//-----
//-----
.pc = $4000 "Main Program"

        lda #$00
        sta $d020
        sta $d021
        lda #$00
        jsr $1000 // init music
        sei
        lda #<irq1
        sta $0314
        lda #>irq1
        sta $0315
        asl $d019
        lda #$7b
        sta $dc0d
        lda #$81
        sta $d01a
        lda #$1b
        sta $d011
        lda #$80
        sta $d012
        cli
this:   jmp this
//-----
irq1:
        asl $d019
        :SetColor(2)
        jsr $1003 // play music
        :SetColor(0)
        pla
        tay
        pla
        tax
        pla
        rti

//-----
.pc=$1000 "Music"
.import binary "ode to 64.bin"

//-----
// A little macro
.macro SetBorderColor(color) {
        lda #color
        sta $d020
}

```

2.3 Configuring the Assembler

Kick Assembler has a lot of command line options (a summary is given in the last chapter ‘Command Line Options’). For example, if you assemble your program with the `--showmem` option you will get a memorymap shown after assembling:

```
java -jar kickass.jar -showmem myCode.asm
```

By placing a file called KickAss.cfg in the same folder as the KickAss.jar, you can set command line options that are used at every assembling. Lets say you always wants to have shown a memorymap after assembling and then have the result executed in the C64 emulator VICE. Then you write the following in the KickAss.cfg file:

```
-showmem  
-execute "c:/c64/winvice/x64.exe -confirmexit"
```

(Replace c:/c64/winvice/ with a path that points to the vicefolder on your machine)

3 Basic Assembler Stuff

This chapter describes the mnemonics and the basic directives that are not related to the script language.

3.1 6510 Commands

In Kick Assembler you can write assembler mnemonics the traditional way:

```
lda #0
sta $d020
sta $d021
```

However, it ignores format statements such as newline and tabs so you can format your program in any way you like. If you wish, you can write your entire program in one line:

```
lda #0 sta $d020 sta $d021
```

This comes in handy when using the script language. Kick Assembler supports all opcodes, also the illegal ones. A complete list of commands and their opcodes in the each mode is shown here:

Mnemonic	noarg	imm	zp	zpx	zpy	izx	izy	abs	Abx	aby	ind	rel
adc		\$69	\$65	\$75		\$61	\$71	\$6d	\$7d	\$79		
ahx							\$93			\$9f		
alr		\$4b										
anc		\$0b										
anc2		\$2b										
and		\$29	\$25	\$35		\$21	\$31	\$2d	\$3d	\$39		
arr		\$6b										
asl	\$0a		\$06	\$16				\$0e	\$1e			
axs		\$cb										
bcc												\$90
bcs												\$b0
beq												\$f0
bit			\$24	\$34				\$2c	\$3c			
bmi												\$30
bne												\$d0
bpl												\$10
brk	\$00											
bvc												\$50
bvs												\$70
clc	\$18											
cld	\$d8											
cli	\$58											
clv	\$b8											
cmp		\$c9	\$c5	\$d5		\$c1	\$d1	\$cd	\$dd	\$d9		
cpx		\$e0	\$e4					\$ec				
cpy		\$c0	\$c4					\$cc				
dcp			\$c7	\$d7		\$c3	\$d3	\$cf	\$df	\$db		
dec			\$c6	\$d6				\$ce	\$de			
dex	\$ca											
dey	\$88											
eor		\$49	\$45	\$55		\$41	\$51	\$4d	\$5d	\$59		

inc			\$e6	\$f6				\$ee	\$fe			
inx	\$e8											
iny	\$c8											
isc			\$e7	\$f7		\$e3	\$f3	\$ef	\$ff	\$fb		
jmp								\$4c			\$6c	
jsr								\$20				
las										\$bb		
lax		\$ab	\$a7		\$b7	\$a3	\$b3	\$af		\$bf		
lda		\$a9	\$a5	\$b5		\$a1	\$b1	\$ad	\$bd	\$b9		
ldx		\$a2	\$a6		\$b6			\$ae		\$be		
ldy		\$a0	\$a4	\$b4				\$ac	\$bc			
lsr	\$4a		\$46	\$56				\$4e	\$5e			
nop	\$ea											
ora		\$09	\$05	\$15		\$01	\$11	\$0d	\$1d	\$19		
pha	\$48											
php	\$08											
pla	\$68											
plp	\$28											
rla			\$27	\$37		\$23	\$33	\$2f	\$3f	\$3b		
rol	\$2a		\$26	\$36				\$2e	\$3e			
ror	\$6a		\$66	\$76				\$6e	\$7e			
rra			\$67	\$77		\$63	\$73	\$6f	\$7f	\$7b		
rti	\$40											
rts	\$60											
sax			\$87		\$97	\$83		\$8f				
sbc		\$e9	\$e5	\$f5		\$e1	\$f1	\$ed	\$fd	\$f9		
sbc2		\$eb										
sec	\$38											
sed	\$f8											
sei	\$78											
shx										\$9e		
shy									\$9c			
slo			\$07	\$17		\$03	\$13	\$0f	\$1f	\$1b		
sre			\$47	\$57		\$43	\$53	\$4f	\$5f	\$5b		
sta			\$85	\$95		\$81	\$91	\$8d	\$9d	\$99		
stx			\$86		\$96			\$8e				
sty			\$84	\$94				\$8c				
tas										\$9b		
tax	\$aa											
tay	\$a8											
tsx	\$ba											
txa	\$8a											
txs	\$9a											
tya	\$98											
xaa		\$8b										

DTV opcodes are also supported. To use these you have to use the `-dtv` option at the command line when running Kick Assembler. The DTV commands are:

Mnemonic	noarg	imm	zp	zpx	zpy	izx	izy	abs	Abx	aby	ind	rel
bra												\$12
sac		\$32										
sir		\$42										

3.2 Addressing Modes / Argument Types

Kick Assembler uses the traditional notation for addressing modes / argument types:

Mode	Example
No argument	Nop
Immediate	lda #\$30
Zeropage	lda \$30
Zeropage,x	lda \$30,x
Zeropage,y	ldx \$30,y
Indirect zeropage,x	lda (\$30,x)
Indirect zeropage,y	lda (\$30),y
Abolute	lda \$1000
Absolute,x	lda \$1000,x
Absolute,y	lda \$1000,y
Indirect	jmp (\$1000)
Relative to program counter	bne loop

An argument is converted to its zeropage mode if possible. This means that `lda $0030` will generate an `lda` command in its zeropage mode* .

You can force the assembler to use a given addressing mode by using mnemonic extensions like this:

```
lda.a $0030 // Uses absolute mode
sta.z label // Uses zeropage mode
ldx.im $10 // Equal to lda #$10
label:
```

Here is a list of the extensions:

Ext	Mode	Example
im, imm	Immediate	
z, zp	Zeropage	ldx.z \$1234
zx, zpx	Zeropage,x	lda.zpx table
zy, zpy	Zeropage,y	
izx, izpx	Indirect zeropage,x	
izy, izpy	Indirect zeropage,y	
a, abs	Abolute	ldx.a \$0010
ax, absx	Absolute,x	lda.absx \$1234
ay, absy	Absolute,y	
i, ind	Indirect	jmp.i \$1000
r, rel	Relative to program counter	

* If the argument is unknown (eg. an unresolved label) in the first pass, the assembler will assume it's a 16 bit value and not reduce it.

3.3 Number formats

Kick Assembler supports the standard number formats:

Prefix	Format	Example
	Decimal	lda #42
\$	Hexadecimal	lda #\$2a, lda #\$FF
%	Binary	lda #%101010

3.4 Labels and Multi Labels

Label declarations in Kick Assembler end with ':' and have no postfix when referred to, as shown in the following program:

```
loop: inc $d020
      inc $d021
      jmp loop
```

Kick Assembler also supports multi labels, which are labels that can be declared more than once. These are useful to prevent name conflicts between labels. A multi label starts with a '!' and when you reference it you have to end with a '+' to refer to the next multi label or '-' to refer to the previous multi label:

```
      ldx #100
!loop: inc $d020
      dex
      bne !loop- // Jumps to the previous instance of !loop

      ldx #100
!loop: inc $d021
      dex
      bne !loop- // Jumps to the previous instance of !loop
```

OR

```
      ldx #10
!loop: jmp !+ // Jumps over the two next nops to the ! label
      nop
      nop
!: jmp !+ // Jumps over the two next nops to the ! label
      nop
      nop
!: dex
      bne !loop- // Jumps to the previous !loop label
```

Another way to avoid conflicting variables is to use user defined scopes, which are explained in the Scopes section of the script chapter.

A '*' returns the value of the current memory location so instead of using labels you can write your jumps like this:

With '*' reference:	With label:
jmp *	this: jmp this
inc \$d020 inc \$d021 jmp *-6	!loop: inc \$d020 inc \$d021 jmp !loop-

3.5 Memory and Data Directives

The `.pc` directive is used to set the program counter. A program should always start with a `.pc` directive to tell the assembler where to put the program. Here are some examples of use:

```
.pc = $1000 "Program"
    ldx #10
!loop: dex
    bne !loop-
    rts

.pc = $4000 "Data"
    .byte 1,0,2,0,3,0,4,0

.pc = $5000 "More data"
    .text "Hello"
```

The last argument is optional and is used to name the memory block created by the directive. When using the `'-showmem'` option when running the assembler a memory map will be generated that displays the memory usage and block names. The map of the above program looks like this:

```
Memory Map
-----
$1000-$1005 Program
$4000-$4007 Data
$5000-$5004 More data
```

By using the `virtual` option on the `.pc` directive you can declare a memory block that is not saved in the resulting file.

```
.pc = $0400 "Data Tables 1" virtual
table1: .fill $100,0
table2: .fill $100,0

.pc = $0400 "Data Tables 2" virtual
table3: .fill $150,0
table4: .fill $100,0

.pc = $1000 "Program"
    ldx #0
    lda table1,x
    ...
```

Note that virtual memory blocks can overlap other memory blocks. They are marked with an asterisk in the memory map.

```
Memory Map
-----
*$0400-$05ff Data Tables 1
*$0400-$064f Data Tables 2
$1000-$1005 Program
```

Since virtual memory blocks aren't saved, the above example will only save the memory from \$1000 to \$1005.

With the `.align` directive, you can align the program counter to a given interval. This is useful for optimizing your code as crossing a memory page boundary yields a penalty of one cycle for memory referring commands. To avoid this, use the `.align` command to align your tables:

```
.pc = $1000 "Program"
    ldx #1
    lda data,x
    rts

.pc = $10ff          //Bad place for the data
.align $100         //Alignment to the nearest page boundary saves a cycle
data: .byte 1,2,3,4,5,6,7,8
```

The `.byte`, `.word`, `.dword` and `.text` directives are used to generate byte, word (one word= two bytes), dword (double word = 4 bytes) and text data as in standard 6510 assemblers (see previous example).

With the `.fill` directive you can fill a section of the memory with bytes. It works like a loop and automatically sets the variable `i` to the byte number.

```
.fill 5, 0          // Generates byte 0,0,0,0,0
.fill 5, i          // Generates byte 0,1,2,3,4
.fill 256, 127.5 + 127.5*sin(toRadians(i*360/256)) // Generates a sine curve
```

In case you want your code placed at position \$1000 in the memory but want it assembled like it was placed at \$2000, you can use the `.pseudopc` directive:

```
.pc = $1000 "Program to be relocated at $2000"
.pseudopc $2000 {
loop:    inc $d020
        jmp loop    // Will produce jmp $2000 instead of jmp $1000
}
```

Here is an overview of the memory and data directives:

Form	Example	Description
<code>.pc = <expr> ["name"]</code>	<code>.pc = \$1000 "Program"</code>	Sets the program counter.
<code>.align <expr></code>	<code>.align \$100</code>	Aligns the program counter to a given interval.
<code>.byte <expr list></code>	<code>.byte 1,2,3</code>	Generates byte data.

<code>.word <expr list></code>	<code>.word 3,2,1</code> <code>.word \$1122</code>	Generates word data (2 bytes).
<code>.dword <expr list></code>	<code>.dword 1,2,3</code> <code>.dword \$11223344</code>	Generates double word data (4 bytes).
<code>.text <expr></code>	<code>.text "Hello"</code>	Generates text data.
<code>.fill <expr>, <expr></code>	<code>.fill 256, 0</code> <code>.fill 256, i</code>	Generates a number of bytes, given by the first expression, with the data given by the second expression. The variable <code>i</code> is set to the byte number in the second expression.
<code>.pseudopc <expr></code> <code>{...}</code>	<code>.pseudopc \$2000</code> <code>{...}</code>	Assembles code as if it was placed at a different location.

3.6 The Import Directive

With the `.import` directive you can import external files into your source. You can import source, binary, C64, and text files:

```
// Import and assemble the sourcefile 'standardlibrary.asm'
.import source "StandardLibrary.asm"

// import the bytes from the file 'music.bin'
.import binary "Music.bin"

// Import the bytes from the c64 file 'charset.c64'
// (Same as binary but skips the first two address bytes)
.import c64 "charset.c64"

// Import the chars from the text file
// (Converts the bytes as a .text directive would do)
.import text "scroll.txt"
```

When Kick Assembler searches for a file, it first look in the current directory. Afterwards it looks in the directories supplied by the `-libdir` parameter when running the assembler. This enables you to create standard libraries for files you use in several different sources. A command line could look like this:

```
java -jar kickass.jar myProgram.asm -libdir ..\music -libdir c:\code\stdlib
```

If you build source code libraries you might want to ensure that the library is only included once in your code. This can be done by placing a `.importonce` directive in the top of the library file:

```
File1:
.importonce
.print "This will only be printed once!"

File2:
.import source "file1.asm" // This will import file1
.import source "file1.asm" // This will not import anything
```

3.7 Comments

Comments are pieces of the program that are ignored by the assembler. Kick Assembler supports line and block comments known from languages such as C++ and Java. When the assembler sees `/**` it ignores the rest of that line. C block comments ignores everything between `/*` and `*/`.

```
/*-----  
This little program is made to demonstrate comments  
-----*/  
    lda #10  
    sta $d020    // This is also a comment  
    sta /* Comments can be placed anywhere */ $d021  
    rts
```

Traditional 6510 asm line comments (`;`) are not supported since the semicolon is used in for-loops in the script language.

Type	Form	Description
C line comments	<code>// This is also a comment</code>	Ignores the rest of the line
C block comments	<code>/* This is a block comment */</code>	Ignores everything between <code>/*</code> and <code>*/</code>

3.8 User Console Output (`.print`, `.printnow` and `.error`)

With the `.print` directive you can output text to the user while assembling. For example:

```
.print "Hello world"  
.var x=2  
.print "x="+x
```

This will give the following output from the assembler:

```
parsing  
flex pass 1  
Output pass  
    Hello world  
    x=2.0
```

Notice that the output is given during the output pass. You can also print the output immediately with the `.printnow` command. This is useful for debugging script where errors prevent the execution of the output pass. The `.printnow` command will print the output in each pass, and in some passes the output might be incomplete due to lack of information. In the following example we print a label that isn't resolved in the first pass:

```
.printnow "loop=$" + toHexString(loop)  
  
.pc = $1000  
loop: jmp loop
```

This will give the following output:

```
parsing
flex pass 1
  loop=$<<Invalid String>>
flex pass 2
  loop=$1000
Output pass
```

If you detect an error while assembling, you can use the `.error` directive to terminate the assembling and display an error message:

```
.var width = 45
.if (width>40) .error "width can't be higher than 40"
```


4 Introducing the Script Language

In this chapter the basics of the script language is introduced. Late chapters will deal with more advanced areas.

4.1 Expressions

Kick assembler has a built in mechanism for evaluating expressions. An example of an expression is $25+2*3/x$. Expressions can be used in many different contexts, for example to calculate the value of a variable or to define a byte:

```
lda #25+2*3/x
.byte 25+2*3/x
```

Standard assemblers can only calculate expressions based on numbers, while Kick Assembler can evaluate expressions based on many different types like: Numbers, Booleans, Strings, Lists, Vectors, and Matrixes. So, if you want to calculate an argument based on the second value in a list you write:

```
lda #35+myList.get(1) // 1 because first element is number 0
```

Or perhaps you want to generate your argument based on the x-coordinate of a vector:

```
lda #35+myVector.getX()
```

Or perhaps on the basis of the x-coordinate on the third vector in a list:

```
lda #35+myVectorList.get(2).getX()
```

I think you get the idea by now. Kick Assembler's evaluation mechanism is much like those in modern programming languages. It has a kind of object oriented approach, so calling a function on a value(/object) executes a function specially connected to the value. Operators like $+$, $-$, $*$, $/$, $==$, $!=$, etc., are seen as functions and are also specially defined for each type of value.

In the following chapters, a detailed description of how to use the value types and functions in Kick Assembler will be presented.

4.2 Variables, Constants and User Defined Labels

Before you can use variables you have to declare them. You do this with the `.var` directive:

```
.var x=25
lda #x // Gives lda #25
```

If you want to change `x` later on you write:

```
.eval x=x+10
lda #x // Gives lda #35
```

This will increase x by 10. The `.eval` directive is used to make Kick Assembler evaluate expressions. In fact, the `.var` directive above is just a convenient shorthand of `'eval var x =25'`, where `'var'` is subexpression that declares a variable (this will come in handy later when we want to define variables in for-loops).

Two other shorthands exist: the `++` and the `--` operator, which automatically calls a referenced variable with `+1` or `-1`. For example:

```
.var x = 0
.eval x++          // Gives x=x+1
.eval x--          // Gives x=x-1
```

Experienced users of modern programming languages will know that assignments return a value, e.g. `x = y = z = 25` first assigns 25 to z, which returns 25 that is assigned to y, which returns 25 that is assigned to x. Kick Assembler supports this as well. Notice that the `++` and `--` works as real `++` and `--` postfix operators, which means that they return the original value and not the new (Ex: `.eval x=0 .eval y=x++`, will set x to 1 and y to 0)

You can also declare constants:

```
.const c=1          // Declares the constant c to be 1
.eval const pi=3.1415 // Declares the constant pi using the eval form
.const name = "Camelot" // Constants can assume any value, for example string
```

A constant can't be assigned a new value, so `.eval pi=22` will generate an error. Note that not all values are immutable. If you define a constant that points to a list, the content of the list can still change. If you want to make a mutable value immutable, you can use its `lock()` function, which will lock it's content:

```
.const immutableList = List().add(1,2,3).lock()
```

After this you will get an error if you try to add an element or modify existing elements.

With the `.enum` statement you can define enumerations, which are series of constants:

```
.enum {singleColor, multiColor} // Defines singleColor=0, multiColor=1
.enum {effect1=1,effect2=2,end=$ff} // Assigns values explicitly
.enum {up,down,left,right, none=$ff} // You can mix implicit and explicit
                                     // assignment of values
```

Variables and constants can only be seen after they are declared while labels can be seen in the entire scope. You can define a label with the `.label` directive like you define variables and constants:

```
// This fails
inc myLabel1
.const myLabel1 = $d020

// This is ok
```

```
inc myLabel2
.label myLabel2 = $d020
```

4.3 Scoping

You can limit the scope of you variables and labels by defining a user defined scope. This is done by `{..}`. Everything between the brackets is defined in a local scope and can't be seen from the outside.

```
Function1: {
    .var length = 10
    ldx #0
    lda #0
loop:    sta table1,x
        inx
        cpx #length
        bne loop
}

Function2: {
    .var length = 20 // doesn't collide with the previous 'length'
    ldx #0
    lda #0
loop:    sta table2,x // the label doesn't collide with the previous 'loop'
        inx
        cpx #length
        bne loop
}
```

Scopes can be nested as many times as you wish as demonstrated by the following program:

```
.var x = 10
{
    .var x=20
    {
        .print "X in 2nd level scope read from 3rd level scope is " + x
        .var x=30
        .print "X in 3rd level scope is " + x
    }
    .print "X in 2nd level scope is " + x
}
.print "X in first level scope is " + x
```

The output of this is:

```
X in 2nd level scope read from 3rd level scope is 20.0
X in 3rd level scope is 30.0
X in 2nd level scope is 20.0
X in first level scope is 10.0
```

4.4 Numeric Values

Numeric values are numbers covering both integers and floats. Standard numerical operators (+, -, *, and /) work as in standard programming languages. You can combine them with each other and they will obey the standard precedence rules. Here are some examples:

```
25+3
5+2.5*3-10/2
charmem + y * $100
```

In practical use they can look like this:

```
.var charmem = $0400
    ldx #0
    lda #0
loop: sta charmem + 0*$100,x
      sta charmem + 1*$100,x
      sta charmem + 2*$100,x
      sta charmem + 3*$100,x
      inx
      bne loop
```

You can also use bitwise operators to perform and, or, exclusive or, and bit shifting operations.

```
.var x=$12345678
.word x & $00ff, [x>>16] & $00ff // (gives .word $0078, $0034)
```

Special for 6510 assemblers are the high and low-byte operators (>,<) that are typically used like this:

```
lda #<interrupt1
sta $0314
lda #>interrupt1
sta $0315
```

These are also available in Kick Assembler. Here is a list of numeric operators that returns a numeric value:

Name	Operator	Examples	Description
Unary minus	-		Inverts the sign of a number
Plus	+	10+2 = 12	Adds two numbers
Minus	-	10-8=2	Subtracts two numbers
Multiply	*	2*3 =6	Multiply two numbers
Divide	/	10/2 = 5	Divides two numbers
High byte	>	>\$1020 = \$10	Returns the second byte of a number
Low byte	<	<\$1020 = \$20	Returns the first byte of a number
Bitshift left	<<	2<<2 = 8	Shifts the bits by a given number of spaces to the left.
Bitshift right	>>	2>>1=1	Shifts the bits by a given number of spaces to the right.
Bitwise and	&	\$3f & \$0f = \$f	Performs bitwise and between two

			numbers
Bitwise or		\$0f \$30 = \$3f	Performs a bitwise or between two numbers
Bitwise eor	^	\$ff ^ \$f0 = \$0f	Performs a bitwise exclusive or between two numbers

You can get the number representation of an arbitrary value by using the general `.number()` function. Eg.

```
.print 'x'.number()
```

4.5 Parentheses

Since traditional 6510 assembler notation has already used soft parenthesis to signal an indirect addressing mode, you will have to use hard parenthesis to specify a sub expression that must be evaluated before others.

```
lda #2+5*2           // gives lda #12
lda #[2+5]*2        // gives lda #14
```

You can nest as many parentheses as you want, so `[[[2+4]]*3]+25.5` is a legal expression.

4.6 String Values

Strings are used to contain text. You can define a string like this:

```
.var message = "Hello World"
.text message // Gives .text "Hello world"
```

Every object has a string representation and you can concatenate strings with the `+` operator. For example:

```
.var x=25
.var myString= "X is " + x // Gives myString = "X is 25"
```

You can use the `.print` directive to print a string to the console while assembling. This is useful when debugging. Printing `x` and `y` can be done like this:

```
.print "x="+x
.print "y="+y
```

You can also print labels to see which location they refer to. If you do this, it's best to convert the label value to hexadecimal notation first:

```
.print "int1=$"+toHexString(int1)

int1: sta regA+1
      stx regX+1
      sty regY+1
      lsr $d019
      // Etc.
```

Here is a list of functions/operators defined on strings:

Function/Operator	Description
+	Appends two strings
size()	Returns the number of characters in the string
charAt(n)	Returns the character at position n
substring(i1,i2)	Returns the substring beginning at i1 and ending at i2 (char at i2 not included)
asNumber()	Converts the string to a number value (eg, "35".asNumber())
asBoolean()	Converts the string to a boolean value (eg, "true".asBoolean())

Here are the functions that take a number value and convert it to a string:

Function	Description
toIntString(x)	Return x as a integer string (eg x=16.0 will return "16")
toBinaryString(x)	Return x as a binary string (eg x=16.0 will return "10000")
toOctalString(x)	Return x as a octal string (eg x=16.0 will return "20")
toHexString(x)	Return x as a hexadecimal string (eg x=16.0 will return "10")

You can get the number representation of an arbitrary value by using the general .string() function.
Eg.

```
.print 1234.string().charAt(2) // Prints 3
```

4.7 Char Values

Char values, or characters, are used like this:

```
lda #'H'  
sta $0400  
lda #'i'  
sta $0401  
  
lda #"?!#".charAt(1)  
sta $0402  
  
.byte 'H','e','l','l','o',' '  
.text "World"+'!'
```

In the above example, chars are used in two ways. In the first examples their numeric representation are used as arguments to the lda commands and in the final example, '!'s string representation is appended to the "World" string.

Char values is a subclass of number values, which means that it has all the functions that placed on the number values so you can do stuff like.

```
lda #'H'+1      // Same as lda #'I'  
sta $0400  
lda #'A'+1      // Same as lda #'B'  
sta $0401  
lda #'L'+1      // Same as lda #'M'  
sta $0402
```

4.8 The Math Library

Kick Assembler's math library is built upon the Java 5.0 math library. This means that nearly every constant and command in Java's math library is available in Kick Assembler. Here is a list of available constants and commands. For further explanation consult the Java 5.0 documentation at Suns homepage. The only non java math library function is mod (modulo).

Constant	Value
PI	3.141592653589793
E	2.718281828459045

Function	Description
abs(x)	Returns the absolute (positive) value of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent x
atan2(y,x)	Returns the angle of the coordinate (x,y) relative to the positive x-axis. Useful when converting to polar coordinates
cbrt(x)	Returns the cube root of x
ceil(x)	Rounds up to the nearest integer.
cos(r)	Returns the cosine of r
cosh(x)	Returns the hyperbolic cosine of r
exp(x)	Returns e^x
expm1(x)	Returns $e^x - 1$
floor(x)	Rounds down to the nearest integer
hypot(a,b)	Returns $\sqrt{x^2+y^2}$
IEEEremainder(x,y)	Returns the remainder of the two numbers as described in the IEEE 754 standard.
log(x)	Returns the natural logarithm of x
log10(x)	Returns the base 10 logarithm of x
log1p(x)	Returns $\log(x+1)$
max(x,y)	Returns the highest number of x and y
min(x,y)	Returns the smallest number of x and y
mod(a,b)	Converts a and b to integers and returns the remainder of a/b
pow(x,y)	Returns x^y
random()	Returns a random number x where $0 \leq x < 1$
round(x)	Rounds x to the nearest integer
signum(x)	Returns 1 if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$
sin(r)	Returns the sine of r
sinh(x)	Returns the hyperbolic sine of x
sqrt(x)	Returns the square root of x
tan(r)	Returns the tangent of r
tanh(x)	Returns the hyperbolic tangent of x
toDegrees(r)	Converts a radian angle to degrees
toRadians(d)	Converts a degree angle to radians

Here are some examples of use.

```
// Load a with a random number
lda #random()*256

// Generate a sine curve
.fill 256,round(127.5+127.5*sin(toRadians(i*360/256)))
```

4.9 List Values

List values are used to hold a list of other values. To create a list you use the 'List()' function. It takes one argument that tells how many elements it contains. If it is left out, the created list will be empty. Use the get and set operations to retrieve and set elements.

```
.var myList = List(2)
.eval myList.set(0,25)
.eval myList.set(1, "Hello world")
    .byte myList.get(0)      // Will give .byte 25
    .text myList.get(1)     // Will give .text "Hello world"
```

You can determine the number of elements in a list with the size function and the add function adds additional elements.

```
.var greetingsList = List()
.eval greetingsList.add("Maniacs of Noise", "Oxyron", "etc." )
.byte listSize = greetingsList.size() // gives .byte 3
```

A compact way to fill a list with elements is:

```
.var greetingsList = List().add("Maniacs of Noise", "Oxyron", "etc.")
```

Here is a list of functions defined on list values:

Functions	Description
get(n)	Gets the n'th element (first element is no zero)
set(n,value)	Sets the n'th element (first element is no zero)
add(value1, value2, ...)	Add elements to the end of the list
size()	Returns the size of the list
remove(n)	Removes the n'th element
shuffle()	Puts the elements of the list in random order
reverse()	Puts the elements of the list in reverse order
sort()	sorts the elements of the list (only numeric values are supported)

4.10 Working with Mutable Values

The list value described in the previous chapter is special since it is mutable, which means it can change its contents. At one point in time a list can contain the values [1,6,7] and at another time

[1,4,8,9]. The values previously described in the manual (Numbers, Strings, Booleans) are immutable since instances like 1, false, or “Hello World” can’t change.

In Kick Assembler 3, you will have to lock mutable values if you want to use them in a pass different from the one in which they were defined. When a value is locked, it becomes immutable and calling a function that modifies its content will cause an error. There are two ways to lock a mutable value. You can call its lock function:

```
// Locking a list with the lock function
.var list1 = List().add(1,3,5).lock()
```

Or you can define it inside a .define directive:

```
// The define directive locks the defined variables outside its scope
.define list2, list3 {
    .var list2 = List().add(1,2)

    .var list3= List()
    .eval list3.add("a")
    .eval list3.add("b")
}
//.eval list3.add("c") // This will give an error
```

The .define directive defines the symbols that are listed after the .define keyword (list2 and list3). The directives inside {...} are executed in a new scope so any local defined variables can’t be seen from the outside. After executing the inner directives, the defined values are locked and inserted as constants in the outside scope.

The inner directives are executed in ‘function mode’, which is a bit faster and requires less memory than ordinary execution. So if you are using for loops to do some heavy calculations, you can optimize performance by placing your loop inside a define directive. As the name ‘function mode’ suggests, directives placed inside functions are also executed in ‘function mode’. In ‘function mode’ you can only use script directives (like .var, .const, .eval, .enum, etc) while byte output generating directives (like lda #10, byte \$22, .word \$33, .fill 10, 0) are not allowed .

4.11 Hashtable Values

Hashtables are tables that map keys to values. You can define a hashtable with the Hashtable function. To enter and retrieve values you use the put and get functions, and with the keys function you can retrieve a list of all keys in the table:

```
.define ht {
    // Define the table
    .var ht = Hashtable()

    // Enter some values (put(key,value))
    .eval ht.put("ram", 64)
    .eval ht.put("bits", 8)
    .eval ht.put(1, "Hello")
    .eval ht.put(2, "World")
    .eval ht.put("directions", List().add("Up","Down","Left","Right"))
}
```

```

// Retrieve the values
.print ht.get(1)           // Prints Hello
.print ht.get(2)           // Prints World
.print "ram = " + ht.get("ram") + "kb" // Prints ram=64kb

// Print all the keys
.var keys = ht.keys()
.for (var i=0; i<keys.size(); i++) {
    .print keys.get(i)     // Prints "ram", "bits", 1, 2, directions
}

```

When a value is used as a key then it is the value's string representation that is used. This means that `ht.get("1.0")` and `ht.get(1)` returns the same element. If you try to get an element that isn't present in the table, null is returned.

Function	Description
<code>put(key,value)</code>	Maps 'key' to 'value'. If the key is previously mapped to a value, the previous mapping is lost.
<code>get(key)</code>	Returns the value mapped to 'key'. A null value is returned if no value has been mapped to the key.
<code>keys()</code>	Returns a list value of all the keys in the table
<code>containsKey(key)</code>	Returns true if the key is defined in the table, otherwise false.
<code>remove(key)</code>	Removes the key and its value from the table.

4.12 Vector and Matrix Values

Kick Assembler supports vector and matrix values. You can read about these in the section "Making 3D Calculations" in the special features chapter.

5 User Defined Structures

It's possible to define your own structures. A structure is a collection of variables like for example a point that consist of an x and a y coordinate:

```
// Define a point structure
.struct Point {x,y}

// Create a point with x=1 and y=2 and print it
.var p1 = Point(1,2)
.print "p1.x=" + p1.x
.print "p1.y=" + p1.y

// Create a point with the default constructor and modify its arguments
.var p2 = Point()
.eval p2.x =3
.eval p2.y =4
```

You define a structure with the `.struct` directive. The above structure has the name 'Point' and consists of the variables `x` and `y`. To create an instance of the structure, you use its name as a function. You can either supply no arguments or give the init values of all the variables. You use the values generated by structures as any other variables, ex:

```
lda #0
ldy #p1.y
sta charset+[p1.x>>3]*height,y
```

6 Branching and Looping

Kick Assembler has control directives that let you put conditions on when a directive is executed and how many time it is executed. These are explained in this chapter.

6.1 Boolean Values

The conditions for control directives are given by Boolean values, which are values that can be true or false. They are generated and used as in programming languages like Java and C#. The following are examples of boolean variables:

```
.var myBoolean1 = true           // Set the variable to true
.var myBoolean2 = false         // Set the variable to false
.var fourHigherThanFive = 4>5  // Sets fourHigherThanFive = false
.var aEqualsB = a==b           // Sets true if a is the same as b
.var xNot10 = x!=10            // Sets true if x doesn't equal 10
```

Here is the standard set of operators for generating Booleans:

Name	Operator	Example	Description
Equal	==	a==b	Returns true if a equals b, otherwise false
Not Equal	!=	a!=b	Returns true if a doesn't equal b, otherwise false
Greater	>	a>b	Returns true if a is greater than b, otherwise false
Less	<	a<b	Returns true if a is less than b, otherwise false
Greater than	>=	a>=b	Returns true if a is greater than or equal to b, otherwise false
Less than	<=	a<=b	Returns true if a is less or equal to b, otherwise false.

All the operators are defined for numeric values, other values have defined a subset of the above. E.g. you can't say that one boolean is greater than another, but you can see if they have the same values:

```
.var b1 = true==true           // Sets b1 to true
.var b2 = true!=[10<20]       // Sets b2 to false
```

Boolean values have a set of operators assigned. These are the following:

Name	Operator	Example	Description
Not	!	!a	Returns true if a is false, otherwise false
And	&&	a&&b	Returns true if a and b are true, otherwise false
Or		A b	Returns true if a or b are true, otherwise false

```
.var allTrue = 10HigherThan100 && aEqualsB // Is true if the two boolean
// arguments are true.
```

6.2 .if

If-directives work like in standard programming languages. With an .if directive you have the proceeding directive executed only if a given boolean expression is evaluated to true. Here are some examples:

```

// Set x to 10 if x is higher than 10
.if (x>10) .eval x=10

// Only show rastertime if the 'showRasterTime' boolean is true
.var showRasterTime = false
.if (showRasterTime) inc $d020
jsr PlayMusic
.if (showRasterTime) dec $d020

```

You can group several statements together in a block with {...} and have them executed together if the boolean expression is true:

```

// If IrqNr is 3 then play the music
.if (irqNr==3) {
    inc $d020
    jsr music+3
    dec $d020
}

```

By adding an else statement you can have an expression executed if the boolean expression is false:

```

// Add the x'th entry of a table if x is positive or
// subtract it if x is negative
.if (x>=0) adc zpXtable+x else sbc zpXtable+abs(x)

// Init an offset table or display a warning if the table length is exceeded
.if (i<tableLength) {
    lda #0
    sta offset1+i
    sta offset2+i
} else {
    .error "Error!! i is too high!"
}

```

6.3 Questionmark if's

As known from languages like Java and C++ you can use the write compact if expression in the following form:

```
condition ? trueExpr : falseExpr
```

Some examples of use:

```

.var x= true ? "hello" : "goodbye"           // Sets x = "hello"
.var y= (20<10) ? 1 : 2                     // Sets y=2

.var max = a>b ? a:b

.var debug=true
inc debug ? $d020:$d013                     // Increases $d020 since debug=true

.var limitedX = max(x,minLimit==null?0:minLimit) // Takes care of null limit

```

6.4 *.for*

With the *.for* directive you can generate loops as in modern programming languages. The *.for* directive takes an init expression list, a boolean expression, and an iteration list separated by a semicolon. The last two arguments and the body are executed as long as the boolean expression evaluates to true.

```
// Prints the numbers from 0 to 9
.for(var i=0;i<10;i++) .print "Number " + i

// Make data for a sine wave
.for(var i=0;i<256;i++) .byte round(127.5+127.5*sin(toRadians(360*i/256)))
```

Since argument 1 and 3 are lists, you can leave them out, or you can write several expressions separated by comma:

```
// Print the numbers from 0 to 9
.var i=0
.for (;i<10;) {
    .print i
    .eval i++
}

// Sum the numbers from 0 to 9 and print the sum at each step
.for(var i=0, var sum=0;i<10;sum=sum+i,i++)
    .print "The sum at step " + i " is " + sum
```

The for loop is good for generating tables and unrolling loops. You can, for example, do a classic ‘blitter fill’ routine like this:

```
.var blitterBuffer=$3000
.var charset=$3800
.for (x=0;x<16;x++) {
    for(var y=0;y<128;y++) {
        if (var y=0) lda blitterBuffer+x*128+y
        else      eor blitterBuffer+x*128+y
        sta charset+x*128+y
    }
}
```

6.5 *Optimization Considerations when using Loops*

Here is a tip if you want to optimize your assembling. Kick assembler has two modes of executing directives. ‘Function Mode’ is used when the directive is placed inside a function or define directive, otherwise ‘Asm Mode’ is used. ‘Function Mode’ is executed fast but is restricted to script commands only (*.var*, *.const*, *.for*, etc.), while ‘Asm Mode’ remembers intermediate results so the assembler won’t have to make the same calculations in succeeding passes.

If you make heavy calculations and get slow performance or lack of memory, then place your for loops inside a define directive or inside a function. No time or memory will be wasted to record intermediate results, and the define directive or the directive that called the function, will remember the result in the succeeding passes.

Read more about the define directive in the chapter ‘Working with mutable values’.

7 Macros, Functions and Pseudo Commands

This chapter shows how to define and use macros, pseudo commands and functions.

7.1 Macros

Macros are collections of assembler directives. When called, they generate code as if the directives were placed at the macro call. The following code defines and executes the macro ‘SetColor’:

```
// Define macro
.macro SetColor(color) {
    lda #color
    sta $d020
}

// Execute macro
:SetColor(1)
```

A macro can have any number of arguments. Macro calls are encapsulated in a scope, hence any variable defined inside a macro can’t be seen from the outside. This means that a series of macro calls to the same macro doesn’t interfere:

```
// Execute macro
:ClearScreen($0400,$20)      // Since they are encapsulated in a scope
:ClearScreen($4400,$20)      // the two resulting loop labels don't
                             // interfere

// Define macro
.macro ClearScreen(screen,clearByte) {
    lda #clearByte
    ldx #0
Loop:                          // The loop label can't be seen from the outside
    sta screen,x
    sta screen+$100,x
    sta screen+$200,x
    sta screen+$300,x
    inx
    bne Loop
}
```

Notice that it is ok to use the macro before it is declared.

Macros are good for building libraries. In my standard library I have macros for moving and filling memory, setting up char matrixes, declaring basic upstart programs, etc[†]. They are also good for doing things like double buffering. Typically you have a routine you want to work on two buffers, but making it take the buffer as an argument would slow it down. Instead you define the routine in a macro, which takes a buffer as an argument and then call the macro with each buffer. This saves you the trouble of maintaining two identical routines only differing by the buffer they use.

[†] The library isn’t included in the assembler.

7.2 Functions

You can also define your own functions. Here is an example of a function:

```
.function area(width,height) {
    .return width*height
}
lda #10+area(4,8)
```

You can use your own functions like you would use any of the library functions described earlier. Functions consist of non-byte generating directives like `.eval`, `.for`, `.var`, and `.if`. When the assembler evaluates the `.return` directives it returns the value given by the preceding expression. If no expression is given, or if no `.return` directive is reached, a null value is returned. Here are some more examples of functions:

```
// Returns a string telling if a number is odd or even
.function oddEven(number) {
    .if ([number&1] == 0 ) .return "even"
    else .return "odd"
}

// Empty function - always returns null
.function emptyFunction() {
}
```

As macros, functions can have side effects as shown in the following function that returns no result (null), but modifies its list argument.

```
// Inserts null in all elements of a list
.function clearList(list) {
    // Return if the list is null
    .if (list==null) .return

    .for(var i=0; i<list.size(); i++) {
        list.set(i,null)
    }
}
```

With functions you can calculate data for your programs. Instead of using other programming languages like C or Java you can code your data generators directly in Kick Assembler. Put your data in a list and use the list to generate your speed code or your `.byte` tables. This eases the development process by making it more integrated.

7.3 Pseudo Commands

Pseudo commands are a special kind of macros that takes command arguments, like #20, table,y or (\$30),y as arguments just like mnemonics do. With these you can make your own extended commands. Here is an example of a mov command that moves a byte from one place to another:

```
.pseudocommand mov src;tar {
    lda src
    sta tar
}
```

You use the mov command like this:

```
:mov #10 ; $1000           // Sets $1000 to 10  (lda #10, sta $1000)
:mov source ; target       // target = source  (lda source, sta target)
:mov source,x ; target,y   // (lda source,x , sta target,y)
:mov #20 ; ($30),y        // (lda #20, sta ($30),y )
```

The arguments to a pseudo command are separated by semicolon and you can use any argument you would give to a mnemonic.

The command arguments are passed to the pseudo command as CmdValues. These are values that contain an argument type and a number value. You access these by their getter functions. Here is a table of the functions:

Function	Description	Example
getType()	Returns a type constant (See the table below for possibilities)	#20 will return AT_IMMEDIATE
getValue()	Returns the value	#20 will return 20

The argument type constants are the following:

Constant	Example
AT_ABSOLUTE	\$1000
AT_ABSOLUTEX	\$1000 , x
AT_ABSOLUTHEY	\$1000 , y
AT_IMMEDIATE	#10
AT_INDIRECT	(\$1000)
AT_IZEROPAGEX	(\$10 , x)
AT_IZEROPAGEY	(\$10) , y
AT_NONE	

Some addressing modes, like absolute zeropage and relative, are missing from the above table. This is because the assembler automatically detect when these should be used from the corresponding absolute mode.

You can construct new command arguments with the CmdArgument function. If you want to construct a new immediate argument with the value 100, you do it like this:

```
.var myArgument = CmdArgument(AT_IMMEDIATE, 100)
lda myArgument // Gives lda #100
```

Now let's use the above functionalities to define a 16 bit instruction set. We start by defining a function that given the first argument will return the next in a 16 bit instruction.

```
.function _16bit_nextArgument(arg) {
    .if (arg.getType()==AT_IMMEDIATE)
        .return CmdArgument(arg.getType(),>arg.getValue())
    .return CmdArgument(arg.getType(),arg.getValue()+1)
}
```

We always return an argument of the same type as the original. If it's an immediate argument we set the value to be the high byte of the original value, otherwise we just increment it by 1. This will supply the correct argument for the ABSOLUTE, ABSOLUTEX, ABSOLUTEY and IMMEDIATE addressing modes. With this we can easily define some 16 bits commands:

```
.pseudocommand incl6 arg {
    inc arg
    bne over
    inc _16bit_nextArgument(arg)
over:
}

.pseudocommand mov16 src;tar {
    lda src
    sta tar
    lda _16bit_nextArgument(src)
    sta _16bit_nextArgument(tar)
}

.pseudocommand add16 arg1 ; arg2 ; tar {
    .if (tar.getType()==AT_NONE) .eval tar=arg1
    lda arg1
    adc arg2
    sta tar
    lda _16bit_nextArgument(arg1)
    adc _16bit_nextArgument(arg2)
    sta _16bit_nextArgument(tar)
}
```

You can use these like this:

```
:incl6 counter
:mov16 #irq1; $0314
:mov16 #startAddress; $30
:add16 $30; #128
:add16 $30; #1000; $32
```

Note how the target argument of the add16 command can be left out. When this is the case an argument with type AT_NONE is passed to the pseudo command and the first argument is then used as target.

With the pseudo command directive you can define your own extended instruction libraries, which speed up some of the more trivial tasks of programming.

8 Name Spaces

Namespaces are named scopes. When you enclose code in a scope, you hide information about the code for the outside. This is useful, since labels names won't collide, but sometimes you want to access these anyway. By using namespaces you can access this information. In this chapter the different uses of namespaces are explained.

8.1 The Namespace Directive

Suppose you want to divide your code into different parts with local labels and variables and want to access to some of the labels from the outside. This can be done with the `.namespace` directive:

```
jsr part1.init
jsr part1.exec
jsr part2.init
jsr part2.exec
rts

.namespace part1 {
init:
    ...
    rts

exec:
    ...
    rts
}

.namespace part2 {
init:
    ...
    rts

exec:
    ...
    rts
}
```

Inside a namespace you reference the labels as usual, but from the outside you append the namespace name as prefix to the label as seen in the `jsr` commands. Namespaces can be nested which is seen in the following example:

```
jsr part1.section1.exec
rts

.namespace part1 {
    .namespace section1 {
        exec:
            ...
            rts
    }
}
```

```

// You don't have to append part1 here since we are already in
// the part1 namespace
jsr section2.exec

.namespace section2 {
exec:
    ...
    rts
}
}

```

User defined labels can also be accessed, so if you want a constant to be exposed to the outside of your namespace then define it as a label:

```

.namespace vic {
    .label borderColor = $d020
    .label backgroundColor0 = $d021
    .label backgroundColor1 = $d022
    .label backgroundColor2 = $d023
}

lda #0
sta vic.backgroundColor0
sta vic.borderColor

```

8.2 File Namespaces

If you want the entire sourcefile to be placed in a namespace, you can put a `.filenamespace` directive in the top of the file.

```

.filenamespace mySpace
    .pc=$1000
start: inc $d020
    jmp start

```

It's equivalent to using the `.namespace` directive but it will save a pair of brackets.

8.3 Label Namespaces (Autonamespacing)

If you declare a scope after a label, the scope will automatically declare a namespace. This is handy if you use scoping to make the labels of your functions local. In the example below the clearScreen label and the succeeding scope creates a namespace with the name clearScreen.

```
    lda #' '  
    sta clearScreen.fillbyte+1  
    jsr clearScreen  
    rts  
  
clearScreen: {  
fillbyte: lda #0  
    ldx #0  
loop:  
    sta $0400,x  
    sta $0500,x  
    sta $0600,x  
    sta $0700,x  
    inx  
    bne loop  
  
    rts  
}
```

The above code fills the screen with black spaces. The code that calls the clearScreen subroutine uses the namespace to access the fillbyte label. If you use the label directive to define the fillbyte label, the code can be done a little nicer:

```
    lda #'a'  
    sta clearScreen2.fillbyte  
    jsr clearScreen2  
    rts  
  
ClearScreen2: {  
    .label fillbyte = *+1  
    lda #0  
    ldx #0  
loop:  
    sta $0400,x  
    sta $0500,x  
    sta $0600,x  
    sta $0700,x  
    inx  
    bne loop  
    rts  
}
```

Now you don't have to remember to add one to the address before storing the fill byte. Autonamespacing works with both normal labels and the label directive, so its also possible to write programs like this:

```
.label mylabel1= $1000 {  
    .label mylabel2 = $1234  
}  
.print "mylable2="+mylabel1.mylable2
```

8.4 Accessing Local Labels of Macros and Pseudocommands

Autonamespacing makes it possible to access local labels of executed Macros and pseudocommands as demonstrated in the following program:

```
.pc=$1000

start:
    inc c1.color
    dec c2.color
c1:   :setColor()
c2:   :setColor()
    jmp start

.macro setColor() {
    .label color = *+1
    lda #0
    sta $d020
}
```


9 Special Features

In the previous chapters we have described general features of Kick Assembler that can be used to solve a wide range of problems. In this chapter, we describe special features that were implemented to solve specific problems, such as converting graphics, importing sid files, or making vector calculations.

9.1 Creating a Basic Upstart Program

To make the assembled machine code run on a C64 or in an emulator, it's useful to include a little basic program that starts your code (for example: `10 sys 4096`). The `BasicUpstart` macro is a standard macro that helps you to create programs like that. The following program shows how it's used:

```
.pc = $0801 "Basic Upstart"
:BasicUpstart(start)      // 10 sys$0810

.pc = $0810 "Program"
start:
    inc $d020
    inc $d021
    jmp start
```

TIP: Insert a basic upstart program in the start of your programs and use the `-execute` option to start Vice. This will automatically load and execute your program in Vice after successful assembling.

There is a second variation of the basic upstart macro that also takes care of setting up memory blocks:

```
:BasicUpstart2(start)    // 10 sys$0810
start:
    inc $d020
    inc $d021
    jmp start
```

If you want to see the script code for the macros, you can look in the `autoinclude.asm` file in the `KickAss.jar` file.

9.2 Opcode Constants

When making self-modifying code or code that unrolls speed code, you have to know the value of the opcodes involved. To make this easier, all the opcodes have been given their own constant. The constant is found by writing the mnemonic in uppercase and appending the addressing mode. For example, the constant for a `rts` command is `RTS` and `'lda #0'` is `LDA_IMM`. So, to place an `rts` command at target you write:

```
lda #RTS
sta target
```

You get the size of a mnemonic by using the `asmCommandSize` command

```
.var rtsSize = asmCommandSize(RTS)           //rtsSize=1
.var ldaSize1 = asmCommandSize(LDA_IMM)      //ldaSize1=2
.var ldaSize2 = asmCommandSize(LDA_ABS)      //ldaSize2=3
```

Here are a list of the addressing modes and example constants:

Argument	Description	Example constant	Example command
	None	RTS	rts
IMM	Immediate	LDA_IMM	lda #\$30
ZP	Zeropage	LDA_ZP	lda \$30
ZPX	Zeropage,x	LDA_ZPX	lda \$30,x
ZPY	Zeropage,y	LDX_ZPY	ldx \$30,y
IZPX	Indirect zeropage,x	LDA_IZPX	lda (\$30,x)
IZPY	Indirect zeropage,y	LDA_IZPY	lda (\$30),y
ABS	Absolute	LDA_ABS	lda \$1000
ABSX	Absolute,x	LDA_ABSX	lda \$1000,x
ABSY	Absolute,y	LDA_ABSY	lda \$1000,y
IND	Indirect	JMP_IND	jmp (\$1000)
REL	Relative	BNE_REL	bne loop

9.3 Colour Constants

Kick Assembler has build in the C64 colour constants:

Constant	Value
BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7
ORANGE	8
BROWN	9
LIGHT_RED	10
DARK_GRAY/DARK_GREY	11
GRAY/GREY	12
LIGHT_GREEN	13
LIGHT_BLUE	14
LIGHT_GRAY/LIGHT_GREY	15

Example of use:

```
lda #BLACK
sta $d020
lda #WHITE
sta $d021
```

9.4 Exporting Labels

By using the `-symbolfile` option at the commandline it's possible to export all the assembled symbols. The line

```
java -jar KickAss.jar source1.asm -symbolfile
```

will generate the file `source1.sym` while assembling. Let's say the content of `source1` is:

```
.filenamespace source1
.pc = $2000
clearColor:
    lda #0
    sta $d020
    sta $d021
    rts
```

The content of `source1.sym` will be:

```
.namespace source1 {
    .label clearColor = $2000
}
```

It's now possible to refer to the labels of `source1.asm` from another file just by importing the `.sym` file:

```
.import source "source1.sym"
jsr source1.clearColor
```

Another way of exporting labels is the `-vicesymbols` option that exports the label to a `.vs` file that can be read by the VICE emulator.

9.5 Import of Binary Files

It's possible to load any file into a variable. This is done with the `LoadBinary` function. To extract bytes of the file from the variable you use the `get` function. You can also get the size of the file with the `getSize` function. Here is an example

```
// Load the file into the variable 'data'
.var data = LoadBinary("myDataFile")

// Dump the data to the memory
myData: .fill data.getSize(), data.get(i)
```

When you know the format of the file, you can supply a template string that describes the memory blocks. Each block is given a name and a start address relative to the start of the file. When you

supply a template to the LoadBinary function, the returned value will contain a get and a size function for each memory block:

```
.var dataTemplate = "XCoord=0,YCoord=$100, BounceData=$200"
.var file = LoadBinary("moveData", dataTemplate)
XCoord: .fill file.getXCoordSize(), file.getXCoord(i)
YCoord: .fill file.getYCoordSize(), file.getYCoord(i)
BounceData: .fill file.getBounceDataSize(), file.getBounceData(i)
```

There is a special template tag named 'C64FILE' that is used to load native c64 files. When this is in the template string, the LoadBinary function will ignore the two first byte of the file, since the first two bytes of a C64 file are used to tell the loader the start address of the file. Here is an example of how to load and display a Koala Paint picture file:

```
.const KOALA_TEMPLATE = "C64FILE, Bitmap=$0000, ScreenRam=$1f40, ColorRam=$2328,
                        BackgroundColor = $2710"
.var picture = LoadBinary("picture.prg", KOALA_TEMPLATE)

.pc = $0801 "Basic Program"
:BasicUpstart($0810)

.pc = $0810 "Program"
    lda #$38
    sta $d018
    lda #$d8
    sta $d016
    lda #$3b
    sta $d011
    lda #0
    sta $d020
    lda #picture.getBackgroundColor()
    sta $d021
    ldx #0

!loop:
    .for (var i=0; i<4; i++) {
        lda colorRam+i*$100,x
        sta $d800+i*$100,x
    }
    inx
    bne !loop-
    jmp *

.pc = $0c00          .fill picture.getScreenRamSize(), picture.getScreenRam(i)
.pc = $1c00 colorRam: .fill picture.getColorRamSize(), picture.getColorRam(i)
.pc = $2000          .fill picture.getBitmapSize(), picture.getBitmap(i)
```

Notice how easy it is to reallocate the screen and color ram by combining the .pc and .fill directives. To avoid typing in format types too often, Kick Assembler has some build in constants you can use:

Binary format constant	Blocks	Description
BF_C64FILE		A C64 file (The two first bytes are skipped)
BF_BITMAP_SINGLECOLOR	ColorRam, ScreenRam,	The Bitmap single color format outputted from Timanthes.

	Bitmap	
BF_KOALA	Bitmap, ScreenRam, ColorRam, BackgroundColor	Files from Koala Paint
BF_FLI	ColorRam, ScreenRam, Bitmap	Files from Blackmails FLI editor.

So if you want to load a FLI picture, just write

```
.var fliPicture = LoadBinary("GreatPicture", BF_FLI)
```

The formats were chosen so they cover the outputs of Timanthes (NB. Timanthes doesn't save the background color in koala format, so if you use that you will get an overflow error). If you feel that some formats are missing, then send their format strings to me and I will include them in future versions of Kick Assembler.

TIP: If you want to know how data is placed in the above formats, just print the constant to the console while assembling. Example: `.print "Koala format="+BF_KOALA`

9.6 Import of SID Files

The script language knows the format of SID files. This means that you can import files directly from the HVSC (High Voltage Sid Collection) which uses this format. To do this you use the LoadSid function which returns a value that represents the sidfile.

```
.var music = LoadSid("C:/c64/HVSC_44-all-of-them/C64Music/Tel_Jeroen/Closing_In.sid")
```

From this you can extract data such as the init address, the play address, info about the music and the song data.

Attribute/Function	Description
Header	The sid file type (PSID or RSID)
Version	The header version
location	The location of the song
Init	The address of the init routine
Play	The address of the play routine
Songs	The number of songs
startSong	The default song
Name	A string containing the name of the module
Author	A string containing the name of the author
Copyright	A string containing copyright information
Speed	The speed flags (Consult the Sid format for details)
Flags	flags (Consult the Sid format for details)
Startpage	Startpage (Consult the Sid format for details)
Pagelength	Pagelength (Consult the Sid format for details)

Size	The data size in bytes
getData(n)	Returns the n'th byte of the module. Use this function together with the size variable to store the modules binary data into the memory.

Here is an example of use:

```
//-----
//-----
//                               SID Player
//-----
//-----
.var music = LoadSid("Nightshift.sid")
:BasicUpstart2(start)
start:
        lda #$00
        sta $d020
        sta $d021
        ldx #0
        ldy #0
        lda #music.startSong-1
        jsr music.init
        sei
        lda #<irq1
        sta $0314
        lda #>irq1
        sta $0315
        asl $d019
        lda #$7b
        sta $dc0d
        lda #$81
        sta $d01a
        lda #$1b
        sta $d011
        lda #$80
        sta $d012
        cli

this:   jmp this
//-----
irq1:
        asl $d019
        inc $d020
        jsr music.play
        dec $d020
        pla
        tay
        pla
        tax
        pla
        rti

//-----
.pc=music.location "Music"
.fill music.size, music.getData(i)

//-----
// Print the music info while assembling
.print ""
.print "SID Data"
.print "-----"
.print "location=$"+toHexString(music.location)
.print "init=$"+toHexString(music.init)
.print "play=$"+toHexString(music.play)
```

```

.print "songs="+music.songs
.print "startSong="+music.startSong
.print "size=$"+toHexString(music.size)
.print "name="+music.name
.print "author="+music.author
.print "copyright="+music.copyright

.print ""
.print "Additional tech data"
.print "-----"
.print "header="+music.header
.print "header version="+music.version
.print "flags="+toBinaryString(music.flags)
.print "speed="+toBinaryString(music.speed)
.print "startpage="+music.startpage
.print "pagelength="+music.pagelength

```

Assembling the above code will create a musicplayer for the given sidfile and print the information in the music file while assembling:

```

SID Data
-----
location=$1000
init=$1d70
play=$1003
songs=1.0
startSong=1.0
size=$d78
name=Nightshift
author=Ari Yliaho (Agemixer)
copyright=2001 Scallop

Additional tech data
-----
header=PSID
header version=2.0
flags=100100
speed=0
startpage=0.0

```

TIP: If you use the `-libdir` option to point to your HVSC main directory, you don't have to write long filenames. For example:

```
.var music = LoadSid("C:/c64/HVSC_44-all-of-them/C64Music/Tel_Jeroen/Closing_In.sid")
```

will be

```
.var music = LoadSid("Tel_Jeroen/Closing_In.sid")
```

9.7 Converting Graphics

Kick Assembler makes it easy to convert graphics from gif and jpg files to the basic C64 formats. A picture can be loaded into a picture value by the `LoadPicture` function. The picture value can then be accessed by various functions depending on which format you want. The following will place a single color logo in a standard 32x8 char matrix charset placed at \$2000.

```

.pc = $2000
.var logo = LoadPicture("CML_32x8.gif")
.fill $800, logo.getSinglecolorByte([i>>3]&$1f, [i&7] | [i>>8]<<3)

```

If you don't like the compact form of the .fill command you can use a for loop instead. The following will produce the same data:

```
.pc = $2000
.var logo = LoadPicture("CML_32x8.gif")
.for (var y=0; y<8; y++)
    .for (var x=0;x<32; x++)
        .for(var charPosY=0; charPosY<8; charPosY++)
            .byte logo.getSinglecolorByte(x, charPosY+y*8)
```

The LoadPicture can take a color table as the second argument. This is used to decide which bit pattern is produced by a pixel. In single color mode there are two bit patterns (%0 and %1) and multi color mode has four (%00, %01, %10 and %11). If you don't specify a color table, a default table is created based on the colors in the picture. However, normally you wish to control which color is mapped to a bit pattern. The following shows how to convert a picture to a 16x16 multi color char matrix charset:

```
.pc = $2800 "Logo"
.var picture = LoadPicture("Picture_16x16.gif",
    List().add($444444, $6c6c6c,$959595,$000000))
.fill $800, picture.getMulticolorByte(i>>7,i&$7f)
```

The four colors added to the list are the RGB values for the colors that are mapped to each bit pattern.

Finally the picture value contains a getPixel function from which you can get the RGB color of a pixel. This comes in handy when you want to make your own format for some special purpose.

Attributes and functions available on picture values:

Attribute/Function	Description
Width	Returns the width of the picture in pixels
Height	Returns the height of the picture in pixels
getPixel(x,y)	Returns the RGB value of the pixel at position x,y. Both x and y are given in pixels.
getSinglecolorByte(x,y)	Convertes 8 pixels to a single color byte using the color table. X is given as a byte number (= pixel position/8) and y is given in pixels.
getMulticolorByte(x,y)	Convertes 4 pixels to a multi color byte using the color table. X is given as a byte number (= pixel position/8) and y is given in pixels. (NB. This function ignores every second pixel since the C64 multi color format is half the resolution of the single color.)

9.8 Writing to User Defined Files

With the createFile function you can create/overwrite a file on the disk. You call it with a file name and it returns a value that can be used to write data to the file:


```
.var myFile = createFile("breakpoints.txt")
.eval myFile.writeln("Hello World")
```

IMPORTANT! For security reasons, you will have to use the `-afo` switch on the command line otherwise file generation will be blocked. Eg `“java -jar KickAss.jar source.asm -afo”` will do the trick.

File creation is useful for generating extra data for emulators. The following example shows how to generate a file with breakpoint for VICE:

```
.var brkFile = createFile("breakpoints.txt")

.macro break() {
    .eval brkFile.writeln("break " + toHexString(*))
}

.print "test"

.pc=$0801 "Basic"
:BasicUpstart (start)

.pc=$1000 "Code"
start:

    inc $d020
    :break()
    jmp start
```

When running VICE with the breakpoint file (use the `-moncommands` switch), VICE will run until the break and then exit to the monitor.

Here is a list of the functions on a file value:

Attribute/Function	Description
<code>writeln(text)</code>	Writes the 'text' to the file and insert a line shift.
<code>writeln()</code>	Insert a line shift

9.9 Passing Command Line Arguments to the Script

From the command line you can assign string values to variables which can be read from the script. This is done with the `‘:’` notation like this;

```
java -jar KickAss.jar mySource.asm :x=27 :sound=true :title="Beta 2"
```

The three variables `x`, `sound` and `beta2` and their string values will now be placed in a hashtable that can be accessed by the global variable `cmdLineVars`:

```

.print "version =" + cmdLineVars.get("version")
.var x= cmdLineVars.get("x").asNumber()
.var y= 2*x
.var sound = cmdLineVars.get("sound").asBoolean()
.if (sound) jsr $1000

```

9.10 Making 3D Calculations

To make it easy to calculate vector data, such as coordinates for a vector object or a pre calculated vector animation, Kick Assembler supports vector and matrix values.

Vector values are used to hold 3D vectors. They are created by the Vector function that takes x, y and z as argument:

```

.var v1 = Vector(1,2,3)
.var v2 = Vector(0,0,2)

```

You can access the coordinates of the vector by its get functions and do the most common vector operations by the assigned functions. Here are some examples:

```

.var v1PlusV2 = v1+v2
.print "V1 scaled by 10 is " + [v1*10]
.var dotProduct = v1*v2

```

Here is a list of vector functions and operators:

Function/Operator	Example	Decription
get(n)		Returns the n'th coordinate (x=0, y=1, z=2)
getX()		Returns the x coordinate
getY()		Returns the y coordinate
getZ()		Returns the z coordinate
+	Vector(1,2,3)+Vector(2,3,4)	Returns the sum of two vectors
-	Vector(1,2,3)-Vector(2,3,4)	Returns the result of a subtraction between the two vectors
* Number	Vector(1,2,3)* 4.2	Return the vector scaled by a number
* Vector	Vector(1,2,3)*Vector(2,3,4)	Returns the dot product
/	Vector(1,2,3)/2	Divides each coordinate by a factor and returns the result
X(v)	Vector(0,1,0).X(Vector(1,0,0))	Returns the cross product between two vectors

The matrix value represents a 4x4 matrix. You create it by using the Matrix function, or one of the other constructor functions described later. You access the entries of the matrix by using its get and set functions:

```

.var matrix = Matrix() // Creates an identity matrix
.eval matrix.set(2,3,100)
.print "Matrix.get(2,3)=" + matrix.get(2,3)

```

```
.print "The entire matrix=" + matrix
```

In 3d graphics matrixes are usually used to describe a transformation of a vector space. That can be to move the coordinates, to scale them, to rotate them, etc. The Matrix() operator creates an identity matrix, which is one that leaves the coordinates unchanged. By using the set function you can construct any matrix you like. However, Kick Assembler has constructor functions that create the most common transformation matrixes:

Function	Description
Matrix()	Creates an identity matrix.
RotationMatrix(aX, aY, aZ)	Creates a rotation matrix where aX, aY and aZ are the angles rotated around the x, y and z axis. The angles are given in radians.
ScaleMatrix(sX, sY, sZ)	Creates a scale matrix where the x coordinate is scaled by sX, the y-coordinate by sY and the z-coordinate by sZ.
MoveMatrix(mX, mY, mZ)	Creates a move matrix that moves mX along the x-axis, mY along the y-axis and mZ along the z-axis.
PerspectiveMatrix(zProj)	Creates a perspective projection where the eye-point is placed in (0,0,0) and coordinates are projected on the XY-plane where z=zProj

You can multiply the matrixes and thereby combine their transformations. The transformation is read from right to left, so if you want to move the space 10 along the x axis and then rotate it 45 degrees around the z-axis, you write:

```
.var m = RotationMatrix(0,0,toRadians(45))*MoveMatrix(10,0,0)
```

To transform a coordinate you multiply the matrix to transformed vector:

```
.var v = m*Vector(10,0,0)
.print "Transformed v=" + v
```

The functions defined on matrixes are the following:

Function/Operator	Example	Description
get(n,m)		Gets the value at n,m
set(n,m,value)		Sets the value at n,m
*Vector	Matrix()*Vector(1,2,3)	Return the product of the matrix and a vector
*Matrix	Matrix()*Matrix()	Returns the product of two matrixes

Here is a little program to illustrate how matrixes can be used. It pre calculates an animation of a cube that rotates around the x, y and z-axis and is projected on the plane where z=2.5. The data is placed at the label 'cubeCoords':

```

//-----
// Objects
//-----
.var Cube = List().add( Vector(1,1,1), Vector(1,1,-1), Vector(1,-1,1), Vector(1,-1,-1),
                      Vector(-1,1,1), Vector(-1,1,-1), Vector(-1,-1,1), Vector(-1,-1,-1))

//-----
// Macro for doing the precalculation
//-----
.macro PrecalcObject(object, animLength, nrOfXrot, nrOfYrot, nrOfZrot) {

    // Rotate the coordinate and place the coordinates of each frames in a list
    .var frames = List()
    .for(var frameNr=0; frameNr<animLength;frameNr++) {
        // Set up the transform matrix
        .var aX = toRadians(frameNr*360*nrOfXrot/animLength)
        .var aY = toRadians(frameNr*360*nrOfYrot/animLength)
        .var aZ = toRadians(frameNr*360*nrOfZrot/animLength)
        .var zp = 2.5 // z-coordinate for the projection plane
        .var m = ScaleMatrix(120,120,0)*
                PerspectiveMatrix(zp)*
                MoveMatrix(0,0,zp+5)*
                RotationMatrix(aX,aY,aZ)

        // Transform the coordinates
        .var coords = List()
        .for (var i=0; i<object.size(); i++) {
            .eval coords.add(m*object.get(i))
        }
        .eval frames.add(coords)
    }

    // Dump the list to the memory
    .for (var coordNr=0; coordNr<object.size(); coordNr++) {
        .for (var xy=0;xy<2; xy++) {
            .fill animLength, $80+round(frames.get(i).get(coordNr).get(xy))
        }
    }
}

//-----
// The vector data
//-----
.align $100
cubeCoords: :PrecalcObject(Cube,256,2,-1,1)
//-----

```

10 Testing

Kick Assembler has some build in `.assert` directives that are useful for testing. They were made to make it easy to test the assembler itself, but you can use them for testing your own pseudo-commands, macros, functions. When assertions are used, the assembler will automatically count the number of assertions and the number of failed assertions and display these when the assembling has finished.

10.1 Asserting expressions

With the `assert` directive you can test the value of expressions. It takes three arguments: a description, an expression, and an expected result.

```
.assert "2+5*10/2", 2+5*10/2, 27
.assert "2+2", 2+2, 5
.assert "Vector(1,2,3)+Vector(1,1,1)", Vector(1,2,3)+Vector(1,1,1),
Vector(2,3,4)
```

When assembling this code the assembler prints the description, the result of the expression and the expected result. If these don't match an error message is appended:

```
2+5*10/2=27.0 (27.0)
2+2=4.0 (5.0) -- ERROR IN ASSERTION!!!
Vector(1,2,3)+Vector(1,1,1)=(2.0,3.0,4.0) ((2.0,3.0,4.0))
```

10.2 Asserting errors in expressions

To make sure that an expression gives an error when the user gives the wrong parameters to a function, use the `.asserterror` directive:

```
.asserterror "Test1" , 20/10
.asserterror "Test2" , 20/false
```

In the above example `test1` will fail since its perfectly legal to divide 20 by 10. `Test2` will produce the expected error so this assertion is ok. The above will give the following output:

```
Test1 -- ERROR IN ASSERTION!
Test2 -- OK. | Can't get a numeric representation from a value of type
boolean
```

10.3 Asserting code

The `assert` directive has a second form which makes it possible to compare pieces of assembled code:

```

.assert "Test2", { lda $1000 }, {ldx $1000}

.assert "Test", {
    .for (var i=0; i<4; i++)
        sta $0400+i
}, {
    sta $0400
    sta $0401
    sta $0402
    sta $0403
}

```

The assert directive will give an ok or failed message and the assembled result as output. The output of the above example is as follows:

```

Test1 -- FAILED! | 2000:ad,00,10 -- 2000:ae,00,10
Test2 -- OK. | 2000:8d,00,04,8d,01,04,8d,02,04,8d,03,04

```

10.4 Asserting errors in code

Like the assert directive the asserterror directive also has a form that can assert code:

```

.asserterror "Test" , { lda #"This must fail"}

```

Output:

```

Test -- OK. | The value of a Command Argument Value must be an integer.
Can't get an integer from a value of type 'string'

```

11 Some technical details

In Kick Assembler 3 some rather advanced techniques have been implemented to make the assembling more flexible and correct. I'll describe some of the main points here. **YOU DON'T NEED TO KNOW THIS**, but if you are curious about technical details then read on.

11.1 The flexible parse algorithm

Kick Assembler 3 uses a flexible pass algorithm, which parses each assembler command or directive as much as possible in each pass. Some commands can be finished in first pass, such as `lda #10` or `sta $1000`. But if a command depends on information not yet given, like `'jmp routine'` where the routine label hasn't been defined yet, an extra pass is required. Kick Assembler keeps executing passes until the assembling is finished or no progress has been made. You can write programs that only need one pass, but most programs will need two or more. This approach is more flexible and gives advantages over normal fixed pass assembling. All directives don't have to be in the same phase of assembling, which gives some nice possibilities for future directives.

11.2 Recording of side effects

Side effects of directives are now recorded and replayed the subsequent passes. Consider the following `eval` directive: `.eval a=[5+8/2+1]*10`. In the first pass the calculation $[5+8/2 + 1]*10$ will be executed and find the result 100, which will be assigned to `a`. In the next pass no calculation is done, only the side effect (`a=100`) is executed. This speeds up programs with heavy scripting, since the script only has to execute once.

11.3 Function Mode and Asm Mode

Kick assembler has two modes for executing directives. 'Function Mode' is used when the directive is placed inside a function or `.define` directive, otherwise 'Asm Mode' is used. 'Function Mode' is executed fast but is restricted to script commands only (`.var`, `.const`, `.for`, etc.), while 'Asm Mode' can handle all directives and records the side effects as described in previous section. All evaluation starts in 'Asm Mode' and enters 'Function Mode' if you get inside the body a function or `.define` directive. This means that at some point there is always a directive that records the result of the evaluation.

11.4 Invalid value calculations

Invalid values occur when the information used to calculate a value that isn't available yet. Usually this starts with an unresolved label value, which is defined later in the source code. Normally you would stop assembling the current directive once you reach an invalid value, but that might leave out some side effects you did intend to happen, so instead of stopping, the assembler now carries on operating on the invalid values. So an unresolved label is just an unresolved Number value. If you add two number values and one of them is invalid then the result will be another invalid number value. If you compare two invalid numbers then you get an invalid boolean and so forth. This helps to track down which values to invalidate. If for example you use an invalid number as index in a set function on a list, you must invalidate the whole list since you don't know which element is overwritten.

```
4+InvalidNumber -> InvalidNumber
InvalidNumber != 5 -> InvalidBoolean
myList.set(3, InvalidNumber) -> [?,?,InvalidNumber]
myList.set(InvalidNumber, "Hello") -> InvalidList
myList.set(4+4*InvalidNumber, "Hello") -> InvalidList
```


12 Going from Kick Assembler 2.x to 3.x

If you have trouble assembling your old Kick Assembler 2.x sources then check the following list:

- The `.if` directive now has its own scope, so you can no longer access variables defined inside an `if` from the outside scope. Instead, just define your variables before the `.if` directive.
- To access a mutable value defined in a different pass it has to be locked. Consult the chapter ‘Working with Mutable Values’ to see how this works.
- If you experience bad performance or lack of memory, then read the chapter “Optimization Considerations when using Loops”

13 Command line options

The command line options for Kick Assembler are:

Option	Example	Description
-o	-o dots.prg	Sets the output file. Default is the input filename with a '.prg' as suffix.
-libdir	-libdir ../stdLib	Defines a library where the assembler will look when it tries to open external files.
-showmem	-showmem	Show a memory map after assembling.
-execute	-execute x64 or -execute "x64 +sound"	Execute a given program with the assembled file as argument. You can use this to start a C64 emulator with the assembled program if the assembling is successful..
-warningsoff	-warningsoff	Turns off warning messages
-log	-log logfile.txt	Prints the output of the assembler to a logfile
-dtv	-dtv	Enables DTV opcodes
-aom	-aom	Allow overlapping memory blocks. With this option, overlapping memory blocks will produce a warning instead of an error.
-time	-time	Displays the assemble time.
-vicesymbols	-vicesymbols	Generates a label file for VICE.
-binfile	-binfile	Sets the output to be a bin file instead of a prg file. The difference between a bin and a prg file is that the bin file doesn't contain the two start address bytes.
-afo	-afo	Allows file output to user defined files
:name=value	:x=34 :version=beta2 :path="c:/C 64/"	The ':' notation denotes string variables passed to the script. They can be accessed by using the 'cmdLineVars' hashtable which is available from the script.
-symbolfile	-symbolfile	Generates a .sym file with the resolved symbols. The file can be used in other sources with the .import source directive.
-fillbyte	-fillbyte 255	Sets the byte used to fill the space between memoryblocks in the prg file.