Kick Assembler V2.24
Reference Manual

By Mads Nielsen

Index

1 Introduction

This is the manual for Kick Assembler. Kick Assembler is the combination of an assembler for doing 6510 machine code and a high level script language. With the assembler functionalities you can write your assembler programs, and with the script language you can write programs that generate data to use in the assembler programs. This could be data such as sine waves, coordinates for a vector object, or graphic converters etc. In addition you can combine assembler commands and scripting commands which is a really powerful combination. A little example: Where other assemblers can do simple unrolling of loops, Kick Assembler can base the unrolling of a loop on a list generated by the script language and select the content of the loop body based on the content of the list. This makes it more flexible when generating speed code.

I would like to thank some people who made it easier to do this assembler. Thanks to Martin 'Cruzer' Kristensen for proofreading and testing the assembler, John 'Graham' Selck for his page about the opcodes at www.oxyron.de, Gerwin Klein for doing JFlex (the lexical analyser used for this assembler) and to Scott Hudson, Frank Flannery and C. Scott Ananian for doing CUP (The parser generator).

I would like to hear from people using this assembler so don't hesitate to write your comments to kickassembler@no.spam.theweb.dk (<- Remove no.spam. for real address). After the publication of the beta-release on CSDB, a lot of cool feedback has found its way to my mailbox. Thanks guys! Your feedback is greatly appreciated!

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

Kick Assembler is written in java and distributed in the executable jar file 'kickass.jar'. To run Kick Assembler, you have to have Java5.0 or better installed on you machine. This can be downloaded for free from Sun's website (http://java.com/en/download/index.jsp). To assemble the file myCode.asm simply write:

```
java –jar kickass.jar myCode.asm
```

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 as the .pc directive, comments and 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
         :SetBorderColor(2)
         jsr $1003 // play music
         :SetBorderColor(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
}
```

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 coding style. 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.

3.3 Number formats

Kick Assembler supports the standard number formats:

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

3.4 Labels and multi labels

Label declarations in Kick Assembler ends 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 your 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 last instance of !loop

       ldx #100
```

```
!loop: inc $d021
       dex
       bne !loop-    // Jumps to the last 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 last !loop label
```

Another way to avoid conflicting variables is to use user defined scopes which is explained in the Scopes section of the Expressions 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 compiler a memory map will be generated which

display the memory usages and the name of the block. 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 aren't 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 a star 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 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 since crossing a memory page boundary gives 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, .text and .word directives are used to generate byte data, word data (one word= two 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 then you can use the .pseudopc directive:

```
.pc = $1000 "Program to be relocated in $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
.pc = <expr> ["name"]	.pc = $1000 "Program"	Set the program counter.
.align <expr>	.align $100	Aligns the program counter to a given interval.
.byte <expr list>	.byte 1,2,3,4	Generates byte data.
.word <expr list>	.word 1,2,3,4	Generates word data.
.text <expr>	.text "Hello"	Generates text data.
.fill <expr>, <expr>	.fill 256, 0 .fill 256, i	Generates a number of bytes, given by the first expression, with the data given by the second expression. The variable i is set to the byte number in the second expression.
.pseudopc <expr> {…}	.pseudopc $2000 {…}	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 in 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 a 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
```

3.7 Comments

Comments are pieces of the program that are ignored by the assembler. Kick Assembler supports line comments and block comments known from language 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	// This is also a comment	Ignores the rest of the line
C block comments	/* This is a block comment */	Ignores everything between /* and */

3.8 User console output (.print and .error)

With the .print directive you can output text to the user while assembling. Eg:

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

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

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

4 Expressions

Kick assembler has a build in mechanism for evaluating expressions. An example of an expression is 25+2*3/x. Expressions can be used in many 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
```

Normal assemblers can only calculate expressions based on numbers, while Kick Assembler can evaluate expressions based on a 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(2)
```

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(3).getX()
```

I think you have got the idea by now. Kick Assembler 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 will be given a detailed description of how to use the value types and functions in Kick Assembler.

4.1 Variables and constants

Before you can use variables you have to declare them. You do this by a 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 exists: The ++ and the −− operator which automatically calls a referenced variables 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 returns a value, so that 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 too. Notice that the ++ and −− works as real ++ and −− postfix operators, which means that they returns 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” // Constant 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, so if you define a constant that points to a list, the content of the list can still change.

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=$ff} // Assigns values explicit
.enum {up,down,left,right, none=$ff} // you can mix implicit and explicit
                                     // assignment of values
```

4.2 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 which is 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
```

4.3 Numeric values

Numeric values are numbers covering both integers and floats. Standard numerical operators (+,-,*,/) 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
charmmem + 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 the 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 #<interupt1
sta $0314
lda #>interupt1
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

4.4 Parentheses

Since traditional 6510 assembler notation have already used soft parenthesis to signal an indirect addressing mode, you will have to use hard parenthesis to specify a sub expression that shall 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.

4.5 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 screen while assembling. This is useful in debugging. Printing x and y can be done like this:

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

You can also print labels to see which place in the memory they refer to. If you do this, its best to convert the labelvalue 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)

4.6 The math library

Kick Assembler's math library is built upon the Java5.0 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 java5.0 documentation at Suns homepage.

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
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
mod(a,b)	Converts a and b to integers and returns the remainder of a/b

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)))
```

4.7 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 the 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 more 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
set(n,value)	Sets the n’th element
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

4.8 Hashtable values

Hashtables are tables that maps 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 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 key then it’s the values string representation that’s used. This means that `ht.get(“1.0”)` and `ht.get(1)` gets the same element. If you try to get an element that isn’t present in table, a null value is returned.

Function	Description
put(key,value)	Maps ‘key’ to ‘value’. If the key is previously mapped to a value, the previous mapping is lost.
get(key)	Returns the value mapped to ‘key’. A null value is returned if no value has been mapped to the key.
keys()	Returns a list value of all the keys in the table

4.9 Vector and Matrix values

Kick Assembler also supports vector values 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 which 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 contructor 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 have the name 'Point' and consist of the variables x and y. To create an instance of the structure, you use its name as a function. You can either give no arguments to the function or give the init values of the variables. You can 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 have control directives which lets 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 and 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 equals 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 number 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 works like in programming languages. With an .if directive you have the following directive executed only if a given boolean expression is evaluated to true. Here are some examples:

```
// Set x to 10 if x is higher that 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 with a {…} 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 and
// the subtract it if x is negative
.if (x>=0) adc zpXtable+x else sbc zpXtable+abs(x)


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

6.3 .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 two last 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
    }
}
```

7 Macros, Functions and Pseudo commands

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

7.1 Macros

Macros are collections of assembler directives. When called they generate code as if the directives where 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)
```

The macro can have any number of arguments. Macro calls are encapsulated in a scope so 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 doesn'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 in the above example the macro execution happens before the definition. This is ok since in the first pass of the code the assembler searches for macros and uses them from the second pass.

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 when 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 otherwise maintaining two identical routines only differing by the buffer they use.

[*] The library isn't included in the assembler.

Macros don't return any values when executed, but they can have side effects so you can use them a bit like functions by giving them an argument in which they can return a value. But usually you would use functions instead.

7.2 Functions

You can also define you 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 following 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 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. Also see how the null value can be used as a null-pointer.

```
// 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 the 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 is 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 which 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_ABSOLUTEY	$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 list. 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 do some 16 bits commands:

```
.pseudocommand inc16 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:

```
:inc16 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 argument 1 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 Special features

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

8.1 Creating a basic upstart program

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

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

.pc =$0810 "Program"
!loop:
       inc $d020
       inc $d021
       jmp !loop-
```

TIP: Insert at 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.

8.2 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. So 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 an 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

8.3 Colour constants

Kick Assembler has build in the colour constants of the C64:

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	11
GRAY	12
LIGHT_GREEN	13
LIGHT_BLUE	14
LIGHT_GRAY	15

Example of use:

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

8.4 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. For 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 now also 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' which 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 byte of a C64 is used to tell the loader on which address it should place the file. Here is an example of how to load and display a picture file from Koala Paint:

```
.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 is skipped)
BF_BITMAP_SINGLECOLOR	ColorRam, ScreenRam, Bitmap	The Bitmap single color format outputted from Timanthes.
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 where chosen so they cover the outputs of Timanthes (NB. Timanthes don'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`

8.5 Import of PSID files

The script language knows the format of PSID 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
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
size	The size of
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:

```
.import source "stdlib.asm"
.pc =$0801 "Basic upstart program"
:BasicUpstart($5000)

//-------------------------------------------------------------
//-------------------------------------------------------------
//                    HVSC Player
//-------------------------------------------------------------
//-------------------------------------------------------------
.var music = LoadSid("C:/c64/HVSC_44-all-of-them/C64Music/Tel_Jeroen/Closing_In.sid")
.pc = $5000 "Main Program"
          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
```

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

```
;------------------------------------------------
;------------------------------------------------
;   Kick Assembler v2.00b - (C)2006 Mads Nielsen
;------------------------------------------------
;------------------------------------------------
…
SID Data
--------
location=$2000
init=$2007
play=$2000
songs=1.0
startSong=1.0
size=$e66
name=Closing In
author=Jeroen Tel
copyright=1990 Maniacs of Noise

Memory Map
----------
$0801-$080e Basic Upstart Program
$2000-$2e65 Music
$5000-$5048 Main Program

Writing file: HVSC_Player prg
```

TIP: If you use the –libdir option to point on your HVSC main directory then you don't have to write such 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")
```

8.6 Converting Graphics

Kick Assembler makes it easy to convert the 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 colortable as a second argument. This is used to decide which bit pattern is produced by a pixel. In single color mode there is two bit patters (%0 and %1) and multi color mode has four (%00, %01, %10 and %11). If you don't specify a colortable, 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 picturevalue 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 pixture 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.)

8.7 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 values 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 identintity 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 then, 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 transform 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 precalculates 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 frams 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)
//-------------------------------------------------------------------------------------------
```

9 Testing

The assembler has an .assert directive which makes it easy and quick to test a large number of expressions. This was mainly made to test the assembler itself. 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 and gives an error if they don’t match:

```
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 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 the 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 give a warning instead of an error.
-time	-time	Displays the assemble time