Kick Assembler

Reference Manual

By Mads Nielsen

Table of Contents

Chapter 1
Introduction

Welcome to Kick Assembler, an advanced MOS 65xx assembler combined with a Java Script like script language.

The assembler has all the features you would expect of a modern assembler like macros, illegal and DTV opcodes and commands for unrolling loops. It also has features like pseudo commands, import of SID files, import of standard graphic formats and support for 3rd party Java plugins. The script language makes it easy to generate data for your programs. 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++.The script language and the assembler is integrated. Unlike other solutions, where scripts are prepassed, the script code and the assembler directives works together giving a more complete solution.

As seen by the size of this manual, Kick Assembler has a lot of functionality. You don't need to know it all to use the assembler, and getting to know all the features may take some time. If you are new to Kick Assembler, a good way to start is to read Chapter 2, *Getting Started*, Chapter 3, *Basic Assembler Functionality* and Chapter 4, *Introducing the Script Language* and then supplement with the features you need. Also notice the quick reference appendix which contains lists of directives, options and values.

This is the fifth version of Kick Assembler. The first version (1.x) was a normal 6510 cross assembler developed around 2003 and was never made public. The second version (2.x) was developed in 2006 and combined the assembler with a script language, giving you the opportunity to write programs that generate data for the assembler code. Finally in august 2006 the project went public. The third version (3.x) improved the underlying assembling mechanism using a flexible pass algorithm, recording of side effects and handling of invalid values. This gave better performance, and made it possible make more advanced feature. The fourth version (4.x) replaced the parsing mechanism, which where made using a parser generator, with a handwritten one which is faster, more flexible and included a preprocessor. This made it possible to do new language constructs and have better error handling. It also replaced the scoping system so it includes all entities, not just symbols. The fifth version (5.x) added segments which give the opportunity to manage the output of directives and channel it to files, disk images and other segments.

Through the years the project have grown quite big, with a professional setup including a its own code repository, a large automated test suite and automatic building and deploying.

A lot of people have contributed with valuable comments and suggestions by mail and on CSDB. Thanks guys. Your feedback is greatly appreciated. Also thanks to 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). And finally, Thanks to XMLMind for sponsoring the project with a pro version of their XML editor in which this manual is written.

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..

Chapter 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 run on any platform with Java8.0 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 comments, how to use macros, how to include external files and how to use the BasicUpstart2-macro which inserts a basic sys-line to start your program.

This should be enough to get you (kick) started.

```
BasicUpstart2(start)
//--------------------------------------------------------
//--------------------------------------------------------
//   Simple IRQ
//--------------------------------------------------------
//--------------------------------------------------------
        * = $4000 “Main Program”
start:  lda #$00
        sta $d020
        sta $d021
        lda #$00
        jsr $1000    // init music
        sei
        lda #<irq1
        sta $0314
        lda #>irq1
        sta $0315
        lda #$7f
        sta $dc0d
        sta $dd0d
        lda #$81
        sta $d01a
        lda #$1b
        sta $d011
        lda #$80
        sta $d012
        lda $dc0d
        lda $dd0d
```

```
        asl $d019
        cli
        jmp *

//---------------------------------------------------------
irq1:   asl $d019
        SetBorderColor(2)
        jsr $1003    // play music
        SetBorderColor(0)
        jmp $ea81

//---------------------------------------------------------
        *=$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 Appendix A, *Quick Reference*). 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”
# This is a comment
```

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

All lines starting with # are treated as comments.

Chapter 3
Basic Assembler Functionality

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

3.1. Mnemonics

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

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

If you want to write several commands on one line then separate them with ; like this:

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

Kick Assembler supports all opcodes, also the illegal ones. A complete list of commands and their opcodes in the each mode is shown here:

Table 3.1. Mnemonics

cmd	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					$2c				
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				

cmd	noarg	imm	zp	zpx	zpy	izx	izy	abs	abx	aby	ind	rel
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	$80	$04	$14				$0c	$1c			
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		

cmd	noarg	imm	zp	zpx	zpy	izx	izy	abs	abx	aby	ind	rel
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:

Table 3.2. DTV Mnemonics

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

3.2. Argument Types

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

Table 3.3. 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[1]

You can force the assembler to use the absolute form of the mnemonic by appending .a or .abs. The same way you can tell the assembler to use zeropage mode when it would otherwise use an absolute mode.

```
lda.abs $0040,x   // Uses absolute mode
lda.a $0030,x     // Same as abs (abbreviation)
```

[1] If the argument is unknown (eg. an unresolved label) in the first pass, the assembler will assume it's a 16 bit value

```
stx.zp zpLabel,y  // Uses zeropage mode
stx.z zpLabel,y   // Same as zp (abbreviation)
.label zpLabel = $10

jmp.z $1000 // Modifies nothing, jmp don't have any zp mode
```

With the following extensions you can force specific modes. The are deprecated and only kept for backward compatibility:

Table 3.4. Deprecated Mnemonic 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	
ax, absx	Absolute,x	lda.absx $1234
ay, absy	Absolute,y	
I, ind	Indirect	jmp.i $1000
r, rel	Relative to program counter	

3.3. Number formats

Kick Assembler supports the standard number formats:

Table 3.5. Number formats

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

3.4. Labels, Arguments 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
```

You can put labels in front of mnemonic arguments. This can be useful when creating self modifying code:

```
        stx tmpX
        ...
        ldx tmpX:#$00
```

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 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 scoping section of Chapter 4, *Introducing the Script Language*.

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

```
// Jumps with '*'
        jmp *

        inc $d020
        inc $d021
        jmp *-6

// The same jumps with labels
this:   jmp this

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

When referencing a label that is not yet resolved, the assembler will assume a two byte address, even though it later is found to be in the zeropage. You can mark labels as being in the zeropage with the .zp directive:

```
        // Uses zeropage form of lda and sta eventhough the labels is first
        // resolved later
        lda zpReg1
        sta zpReg2

*=$10 virtual
.zp {
zpReg1: .byte 0
zpReg2: .byte 0
}
```

Note: Currently the .zp directive doesn't handle macros and pseudocommands called within the {}. Labels inside these will be in the form defined in the macro.

3.5. Memory Directives

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

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

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

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

Note: The old notation ('.pc=$1000') from Kick Assembler 2.x and 3.x is still supported.

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.

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

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

        *=$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:

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

        *=$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
```

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:

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

3.6. Data Directives

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 65xx assemblers.

```
.byte 1,2,3,4      // Generates the bytes 1,2,3,4
.word $2000,$1234  // Generates the bytes $00,$20,$34,$12
.dword $12341234   // Generates the bytes $34,$12,$34,$12
.text "Hello World"
```

You can use .by, .wo and .dw as aliases for .byte, .word and .dword, so '.by $10' is the same as '.byte $10'.

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
```

3.7. Encoding

The .text directive outputs bytes to the memory that represents the given textstring. The default encoding is 'screencode_mixed', which maps to the screencode representations of the charset with both uppercase and lowercase letters. To change the encoding, use the .encoding directive:

```
// How to use encoding
.encoding "screencode_upper"
.text "THIS IS WRITTEN IN THE UPPERCASE SINCE LOWERCASE CHARS ARE USE FOR GFX
 SIGNS"

.encoding "screencode_mixed"
.text "In this ENCODING we have both UPPER and lower case chars."
.text "Remember to swith to a charset that fits the encoding."
```

The encoding affects every operation that converts characters in the sourcecode to byte values, for instance the '.import text' directive is also affected.

The supported encodings are:

Table 3.6. Encodings

Name	Description
petscii_mixed	The petscii representation of the charset with both upper and lower case characters.
petscii_upper	The petscii representation of the charset with upper case and graphics characters.
screencode_mixed	The screencode representation of petscii_mixed
screencode_upper	The screencode representation of petscii_upper

3.8. Importing source code

Use the preprocessor to import other source files.

```
// Import the file "mylibrary.asm"
#import "MyLibrary.asm"

// Only import "UpstartCode.asm" if STAND_ALONE is defined
#importif STAND_ALONE "UpstartCode.asm"
```

Note that preprocessor commands starts with #. Refer to the chapter on the preprocessor for a detailed description.

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.asm:
#importonce
.print "This will only be printed once!"

File2.asm:
#import "File1.asm" // This will import File1
#import "File1.asm" // This will not import anything
```

NOTE! The v3.x directives for importing source files using the import directive (.import source "myfile.asm" and .importonce), not the preprocessor, is still supported. But its recommended to use the preprocessor directives, since they will give a more natural order of evaluation. Using the preprocessor will import the source at once while using the old import directive will first parse the entire file, and then import external files during evaluation.

3.9. Importing data

With the .import directive you can import data from external files into your code. You can import binary, C64, and text files:

```
// 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"
```

The binary, c64 and text import takes an offset and a length as optional parameters:

```
// import the bytes from the file 'music.bin', but skip the first 100 bytes
.import binary "Music.bin", 100

// Imports $200 bytes starting from position $402 (the two extra bytes is because
 its a c64 file)
.import c64 "charset.c64", $400, $200
```

As when importing sources files, the import directive also searches the folders given by the –libdir option when looking for a file.

3.10. 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 65xx assembler line comments (;) are not supported since the semicolon is used in for-loops in the script language.

3.11. Console Output

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)

      *=$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"
```

Another way of writing this is to use the .errorif directive that takes a boolean expression and a message text. An error is raised if the boolean expression is evaluated to true:

```
.var width = 45
.errorif with>40, "width can’t be higher than 40"
```

This is more flexible since it standard .if's has to be decided in the first pass which will give an (unwanted) error if you for example compare not yet resolved labels. If you for instance want to check for a page boundary crossing you can do like this:

```
    beq label1
    .errorif (>*) != (>label1), "Page crossed!"
    nop
    nop
label1:
```

3.12. Breakpoints and watches

Breakpoints and watches changes nothing in the code. They add debug information to emulators/debuggers. Currently this means adding info to the vice symbol file or the DebugDump file (C64Debugger).

You can set breakpoints in your code with the .break directive:

```
    // Example 1
    ldy #10
loop:
    .break         // This will put a breakpoint on 'inc $d020'
    inc $d020
    dey
    .break "if y<5" // This will add a string as argument for the breakpoint
    bne loop


    // Example 2
    lda #10
    .break     // Will place a breakpoint at the first nop in the macro
    MyMacro()

.macro MyMacro() {
    nop
    nop
    nop
}
```

The .break directive puts a breakpoint on the current memory position. As seen in the second breakpoint, you can add an argument to a breakpoint. The syntax of this is dependant on the consumer. The above case (.break "if y<5") is written for VICE's conditional expressions. VICE will then break if the y register is below 5.

Watches are defined like this

```
.watch $d018           // Watches $d018
.watch xpos+1          // Watches the address xpos+1
.watch $d000,$d00f     // Watches the range $d000-$d00f
.watch xpos,xpos+10,"store"   // Watches a range with the additional parameter
 "store"
.watch count,,"hex8"   // you can leave the second argument empty
```

First argument is the address. If second argument is given its the range between the two. Third argument is an optional text string with additional information. Consult your emulater/debugger manual for possible content of third argument.

Chapter 4
Introducing the Script Language

In this chapter the basics of the script language is introduced. We will focus on how Kick Assembler evaluates expressions, the standard values and libraries. Later 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

With variables you can store data for later use. Before you can use a variable you have to declare it. 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).

Other shorthands exist. The operators ++, --, +=, -=, *= and /= will automatically call a referenced variable with +1,-1, +y, -y, *y and /y. For example:

```
        .var x = 0
        .eval x++    // Gives x=x+1
        .eval x--    // Gives x=x-1
        .eval x+=3   // Gives x=x+3
        .eval x-=7   // Gives x=x-7
        .eval x*=3   // Gives x=x*3
        .eval x/=2   // Gives x=x/2
```

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
.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 your 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 65xx assemblers are the high and low-byte operators (>,<) that are typically used like this:

```
        lda #<interrupt1   // Takes the lowbyte of the interupt1 value
        sta $0314
        lda #>interrupt1   // Takes the high byte of the interupt1 value
        sta $0315
```

Table 4.1. Numeric Values

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.
Bitwise not	~	~%11 = %...11111100	Performs bitwise negation of the bits.

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

```
        .print ‘x’.number()
```

4.5. Parentheses

You can use both soft parentheses () and har parentheses [] to tell the order of evaluation.

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

Note that 65xx assemblers use soft parenthesis to signal an indirect addressing mode:

```
        jmp ($1000)    // Creates a jmp indirect command
        jmp [$1000]    // Creates a jmp absolute command
```

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 plain strings or escape code strings like this:

```
        // Plain strings
        .var message = "Hello World"
        .text message                          // Gives .text "Hello world"
        .const file="c:\newstuff"

        // String with escape codes ('\esc') start with @
        .print @"First line.\nSecond line."   // Using newline
        .print @“He said: \"Hello World\""    // Using " inside the string

        .text @"This text will loop now\$ff"  // placing hex values ($ff) in the
 text
```

@ in front of a string means you can use escape characters. Notice how '\n' in "c:\newstuff" is not a newline while '\n' in @"First line.\nSecond line." is. (Note: This is the opposite of C# and is this way to avoid breaking file references in old sources).

The supported escape codes are:

Table 4.2. Escape codes

Code	Example	Description
\b	@"\b"	Backspace
\f	@"\f"	Form feed
\n	@"Line1\nLine2"	Newline
\r	@"\r"	Carriage return
\t	.print @"Hello\tWorld"	Tab
\\	@"c:\\tmp\\myfile.txt”	Backslash
\"	@"It's called \"Bodiam Castle\""	Double quotes
\$	@"Hello world\$ff"	Two digit hex values

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:

Table 4.3. String Values

Function/Operator	Description
+	Appends two strings.
asBoolean()	Converts the string to a boolean value (eg, “true”.asBoolean()).
asNumber()	Converts the string to a number value. Ex, “35”.asNumber().
asNumber(radix)	Converts the string to a number value with the given radix (16=hexadecimal, 2=binary etc.). Ex, “f”.asNumber(16) will return 15.
charAt(n)	Returns the character at position n.
size()	Returns the number of characters in the string.
substring(i1,i2)	Returns the substring beginning at i1 and ending at i2 (char at i2 not included).
toLowerCase()	Return the lower version of the string.
toUpperCase()	Return the uppercase version of the string.

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

Table 4.4. Numbers to Strings

Function	Description
toIntString(x)	Return x as an integer string (eg x=16.0 will return “16”).
toIntString(x,minSize)	Return x as an integer string space-padded to reach the given minsize. (eg toIntString(16,5) will return “ 16”).
toBinaryString(x)	Return x as a binary string (eg x=16.0 will return “10000”).
toBinaryString(x,minSize)	Return x as a binary string zero-padded to reach the given minSize (eg toBinaryString(16,8) will return “00010000”).
toOctalString(x)	Return x as an octal string (eg x=16.0 will return “20”).
toOctalString(x,minSize)	Return x as an octal string zero-padded to reach the given minSize (eg toBinaryString(16,4) will return “0020”).
toHexString(x)	Return x as a hexadecimal string (eg x=16.0 will return “10”).
toHexString(x,minSize)	Return x as an hexadecimal string zero-padded to reach the given minSize (eg toBinaryString(16,4) will return “0010”).

You can get the string 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 are 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 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 documentation at Suns homepage. The only non Java math library function is mod (modulo).

Table 4.5. Math Constants

Constant	Value
PI	3.141592653589793
E	2.718281828459045

Table 4.6. Math Functions

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 ex.
expm1(x)	Returns ex-1.

Function	Description
floor(x)	Rounds down to the nearest integer.
hypot(a,b)	Returns sqrt(x2+y2).
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 raised to the power of 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)))
```

Chapter 5
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.

5.1. 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:

Table 5.1. Boolean generating Functions

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:

Table 5.2. Boolean Operators

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.

Name	Operator	Example	Description
Or	||	A||b	Returns true if a or b are true, otherwise false.

And are used like this:

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

Like in languages like C++ or Java, the && and || operators are short circuited. This means that if the first argument of an && operator is false, then the second argument won't be evaluated since the result can only be false. The same happens if the first argument of an || operator is true.

5.2. The .if directive

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 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 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!"
}
```

5.3. Question mark 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 boolean = max(x,minLimit==null?0:minLimit) // Takes care of null limit
```

5.4. The .for directive

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
```

With the for loop you can quickly generate tables and unroll 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
    }
}
```

5.5. The .while directive

The .while directive executes as long as a given expressions is true. That is, it works like a .for-loop but without the init and iteration parameters:

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

5.6. 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 section ‘Working with mutable values’.

Chapter 6
Data Structures

In the chapter, we will examine user defined data and predefined structures.

6.1. 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 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 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
```

You can get access to informations about the struct and access the fields in a more generic way by using the struct’s functions:

```
.struct Person{firstName,lastName}
.var p1 = Person(“Peter”,”Schmeichel”)

.print p1.getStructName()        // Prints ‘Person’
.print p1.getNoOfFields()        // Prints ‘2’
.print p1.getFieldNames().get(0) // Prints ‘firstName’

.eval p1.set(0,”Kasper”)         // Sets firstName to Kasper
.print p1.get(“lastName”)        // Prints “Schmeichel”


// Copy values from one struct to another
.var p2 = Person()
.for (var i=0; i<p1.getNoOfFields(); i++)
    .eval p2.set(i,p1.get(i))

// Print the content of a struct:
//   firstName = Casper
//   lastName = Schmeichel
.for (var i=0; i<p1.getNoOfFields(); i++) {
    .print p1.getFieldNames().get(i) + “ = “ + p1.get(i)
}
```

Here is a list of the functions defined on struct values:

Table 6.1. Struct Value Functions

Functions	Description
getStructName()	Returns the name of the structure.
getNoOfFields()	Returns the number of defined fields.
getFieldNames()	Returns a list containing the field names.
get(index)	Returns the field value of the field given by an integer index (0 is the first defined filed).
get(name)	Returns the value of the field given by a field name string.
set(index,value)	Sets the value of a field given by an integer index..
set(name,value)	Sets the value of a field given by a name.

6.2. 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("Fairlight", "Booze Design", "etc." )
.byte listSize = greetingsList.size()    // gives .byte 3
```

A compact way to fill a list with elements is:

```
.var greetingsList = List().add("Fairlight", "Booze Design", "etc.")
```

Here is a list of functions defined on list values:

Table 6.2. List Values

Functions	Description
get(n)	Gets the n'th element (first element starts at zero).
set(n,value)	Sets the n'th element (first element starts at zero).
add(value1, value2, …)	Add elements to the end of the list.
addAll(list)	Add all elements from another 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).

6.3. 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 and later, 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.

6.4. 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"))

    // Brief ways of initialising tables
    .var ht2 = Hashtable().put(1, "Yes").put(2, "No")
    .var ht4 = Hashtable().put(1,"a", 2,"b", 3,"c")
}

// 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 values 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.

Table 6.3. Hashtable Values

Function	Description
put(key,value)	Maps 'key' to 'value'. If the key is previously mapped to a value, the previous mapping is lost. The table itself is returned.
put(key,value,key,value,key,value....)	Maps several keys to several values. The table itself is returned.
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.
containsKey(key)	Returns true if the key is defined in the table, otherwise false.
remove(key)	Removes the key and its value from the table.

Chapter 7
Functions and Macros

This chapter shows how to group directives together in units for later execution. In other words, how to define and use functions, macros and finally pseudo commands which are a special kind of macros.

7.1. Functions

You can define you own functions which you can use like any of the build in library functions. Here is an example of a function:

```
.function area(width,height) {
    .return width*height
}
.var x = area(3,2)
lda #10+area(4,8)
```

Functions consist of non-byte generating directives like .eval, .for, .var, and .if. When the assembler evaluates the .return directive it returns the value given by the proceeding 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”
}

// 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)
    }
}

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

You can have several functions of the same name, as long as they have different number of arguments. So this is valid code:

```
.function polyFunction() { .return 0 }
.function polyFunction(a) { .return 1 }
.function polyFunction(a,b) { .return 2 }
```

7.2. 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)
SetColor(2)   // The colon in front of macro calls is optional from version 4.0
```

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 in Kick Assembler are a little more flexible than ordinary macros. They can call other macros or even call themselves - Just make sure there is a condition to stop the recursion so you won't get an endless loop.

7.3. Pseudo Commands

Pseudo commands are a special kind of macros that take 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 colon and you can use any argument you would give to a mnemonic.

Note: In version 3.x, arguments where separated by semicolon. To make old code compile use the -pseudoc3x commandline option or convert the code with the 3.x to 4.x converter.

You can add an optional colon in front of the pseudocommand calls. This enables you to call a command with the same name as a mnemonic.

```
.pseudocommand adc arg1 : arg2 : tar {
    lda arg1
    adc arg2
```

```
    sta tar
}

adc #$10                // This calls the standard mnemonic
:adc #$20 : $10 : $20   // This calls the pseudocommand
```

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:

Table 7.1. CmdValue 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:

Table 7.2. Argument Type Constants

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 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 _16bitnextArgument(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 inc16 arg {
    inc arg
    bne over
    inc _16bitnextArgument(arg)
over:
}

.pseudocommand mov16 src:tar {
    lda src
    sta tar
    lda _16bitnextArgument(src)
    sta _16bitnextArgument(tar)
}

.pseudocommand add16 arg1 : arg2 : tar {
    .if (tar.getType()==AT_NONE) .eval tar=arg1
    clc
    lda arg1
    adc arg2
    sta tar
    lda _16bitnextArgument(arg1)
    adc _16bitnextArgument(arg2)
    sta _16bitnextArgument(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 the first argument is then used as target.

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

Chapter 8
Preprocessor

Before the contents of the source file is handed to the main parser, it goes through the preprocessor. The preprocessor knows nothing of mnemonics or the script language. It's a simple mechanism that enables you to select pieces of the source to be discarded or included in what the main parser sees. This chapter explains how. (NOTE: The preprocessor is made like the one used in C# with the addition of #import, #importif and #importonce so you might find this familiar)

8.1. Defining preprocessor symbols

The preprocessor uses symbols do determine if it should discard or include portions of the source file. There are two methods to define a symbol. The first is from the command line. This defines a symbol called 'TEST':

```
java -jar KickAss.jar -define TEST
```

A symbol is either defined or not defined. It has no assigned value.

The other way is using the #define directive:

```
#define TEST
```

You can recognize a preprocessor directive on the '#'. If the first non-whitespace character on a line is a '#' then the line is a call to the preprocessor. If you want to remove the definition of a symbol you use the #undef directive.

```
#undef TEST
```

8.2. Deciding what gets included

Including or discarding parts of the a source file is done by using #if directives, just like in the script language.

```
// Simple if block
#if TEST
    inc $d020
#endif       // <- Use an endif to close this if block

// You can also use else
#if A
    .print "A is defined"
#else
    .print "A is not defined"
#endif
```

Since the source isn't passed on to the main parser, you can write anything inside an untaken if, and it will still compile.

```
#undef UNDEFINED_SYMBOL
#if UNDEFINED_SYMBOL
    Here we can write anything since it will never be seen by the main parser...
#endif
```

#elif is the combination of an #else and an #if. It can be used like this:

```
#if X
    .print "X"
#elif Y
    .print "Y"
```

```
#elif Z
    .print "Z"
#else
    .print "Not X, Y and Z"
#endif
```

The #if blocks can be nested:

```
#if A
    #if B
        .print "A and B"
    #endif
#else
    #if X
        .print "not A and X"
    #elif Y
        .print "not A and Y"
    #endif
#endif
```

The indentations doesn't change anything, its just to make the code easier to read.

8.3. Importing files

To include another sourcefile in your code, use the #import directive. You can also make a conditional import with the #importif directive.

```
#import "MyLibrary.asm"

#importif STAND_ALONE "UpstartCode.asm" //<- Only imported if STAND_ALONE is
 defined
```

To ensure that a file (e.g. a library) is only imported once, place an #importonce inside the imported file

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

File2.asm:
#import "File1.asm" // This will import File1
#import "File1.asm" // This will not import anything
```

8.4. List of preprocessor directives

All the preprocessor directives are seen here:

Table 8.1. Preprocessor directives

Directive	Description
#define NAME	Defines a preprocessor symbol by the given name
#undef NAME	Removes the symbol definition of the given name, if any.
#import "filename"	Imports a file at the given place in the source.
#importif EXPR "filename"	Imports a file if a given expression evaluates to true.
#importonce	Makes sure the current file is only imported once
#if EXPR	Discards the following source if the given expression evaluates to false.

Directive	Description
#endif	Ends an #if or #else block.
#else	Creates an else block.
#elif EXPR	The combination of an #else and an #if directiveB

8.5. Boolean operators

A symbol works like a boolean. Either its defined or its not. The #if, #elif and #importif directives takes an expression that contains symbols and operators and returns either true of false. Here are some examples:

```
#if !DEBUG && !COMPLICATED
   // some stuff
#endif

#if DEBUG || (X && Y && Z) || X==DEBUG
   // Note that you can also use parenthesis#

#importif DEBUG&&STANDALONE "UpstartWithDebug.asm"
```

Here is a list of operators:

Table 8.2. Preprocessor operators

Operator	Description
!	Negates the expression
&&	Logical and.
||	Logical or.
==	Returns true if the operands are equal.
!=	Returns true if the operands are not equal.
()	Parenthesis can be used to controll order of evaluation

Chapter 9
Scopes and Namespaces

Scopes and namespaces are use to avoid entities like symbols and functions in different parts of the program to collide with each other. This section will cover how they works.

9.1. Scopes

Scopes are containers of symbols (variables, constants and labels). There can only be one symbol of each name in a scope. Scopes are automatically in many situations. For example, a scope is set up when you execute a macro. This prevent the internal labels to collide if you execute the macro twice.

The easiest way to define a scope yourself is using brackets.

```
.var x = 1
{
   .var x = 2  // <- this x won't collide with the previous
}
```

9.2. Namespaces

Namespaces are containers of functions, macros and pseudocommands. There can only be one of each of these entities in namespace. Every namespace also have an its own associated scope so each time you define a namespace a scopes is automatically defined.

A simple way to declare a namespace is shown in the following example. The namespace directives is covered in more detail later (and often the .filenamespace directive is more handy):

```
.function myFunction() { .return 1 }
label1:
.namespace mySpace {
   .function myFunction() { .return 1 } // <- This won't collide
   label1:  <- This won't collide
}
```

Namespace can be declared more than once. The second time you declare it, it will simply continue with the already existing namespace.

```
.namespace repeatedSpace {
   endless: jmp *
   .function myFunc() { return 1}
}

.namespace repeatedSpace { // <- Don't give an error, we reuse the namespace
   jmp endless
   .function myFunc() { return 2} // <-- This gives an error, myFunc is already
 defined
}
```

If you are in doubt of which namespace you are in, you can get its name by the 'getNamespace()' function.

```
.print "Namespace = "+getNamespace()
.namespace MySpace {
   .print "Namespace = "+getNamespace()
   .namespace MySubSpace {
      .print "Namespace = "+getNamespace()
   }
}
```

The above will output:

```
  Namespace = <RootNS>
  Namespace = MySpace
  Namespace = MySpace.MySubSpace
```

9.3. Scoping hierarchy

Namespaces and scopes are organized in an hierarchy. Every namespace have a parent, except for the **system namespace** which is the namespace that contains all the build in functionality of Kick Assembler. Below this is the **root namespace**. As the name implies its the root namespace of the source code.

So the hierarchy is like this:

1. **System namespace & scope** - Contains system mnemonics, constants, functions, macros and pseudocommands.
2. **Root namespace & scope** - The root of the source code.
3. **User defined namespace & scopes** - Created by namespace directives.
4. **User defined scopes** - Created by macros, functions, for-loops, brackets {}, etc.
5. More user defined scopes...

Lets look at an simple example. It contains some scopes and some nonsense code :

```
*=$1000

start:
loop: //<-- 'loop' defined in the root scope

{     //<-- bracket scope 1
loop:
   {  // <-- bracket scope 2
         ldx #0
   loop: stx $d020
         inx
         bne loop
         jmp start
   }
}
```

The above code will form the scope hierarchy: System scope <- Root Scope <- BracketScope1 <- BracketScope2.

When Kick Assembler resolves a symbol, it checks if it is present in the the current scope. If it can't be found it looks in the parent scope. If it still can't be found it looks in the parent scope of the parent and so forth. In the above example, the 'jmp loop' is placed in BracketScope2, so 'loop' is resolved to the loop symbol in BracketScope2. But 'start' is not defined in BracketScope2 or BracketScope1 so it will be resolved to the label in the root scope.

Since no namespaces are defined in the above, the namespace hierarchy is: System namespace <- Root Namespace. The entities of namespaces is resolved similar to the scope resolving mechanism explained above.

9.4. The Namespace Directives

As already seen you can declare namespaces with the namespace directive. When declared it places a symbol inside the scope the parent namespace so the labels inside can be accessed as local fields of the namespace symbol:

```
.namespace vic {
    .label borderColor = $d020
    .label backgroundColor0 = $d021
    .label backgroundColor1 = $d022
```

```
    .label backgroundColor2 = $d023
}

        lda #0
        sta vic.backgroundColor0
        sta vic.borderColor
```

Namespaces are normally used to make sure that code in a source file (Like a library) is not colliding with other parts of the code. For this, Place the filenamespace directive at the top of the file and everything after that is placed in the desired namespace:

```
/* FILE 0 */

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

```
/* FILE 1 */
.filenamespace part1
init:
        ...
        rts

exec:
        ...
        rts
```

```
/* FILE 2 */
.filenamespace part2
init:
        ...
        rts

exec:
        ...
        rts
```

9.5. Escaping the current scope or namespace

To escape the current scope, use @ to reference the root scope. In the following code '@myLabel' access the myLabel label in the root scope:

```
.label myLabel = 1
{
   .label myLabel = 2

   .print "scoped myLabel="+ myLabel //<-- Returns 2
   .print "root myLabel="+ @myLabel  //<-- Returns 1
}
```

The same can be done for functions, macros and pseudo commands. So the following example will print 'root' not 'mySpace':

```
.function myFunction() { .return "root"}
.namespace mySpace {
   .function myFunction() { .return "mySpace" }
   .print @myFunction()
}
```

You can also put new entities in the root scope when defining them from within another scope:

```
   jsr outside_label
   rts
{
@outside_label:

   lda #0
   sta $d020
   sta $d020
   rts
}
```

or:

```
{
   .label @x = 1234
   .var @y= "Hello world"
   .const @z= true
}

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

Or for functions, macros or pseudo commands, here shown in a library file:

```
#import "mylib.lib"

.print myFunction()
MyMacro()
MyPseudoCommand
```

```
/* File mylib.lib */
#importonce
.filenamespace MyLibrary

.function @myFunction() {
   .return 1
}

.macro @MyMacro() {
   .print "Macro Called"
}

.macro @MyPseudoCommand {
   .print "PseudoCommand Called"
}
```

9.6. Label Scopes

If you declare a scope after a label you can access the labels inside the scope as fields on the declared label. This is handy if you use scoping to make the labels of your functions local:

```
        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 use clearScreen.fillbyte 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.

Label scopes also works with the label directive, so its also possible to write programs like this:

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

9.7. Accessing Local Labels of Macros and Pseudocommands

Label scopes are also created when placing a label before a macro or pseudocommand execution as demonstrated in the following program:

```
        *=$1000
start:  inc c1.color
        dec c2.color
c1:     :setColor()
c2:     :setColor()
        jmp start

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

In this way, you can access the labels of an executed macro.

9.8. Accessing Local Labels of For / While loops

By placing a label in front of a for or a while loop, a label scope array is created. This way you can access the inner labels of a loop from the outside or the labels of one loop from another loop:

```
        .for (var i=0; i<20; i++) {
            lda #i
            sta loop2[i].color+1
        }

loop2:  .for (var i=0; i<20; i++) {
color:      lda #0
            sta $d020
        }
```

9.9. Accessing Local Labels of if's

By placing a label in front of an .if directive you can access the labels of the taken branch (true or false) of the directive. The symbol need only to be defined in the taken branch. If the condition is evaluated to false and no false branch exists, all references to symbols give an 'symbol undefined' error . Here is an example:

```
       jmp myIf.label

myIf: .if (true) {
          ...
label:    lda #0  // <-- Jumps here
          ...
      } else {
          ...
label:    nop
          ...
}
```

Chapter 10
Segments

10.1. Introduction

Segments are lists of memory blocks which are used to organize your code. You can use them to define the order which things are placed in memory (data after code etc). Your can combine segments to form new segments and you can use modifiers to process the output of a segment. Finally, you can direct the output of a segment to a files or disks or simply throw it away.

This is implemented in Kick Assembler in a backward compatible way, so if you don't use segments, everything is placed on a default segment and directed to the standard output file as you are used to.

10.2. Some quick examples

Before we go into detail with how segments work, let us take a look at some examples of use. You might not understand everything in the following examples, but it helps to know where we are heading before going into the details.

If you want to have one section of you code output to another file you can assemble it into a segment and write that segment to a file like this:

```
.segment File1 [outPrg="MyFile.prg"]
    *=$1000
    lda #00
    ... more code ...
.segment Default
```

If you want to patch a file you can load the file into a Base segment, put a Patch segment on top of it with the modifications and write the result to a file. Since the Patch is on top it will overwrite the base:

```
.file [name="Out.prg", segments="Base,Patch", allowOverlap]
.segment Base [prgFiles="basefile.prg"]
.segment Patch []

    *=$8021 "Insert jump"
    jmp $8044
```

Segments can also be used for outputting code in alternative formats. Here is an example writing code for a cartridge with 4 banks:

```
.segment CARTRIDGE_FILE [outBin="myfile.bin"]
    .segmentout [segments ="BANK1"]
    .segmentout [segments ="BANK2"]
    .segmentout [segments ="BANK3"]
    .segmentout [segments ="BANK4"]

.segmentdef BANK1 [min=$1000, max=$1fff, fill]
.segmentdef BANK2 [min=$1000, max=$1fff, fill]
.segmentdef BANK3 [min=$1000, max=$1fff, fill]
.segmentdef BANK4 [min=$1000, max=$1fff, fill]

.segment BANK1
..code for segment 1 goes here...

.segment BANK2
..code for segment 2 goes here...

.segment BANK3
..code for segment 3 goes here...
```

```
.segment BANK4
..code for segment 4 goes here...
```

A segment is set up for each bank and they are output in the right order to a binary file. The code in the 4 segments is restricted to the address space $1000-$1fff. Notice how the same address space can be used multiple times, since the code resides in different segments.

10.3. Segments

In Kick Assembler, a segment is a list of memory blocks, so let's look at these first.

A memory block is generated each time you use the *= directive. It has a start, an optional name and might be marked as virtual. If you add code without defining a memory block first, a default block is created for you. Here are examples of memory blocks.

```
        inc $d020         // This create a default memory block
        jmp *-3

        *=$1000           // Start of memoryblock 2 (unnamed)
        lda #1
        sta $d020
        rts

        *=$4000 "block3"  // Start of memoryblock 3
        lda #2
        sta $d021
        rts
```

A segment is a list of memory blocks. Since you haven't selected any segment in the above code, they are all placed on the 'Default' segment.

A segment is defined by the .segmentdef directive and you use the .segment directive to decide which segment to add code to:

```
        // Define two segments
        .segmentdef MySegment1
        .segmentdef MySegment2 [start=$1000]

        // Add code to segment1
        .segment MySegment1
        *=$4000
        ldx #30
 l1:    inc $d021
        dex
        bne l1

        // Add code to segment2 (Using default block starting in $1000)
        .segment MySegment2
        inc $d021
        jmp *-3

        // Switch back to segment1 and add more code.
        .segment MySegment1
        inc $d020
        jmp *-3
```

In the above code MySegment1 is defined used the default parameters for a segment. While MySegment2 is defined setting the start address for the default memory block to $1000. A complete list of parameters is given in the end of this chapter.

Notice that you can switch back to a segment at any time and continue adding code to its current memory block.

Sometimes, it's convenient to define a memory block and switch to it with the same command. This is done by adding a parameters block ([...]) to the segment directive.

```
        // This:
        .segment MySegment [start=$1000]

        // Is a shorthand notations for this:
        .segmentdef MySegment [start=$1000]
        .segment MySegment
```

A segment can only by be defined once so the above will give produce an error saying that 'MySegment' is double defined.

10.4. Where did the output go?

If you compile the previous segment examples you will find that it produces no output. So where did the code go? The answer is nowhere - we defined segments but didn't direct their content anywhere. However we can still see their content using the -bytedump option on the command line when running KickAsssembler. That will generate the file 'ByteDump.txt' with the bytes of the segments. The example from the previous section looks like this:

```
****************************** Segment: Default ******************************
****************************** Segment: MySegment1 ******************************
[Unnamed]
4000: a2 1e     - ldx #30
4002: ee 21 d0  - inc $d021
4005: ca        - dex
4006: d0 fa     - bne l1
4008: ee 20 d0  - inc $d020
400b: 4c 08 40  - jmp *-3
****************************** Segment: MySegment2 ******************************
[MySegment2]
1000: ee 21 d0  - inc $d021
1003: 4c 00 10  - jmp *-3
```

The simplest way of getting the code to a program file is to specify a 'outPrg' parameter:

```
        .segment Code [outPrg="colors.prg"]

        *=$1000
        inc $d020
        jmp *-3
```

If you use the 'outBin' parameter instead a binary file will be output. In the output chapter you can see more options for outputting segments to files or disks images.

10.5. The Default segment

If you don't want to use segments you don't have to. If you don't switch segment using the .segment directive the code is placed on the 'Default' segment which is connected to the the standard output file. In the byte dump in the previous sections you can see the 'Default' segment is empty.

If you want to return the default segment after adding code to another segment simply write:

```
        .segment Default
```

10.6. Naming memory blocks while switching segment

One use of segments is to place code/data that is tied together but should be located different places in memory, close together in the source code. This leads to a coding style where you may want to name a new block of code every time you switch segment. You could do this by adding a memblock directive right after the segment directive. But as a convenient shorthand you can just place a text string after the segment switch:

```
        // This
        .segment Code "My Code"

        // Is the same as this
        .segment Code
        .memblock "My Code"
```

To demonstrate this style is here given a larger example. Some of the features are first covered later. :

```
.segmentdef Code [start=$0900]
.segmentdef Data [start=$8000]
.file [name="segments.prg", segments="Code,Data", modify="BasicUpstart", marg1=
$0900]

//-------------------------------------------------------
// Main
//-------------------------------------------------------
        .segment Code "Main"
        jsr colorSetup
        jsr textSetup
        rts

//-------------------------------------------------------
// Color
//-------------------------------------------------------
        .segment Code "Color Setup"
colorSetup:
        lda colors
        sta $d020
        lda colors+1
        sta $d021
        rts

        .segment Data "Colors"
colors: .byte LIGHT_GRAY, DARK_GRAY


//-------------------------------------------------------
// Text
//-------------------------------------------------------
        .segment Code "Text Setup"
textSetup: {
        ldx #0
loop:   lda text,x
        cmp #$ff
        beq out
        sta $0400,x
        inx
        jmp loop
out:
        rts

        .segment Data "Static Text"
text:   .text "hello world!"
        .byte $ff
}
```

You will now get a memory map like this, when you use the -showmem’ option:

```
Code-segment:
  $0900-$0906 Main
  $0907-$0913 Color Setup
```

```
  $0914-$0924 Text Setup

Data-segment:
  $8000-$8001 Colors
  $8002-$800e Static Text
```

The code and data are now separated in memory, but close together in the source code.

Note that scoping and segments don't affect each other so you can switch segments within a scope. In the above its used so the 'text' label is local. It can be seen from textSetup code but not from other routines. If you want to have a scroll text routine it could have its own 'text' label and they wouldn't collide.

10.7. The default memory block

Code placed inside a segment is added to the default memory block until a block is explicitly defined (Not to be confused with the 'Default' segment):

```
        .segment Code [start=$1000]
        inc $d020         // Places code in the default memoryblock
        jmp *-3

        *=$2000           // Start a new memoryblock
        inc $d021
        jmp *-3
```

The default memory block is special since it can be controlled by parameters given when the segment is defined. Notice the 'start=$1000' parameter that sets the start of the default memory block.

In some cases you want one segment to start after each other. This is done with the 'startAfter' parameter.

```
        .segmentdef Code [start=$1000]
        .segmentdef Data [startAfter="Code"]
```

The ability to control code in this way can be useful, for instance when you want to save memory. If you have some initialization code, that is only used once in the upstart phase, then you could place it after the rest of the code, and use the same memory for a buffer that is used after the init phase:

```
        .file [name="program.prg", segments="Code, InitCode"]

        .segmentdef Code      [start=$1000]
        .segmentdef InitCode  [startAfter="Code"]
        .segmentdef Buffer    [startAfter="Code"]

        .segment Buffer
table1: .fill $100, 0
table2: .fill $100, 0
```

Notice that overlapping code only gives an error if it's inside the same segment. So you can place code in both 'InitCode' and 'Buffer' without getting errors. The Code and InitCode segments are saved in the file while the Buffer is thrown away.

By using the 'align' parameter together with 'startAfter' you align the default memory block.

```
        .segmentdef Code       [start=$8000]
        .segmentdef Virtual100 [startAfter= "Code", align=$100, virtual]

        .segment Code "Some code"
        ldx #$ff
        lda table,x

        .segment Virtual100 "Table"
 table: .fill $100,0
```

By the memory map printed while assembling, you see the start of the Virtual100 segment is aligned to a $100 boundary to avoid spending an extra cycle when accessing the table:

```
Code-segment:
  $8000-$8004 Some code

Virtual100-segment:
  *$8100-$81ff Table
```

In the above example was also used 'virtual' (When no '=' is present its shorthand for 'virtual=true') to declare all the memory blocks in the virtual100 segment virtual. In most cases this won't be necessary since you just don't direct the segment anywhere so the generated bytes are thrown away, but in some cases it can come in handy.

'segmentAfter' works by taking the last defined memory block (Either the default or user defined by *=) and starts where this ends. Block included in other ways (imported from other segments, included from files etc.) are not considered.

10.8. Including other segments

You can include the memory blocks from other segments into the a segment by using the 'segments' parameter when defining the segment:

```
        .segmentdef Upstart [start=$0801]
        .segmentdef Code    [start=$1000]
        .segmentdef Data    [start=$3000]
        .segmentdef Combi1  [segments="Code, Data"]
        .segmentdef Combi2  [segments="Code, Data, Upstart"]
```

A segment can be included in multiple other segment as seen by the 'Code' and 'Data' segment in the above example.

This can be combined freely with adding code from other sources or directly using commands (lda, sta) inside the segment.

10.9. Including .prg files

A prg-file contains a start address (the two first bytes) and some data. Prg files can be imported as memory blocks using the prgFiles parameter when defining the segment:

```
        // Importing prg files when defining segment
        .segmentdef Misc1 [prgFiles="data/Music.prg, data/Charset2x2.prg"]


        // Another way of producing the same result
        .segment Misc2 []
        *=$1000 // Here we have to place the block manually
        .import c64 "data/Music.prg"
        *=$2000 // Here we have to place the block manually
        .import c64 "data/Charset2x2.prg"
```

Again, this can freely be combined with other ways of adding blocks to the segment.

10.10. Including sid files

Sid music files are imported as memory blocks using the 'sidFiles' parameter. Here is an example that plays a sid tune:

```
        .segment Main [sidFiles="data/music.sid", outPrg="out.prg"]
```

```
        BasicUpstart2(start)

start:  sei
        lda #00
        tax
        tay
        jsr $1000

loop:   lda #$f0
        cmp $d012
        bne loop
        inc $d020
        jsr $1003
        dec $d020
        jmp loop
```

10.11. Boundaries

It is possible to set a minimum and maximum address of the segment using the 'min' and 'max' parameters. If a block gets outside the given boundaries, it will give an error:

```
        .segment Data [start=$c000, min=$c000, max=$cfff]

        .fill $1800, 0  // Error since range is $c000-$d7ff
```

In some cases it is useful to ensure a segment have a specific site. By setting the 'fill' parameter to true all non used values in the min-max range is set to the fill byte:

```
        // This will generate $1000: 0,0,1,2,3,0,0,0
        .segment Data [min=$1000, max=$1008, fill]
        *=$1002
        .byte 1,2,3
```

In the above example the fill byte is zero, but it can be specified with the 'fillByte' parameter.

Restricting size can be used to avoid using the ROM area or simply enforcing the rules of a maximum size of 256 or 128 bytes.

The following entry was submitted to the 128 byte font competition on CSDb by Jesper Balman Gravgaard (Rex). It rotates the ROM font 90 degrees. The max size of 128 bytes includes the two address bytes of the prg file.

```
// 90 degree rotated ROM font in 69 bytes of code
.segment Main [min=$0801, max=$0880-2, outPrg="out.prg"]

.label SCREEN = $400
.label CHARGEN = $d000
.label CHARSET = $3000

        *=$0801 "Basic"
        BasicUpstart(ch2)

        *=$080d "Program"
ch4:    dey         // Wait for 8 char lines
        bne ch
        lda pix+1   // Next char
        clc
        adc #8
        sta pix+1
ch2:    sei         // Start char
        lda #$32
        sta $1
        ldy #8
```

```
ch:     lda CHARGEN // Start char line
        ldx #7
npi:    asl         // Start pixel
pix:    rol CHARSET,x
        dex
        bpl npi
        inc ch+1    // Next char line
        bne ch4
        inc pix+2   // Inc both high bytes
        inc ch+2
        bne ch4     // Run until CHARGEN is $0000
ee:
        lda #$37
        sta $1
        cli
        lda #SCREEN/$40|CHARSET/$400
        sta $d018
        rts
```

10.12. Overlapping memory block

When all blocks of a segment are assembled, any overlaps are detected. Normally overlaps will give an error but you can allow overlap with the 'allowOverlap' parameter. This is useful if you want to patch files. Here is an example where the file "base.prg" is applied two changes and saved to the file "patched.prg":

```
        // Setup
        .file [name="patched.prg", segments="Base,Patch", allowOverlap]
        .segmentdef Base [prgFiles="data/base.prg"]
        .segmentdef Patch []

        // Patch Code
        .segment Patch
        *=$3802 "Insert jmp"
        jmp $3fe0

        *=$38c2 "Insert lda #$ff"
        lda #$ff
```

The memory map looks like this:

```
Base-segment:
  $3800-$39ff base.prg

Patch-segment:
  $3802-$3804 Insert jmp
  $38c2-$38c3 Insert lda #$ff
```

In the above example we have a base segment with the original file and a patch segment with the modifications. They are combined in the intermediate segment generated by the file directive which has the allowOverlap parameter set.

Overlapping blocks are cut so the byte from the block with the highest priority are returned. The latest added blocks wins so since the 'Patch' segment lies after 'Base' in the segments list the patch code is chosen.

10.13. Segment Modifiers

The memory block of a segment can be modified before it is passed on to its consumers. A segment-modifier takes a list of memory blocks as input and outputs a modified list of memory blocks.

The build in 'BasicUpstart' modifier adds a memory block in $0801 with a basic upstart program that jumps to a given address:

```
        .file [name="test.prg", segments="Code"]
        .segment Code [start=$8000, modify="BasicUpstart", marg1=$8000]
        inc $d020
        jmp *-3
```

The 'modify' parameter assigns the modifier while 'marg1', 'marg2',..., 'marg5' assigns parameters to the modifier call - in this case marg1 is the start address to jump to.

Users can write their own modifiers as plug-ins (Crunchers etc.) as shown in the plug-in chapter.

Here is a list of build in segment modifiers:

Table 10.1. Build in modifiers

Name	Parameters	Description
BasicUpstart	startAddr	Adds a memory block with a basic upstart program that points to the given start address.

10.14. Intermediate segments

When segments are used in other directives than the .segment and .segmentdef directive its often done using an intermediate segment. Memory blocks are passed on through an implicit created segment which gives you a lot of the functionality explained in this chapter simply by using the same parameters.

E.g. This means that you can use the file directive like this:

```
.file [name="myfile.prg", segments="Code,Data", sidFiles="music.sid"]
```

The only parameter that is special for the file directive is 'name'. The rest is standard parameters for directives using intermediate segments. For a complete list of intermediate parameters see the 'List of segment parameters' section placed last in this chapter.

10.15. The .segmentout directive

The .segmentout directive places the bytes of an intermediate segment in the current memory block. This can be used for reallocating code or data like with the .pseudopc directive. It is also good for outputting data in alternative formats as shown in the 'quick examples' section.

Here is an example that execute some code in the zeropage:

```
        // Main code
        BasicUpstart2(start)
start:  sei
        ldx #0
loop:   lda zpCode,x
        sta zpStart,x
        inx
        cpx #zpCodeSize
        bne loop
        jmp zpStart

zpCode: .segmentout [segments="ZeroPage_Code"]
        .label zpCodeSize = *-zpCode

        // Zeropage code
        .segment ZeroPage_Code [start=$10]
zpStart:
        inc $d020
        jmp *-3
```

In the memory map, you can now see the zeropage code:

```
Memory Map
----------
Default-segment:
  $0801-$080c Basic
  $080e-$0824 Basic End

ZeroPage_Code-segment:
  $0010-$0015 ZeroPage_Code
```

Since the bytes are supplied through an intermediate segment all intermediate parameters can be used. In the following example, a sid file is placed at an alternative address:

```
        *=$8000 "Music Data"
        .segmentout [sidFiles="data/music.sid"]
```

10.16. Debugger data

You can mark segments with a destination using the 'dest' parameter. A destination could be 'DISKDRIVE', 'BANK1', 'BANK2' etc. The parameter doesn't change anything, but is passed on to debuggers that can use the value to organize debug data. For example labels defined in a segment which destination is the disk-drive, should not be mixed with the code which is in the computer. The parameter is used like this:

```
.segmentdef [dest="DISKDRIVE"]
```

The meaning of each destination name is defined by the debugger.

10.17. List of segment parameters

Table 10.2. Segment parameters

Intermediate	Parameter	Example	Description
	align	align=$100	Aligns the default memory block to a given page size. Used together with 'startAfter'
X	allowOverlap	allowOverlap	Allows overlapping memory blocks
	dest	dest="1541"	Set the destination of the segment. (This is info for external programs like C64debugger)
X	fill	fill	Fills unused bytes between min and max with the fill byte
X	fillByte	fillByte=$88	Set the value of the fill byte. If not specified, it will be zero.
X	hide	hide	Hides the segments in memory dumps.
X	marg1, marg2,.., marg5	marg1=$1000, marg2="hello"	Arguments for a modifier.
X	max	max=$cfff	Sets the maximum address of the segment.
X	min	min=$c000	Sets the minimum address of the segment.

Intermediate	Parameter	Example	Description
X	modify	modify="BasicUpstart"	Assigns a modifier to the segment.
	outBin	outBin="myfile.bin"	Outputs a bin-file with the content of the segment.
	outPrg	outPrg="myfile.prg"	Outputs a prg-file with the content of the segment.
X	prgFiles	prgFiles="data/music.prg, data/charset2x2.prg"	Includes program files as memory blocks.
X	segments	segments="Code, Data"	Includes memory blocks from other segments.
X	sidFiles	sidFiles="music.sid"	Include the data of a sid file as a memory block.
	start	start=$1000	Set the start of the default memory block to the given expression
	startAfter	startAfter="Code"	Makes the default memory block start after the given segment.
	virtual	virtual	Makes all the memory blocks in the segment virtual.

Chapter 11
PRG files and D64 Disks

11.1. Introduction

This chapter explains how to create prg-files and d64 disk images using the .file and .disk directive.

The .file directive is quite straight forward, but adds a few extra options over the outPrg parameter for segments.

With the .disk directive you can use Kick Assembler as a standalone disk creation tool, by selecting files from the hard disk to add to a disk image, or you can assemble directly to the disk using segments, or you can mix the two methods.The directive collects parameters and sends them to a disk writer which can either be the build in disk writer or one given by a plug in. The build in default writer is based on the 'CC1541' disk tool by Andreas Larsson, and should cover all needs when creating standard disks. With specialized writers from plugins you can write disks for specific loaders etc.

A big thanks to Andreas for rewriting CC1541 to Java for use in Kick Assembler!

11.2. Parameter Maps

The .file and .disk directives use parameter maps to describe their parameters. These are square brackets with comma separated parameters:

```
[name="Bob", age=27, useGlasses=false, wearsTshirt]
```

You can assign any type of value (strings, numbers, booleans, etc) to a parameter. Notice the last parameter has no assignment. This is a short notation for assigning the boolean value 'true' to the parameter ('wearsTshirt=true').

11.3. The File Directive

The file directive is used like this:

```
        // Save a prg file containing the code segment
        .file [name="MyFile.prg", segments="Code"]

        // Save a bin file containing the code and data segment
        .file [name="MyFile.bin", type="bin", segments="Code,Data"]

        // Save one file for each memoryblock in the DATA segment
        // ('Data_Sinus.prg' and 'Data_Mul3.prg' are created)
        .file [name="Data.prg", mbfiles, segments="Data"]


        // Define some segments
        .segment Code []
        BasicUpstart2(start)
start:  inc $d020
        jmp *-3

        .segment Data []
        *=$0f00 "Mul3"
        .fill $40, i*3

        *=$2000 "Sinus"
        .fill $100, 127.5 + 127.5*sin(toRadians(i*360/256))
```

The content of the file is given using an intermediate segment which makes it quite flexible. See the segment chapter for all options or the disk directive sections for more examples.

The name parameter is mandatory, the rest is optional. Here are the list of specific .file directive parameters:

Table 11.1. File Directive Parameters

Parameter	Default	Example	Description
mbfiles	false	mbfiles	If set to true, a file is created for each memory block.
name		name="MyFile.prg"	The name of the file.
type	"prg"	type="bin"	Sets the file type. Valid types are "prg" and "bin"

11.4. The Disk Directive

The disk directives has the following format:

```
.disk OPT_WRITERNAME [...DISK PARAMETERS..] {
    [..FILE1 PARAMETERS..],
    [..FILE2 PARAMETERS..],
    [..FILE3 PARAMETERS..],
    ....
}
```

The writer name is optional, if left empty the default disk writer is called. Otherwise the writer name is used to look up a 3rd party disk writer imported from a plug in. In the following sections described how the default writer works.

11.5. Disk Parameters

The simplest disk you can create is by only giving the filename of the disk image. The rest of the parameters is then filled out by default values:

```
.disk [filename="MyDisk.d64"]
{
}
```

You fill in extra parameters as a comma separated list. Here we add a disk name and an id, which is displayed in the top of the directory:

```
.disk [filename="MyDisk.d64", name="THE DISK", id="2021!" ]
{
}
```

The complete of possible parameters for the disk is:

Table 11.2. Disk parameters

Parameter	Default	Example	Description
dontSplitFilesOverDir	false	dontSplitFilesOverDir	If set to true, the file that would otherwise have sectors on both sides of the directory track will be moved to after the directory track.
filename		filename="MyDisk.d64"	The name of the disk image
format	"commodore"	format="commodore"	Sets the format of the disk. The options are: "commodore", "speeddos", "dolphindos"
id	" 2A"	id="2021!"	The disk id

Parameter	Default	Example	Description
interleave	10	interleave=10	Sets the default interleave value for the disk
name	"UNNAMED"	name="THE DISK!"	The disk name
showInfo	false	showInfo	Print info about the generated disk after creation. (Start track, sector etc.)
storeFilesInDir	false	storeFilesInDir	If set to true, files can be stored in the sectors of the directory track not used by the directory itself.

11.6. File Parameters

Now let's get some files from different sources on the disk:

```
.disk [filename="MyDisk.d64", name="THE DISK", id="2021!" ]
{
        [name="----------------", type="rel"                            ],
        [name="BORDER COLORS   ", type="prg",  segments="BORDER_COLORS" ],
        [name="BACK COLORS     ", type="prg<", segments="BACK_COLORS"   ],
        [name="HIDDEN          ", type="prg",  hide, segments="HIDDEN"  ],
        [name="----------------", type="rel"                            ],
        [name="MUSIC FROM PRG  ", type="prg", prgFiles="data/music.prg" ],
        [name="MUSIC FROM SID  ", type="prg", sidFiles="data/music.sid" ],
        [name="----------------", type="rel"                            ],
}

        .segment BORDER_COLORS []
        BasicUpstart2(start1)
start1: inc $d020
        jmp *-3

        .segment BACK_COLORS []
        BasicUpstart2(start2)
start2: dec $d021
        jmp *-3

        .segment HIDDEN []
        .text "THIS IS THE HIDDEN MESSAGE!"
```

The content of a file is done using an intermediate segment, which gives a wide range possibilities of specifying input. In the example, the content of the first three prg files comes from the segments specified below. The third uses a prg file from the hard drive and the fourth the content of a sid file. For all the possibilities of working with intermediate segments, see the segments chapter.

The 'name' and 'type' parameters specifies the name and type of the file. Notice the '<' at the end of the second prg type which means the file is locked.

The third prg file is not shown in the directory due to the 'hide' option. You can get its start track and sector by using the 'showInfo' disk parameter.

A complete list of parameters is given here.

Table 11.3. General File parameters

Parameters	Default	Example	Description
hide	false	hide	If set to true, the file will not be shown in the directory.

Parameters	Default	Example	Description
interleave	The disks default	interleave = 10	Sets the interleave of the file.
name	""	name="NOTE"	The filename
type	"prg"	type="prg<"	The type of the file. Available types are: "del", "seq", "prg", "usr", "rel". You can append a "<" to the end of the type to mark it as locked

11.7. Custom Disk Writers

A custom disk writer is written in a plug in. Refer to the "3rd Party Java plugins" if you want to implement one yourself.

It is called like this:

```
.plugin "myplugins.Mydiskwriter"

.disk MyDiskWriter [.. disk params...]
{
        [ ..file params.., segments="Code,Data"],
        [ ..file params.., prgFiles="data/music.prg"],
}
```

Chapter 12
Import and Export

In this chapter we will look at other ways to get data in and out of Kick Assembler.

12.1. 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
```

12.2. 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)
```

The get function extracts signed bytes as defined by java, which means the byte value $ff gives the number -1. This is not a problem when dumping bytes to memory, however if you want to process the data you might want an unsigned byte. To get an unsigned byte use the uget function instead. The byte value $ff will then return 255.

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)
```

Again, file.ugetXCoord(i) will return an unsigned byte.

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)
```

```
        *=$0801 "Basic Program"
        BasicUpstart($0810)

        *=$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 *

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

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

Table 12.1. BinaryFile Constants

Binary format constant	Blocks	Description
BF_C64FILE		A C64 file (The two first bytes are 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 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).

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
```

12.3. 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.

Table 12.2. SIDFileValue Properties

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
        jmp *
//--------------------------------------------------------
irq1:
        asl $d019
        inc $d020
        jsr music.play
        dec $d020
        pla
        tay
        pla
        tax
        pla
        rti
//--------------------------------------------------------
        *=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")
```

12.4. 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.

```
*=$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:

```
*=$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 patters (%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:

```
*=$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:

Table 12.3. PictureValue Functions

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.

Attribute/Function	Description
getSinglecolorByte(x,y)	Converts 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)	Converts 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.)

12.5. 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(*))
}

*=$0801 “Basic”
BasicUpstart(start)

*=$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:

Table 12.4. FileValue Functions

Attribute/Function	Description
Attribute/Function	Description.
writeln(text)	Writes the ‘text’ to the file and insert a line shift.
writeln()	Insert a line shift.

12.6. Exporting Labels to other Sourcefiles

By using the –symbolfile option at the commandline it’s possible export all the assembled symbols. The line

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

will generate the file source1.sym while assembling. Lets say the content of source1 is:

```
.filenamespace source1
        *=$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
```

12.7. Exporting Labels to VICE

By using the –vicesymbols option you can export the labels to a .vs file that can be read by the VICE emulator. For example:

```
java –jar KickAss.jar source1.asm –vicesymbols
```

Chapter 13
Modifiers

With modifiers, you can modify assembled bytes before they are stored to the target file. It could be you want to encrypt, pack or crunch the bytes. Currently, the only way to create a modifier is to implement a java plugin (See the plugin chapter).

13.1. Modify Directives

You can modify the assembled bytes of a limited block or of the whole source file. To modify the whole source file use the .filemodify directive at the top of the file. The following modifies the whole file with the modifier 'MyModifier' called with the parameter 25.

```
.filemodify MyModifier(25)
```

To modify a limited block you use the .modify directive:

```
.modify MyModifier() {
    *=$8080
main:
    inc $d020
    dec $d021
    jmp main

    *=$1000
    .fill $100, i
}
```

Chapter 14
Special Features

Misc features

14.1. Name and path of the sourcefile

You can get the filename and the path of the current sourcefile with the getPath() and getFilename() functions:

```
.print "Path : " + getPath()
.print "Filename : " + getFilename()
```

14.2. 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 standard macro that helps you to create programs like that. The following program shows how it’s used:

```
        *= $0801 "Basic Upstart"
        BasicUpstart(start)    // 10 sys$0810

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

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.

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.

14.3. 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 constant examples:

Table 14.1. Addressing Modes

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

14.4. Colour Constants

Kick Assembler has build in the C64 colour constants:

Table 14.2. 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
```

14.5. Making 3D Calculations

To make it easy to to make 3D Calculations, 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:

Table 14.3. Vector Value Functions

Function/Operator	Example	Description
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 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 transformation matrixes:

Table 14.4. Matrix Value Constructors

Function	Description
Matrix()	Creates an identity matrix.

Function	Description
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:

Table 14.5. Matrix Value Functions

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 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)
//--------------------------------------------------------------------------------
```

Chapter 15
Assemble Information

Kick Assembler 4, and later versions, exposes information of build in features and of the assembled source files. This is intended for authors of editors who want to provide extra support for Kick Assembler such as realtime error and syntax feedback and help text for build in directives and libraries. These features are under development and the interface might change. If you plan to use this get in touch with the author so we can coordinate our efforts.

15.1. The AsmInfo option

To get assemble info back from Kick Assembler, use the -asminfo option:

```
java -jar KickAss.jar mysource.asm -asminfo all
```

When executing the above statement, output is written to the file "asminfo.txt", but you can specify the file by the -asminfofile option:

```
java -jar KickAss.jar mysource.asm -asminfo all -asminfofile myAsmInfo.txt
```

The content of the file will have different sections dependent on what info you have requested. The second parameter describes which info is returned, so in the above example all possible info is returned. The output divided into sections, with different types of information, here is an example:

```
[libraries]
Math;constant;PI
Math;constant;E
Math;function;abs;1
Math;function;acos;1
...
[directives]
.text;.text "hello";Dumps text bytes to memory.
.by;.by $01,$02,$03;An alias for '.byte'.
...
[files]
0;KickAss.jar:/include/autoinclude.asm
1;mySource.asm
[syntax]
symbolReference;38,8,38,17,0
symbolReference;41,20,41,26,0
functionCall;41,8,41,18,0
symbolReference;56,8,56,17,0
...
[errors]
...
```

The details of the sections will be explained later.

There are two categories of asmInfo: Predefined info, which contains information about the features that is build into the assembler like directives and libraries. The other main category is source specific informations, like the syntax of the source or errors in the source. You can turn on one or several categories or sections:

This will export all predefined assemble info sections:

```
java -jar KickAss.jar mysource.asm -asminfo allPredefined
```

And this will export all predefined assemble info sections and any errors:

```
java -jar KickAss.jar mysource.asm -asminfo allPredefined|errors
```

Notice the '|' is used to give several selections - you can add as many as you want. This is the available options:

Table 15.1. AsmInfo

Name	Category	Description
all	meta	Exports all info, both predefined and source specific
allPredefined	meta	All predefined infos
allSourceSpecific	meta	All source specific infos
libraries	predefined	The defined libraries (Functions and constants)
directives	predefined	The defined directives
preprocessorDirectives	predefined	The defined preprocessor directives
files	sourceSpecific	The files involved in the assembling
syntax	sourceSpecific	Syntax info of the given files
errors	sourceSpecific	Errors of the assembling

When the category says 'meta' the option is used to select several of the sections. When the category is not 'meta' the option refers to a specific section. The details of the sections is given in later chapters.

15.2. Realtime feedback from the assembler

For writers of editors Kick Assembler has some special features which enables you to get info about the source file while the user is editing it. This is done by calling Kick Assembler in strategic places like, when the user hasn't typed anything for a given period of time.

First, the content of the one or several source files might not be saved. To get by this, save the content to a temporary file and use the replaceFile option to substitute the content of the original file:

```
java -jar KickAss.jar mysource.asm -replacefile c:\ka\mysource.asm c:\tmp
\tmpSource.asm
```

This replaces the content of the first file with the second. It doesn't matter if the file is the main file or included by another filer, and your can have as many replaceFile options as you want.

Secondly, you don't want Kick Assembler to do a complete assembling each time you call it. It might take too much time to assemble and you don't want the assembler to overwrite output. To take care of this, use the -noeval option.

```
java -jar KickAss.jar mysource.asm -noeval ...
```

This make Kick Assembler parse the source file and do an initial pass, no evaluation will be done. This will detect syntax errors and return syntax information.

15.3. The AsmInfo file format

The assembly info files is divided into sections. If the first char of a line is '[' it marks a new section, and the name of the section is written between square brackets. Each line consist of one or more semicolon separated fields. Notice that in special cases, the last field might contain a semicolon itself (This will be noted in the involved sections). So the basic file format looks like this:

```
[section1]
field1;field2;field3
field1;field2;field3
field1;field2;field3
[section2]
field1;field2
field1;field2
field1;field2
```

As special type of field, which is used in several sections is a 'source range' which describes a range of chars in a source file. It consist of 5 integers:

```
startline, startposition, endline, endposition, fileindex
```

The positions is the positions in a given line. The file index tell which file it is and is an index pointing to an entry in the files section. An example of a source range is:

```
38,8,38,17,1
```

15.4. The sections

Here, the details of the different sections in the asminfo file is explained.

15.4.1. Libraries section

The format of the libraries section are:

```
libraryname;entrytype;typedata
```

There are two entry types: 'function' and 'constant'. The type data depends on the entry type, and is either:

```
libraryname;constant;constantname
libraryname;function;functionname;numberOfArguments
```

Examples:

```
[libraries]
Math;constant;PI
Math;constant;E
Math;function;abs;1
Math;function;acos;1
```

15.4.2. Directives section

The format of the directives section is:

```
directive;example;description
```

Example:

```
[directives]
.text;.text "hello";Dumps text bytes to memory.
```

15.4.3. Preprocessor directives section

The format of the preprocessor directives section is:

```
directive;example;description
```

Example:

```
[ppdirectives]
#define;#define DEBUG;Defines a preprocessor symbol.
```

15.4.4. Files section

The file list section is a list of the involved files. The fields are:

```
index;filepath
```

Important: The file path might contain semicolons!

An example of a list is:

```
[files]
0;KickAss.jar:/include/autoinclude.asm
1;test1.asm
```

Notice the first entry starts with KickAss.jar. This means that its a file included from inside the KickAss.jar file.

15.4.5. Syntax section

The syntax section has the format:

```
type;sourcerange
```

Example:

```
[syntax]
operator;21,20,21,20,0
```

Note: Its the plan to add more fields here, like where a the label is defined if its a label reference, etc.

15.4.6. Errors section

The errors section has the format:

```
level;sourcerange;message
```

Example:

```
[errors]
Error;41,2,41,7,1;Unknown preprocessor directive #defin
```

Chapter 16
Testing

Kick Assembler has .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.

16.1. 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))
```

16.2. 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
```

16.3. 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
```

16.4. 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’
```

Chapter 17
3rd Party Java plugins

It's possible to write you own plugins for Kick Assembler. Currently the following types of plugins are supported:

- *Macro Plugins* - Implements macros
- *Modify Plugins* – Implements modifiers
- *SegmentModifier Plugins* – Implements segment modifiers
- *Archive Plugins* – Used to group multiple plugins in one unit
- *AutoIncludeFile Plugins* – Used to include a source code file in an archive
- *DiskWriter Plugins* – Used to write d64 image disk writers.

17.1. The Test Project

Before going any further I suggest you download the plugin development test eclipse project from the Kick Assembler website.

To use it do the following:

1. Create an Eclipse workspace.
2. 'Import->Existing Projects into workspace->Select archive file' and select the downloaded project file.
3. Replace the KickAss.jar file in the jars folder with the newest version, if necessary.

You are now ready to start. In the src folder you can see examples of how the plugins are made. The files in PluginTest shows how to use them and in the launch folder is launch files for running the examples (Rightclick->Run As).

17.2. Registering your Plugins

To work with plugins you should do two things. When assembling you should make your compiled java class visible from the java classpath. If you are using eclipse to run your Kick Assembler, like in the example project, you don't have to worry about this. If you are using the command line you will have to set either the classpath environment variable or use the classpath option of the java command.

Secondly you should tell Kick Assembler about your plugin. There are two ways to do this. If your plugin is only used in one of your projects, you should use the .plugin directive. Eg:

```
.plugin "test.plugins.macros.MyMacro"
```

If the plugin should be available every time you use Kick Assembler, you place the class name in a line in the file 'KickAss.plugin' which should be placed in the same locations as the KickAss.jar. Using // in the start of the line makes it a comment. Example of a KickAss.plugin file:

```
// My macro plugins
test.plugins.macros.MyMacro1
test.plugins.macros.MyMacro2
test.plugins.macros.MyMacro3
```

17.3. A quick Example (Macros)

First, let's see a quick example of an implemented plugin. To implement a macro plugin you must create a java class that implements the IMacro interface:

```
public interface IMacro extends IPlugin {
   MacroDefinition getDefinition();
   byte[] execute(IValue[] parameters, IEngine engine);
}
```

The interface has two methods, one that gets parameters that defines the macro, and one executes it. This is the basic structure of nearly all the plugins. The MacroDefinition class is really simple. It consist of a getter and setters for the defining properties. Since the only defining property of a macro is its name, it looks like this:

```
public class MacroDefinition {
   // Properties
   private String name;

   // Getters/setters for properties, in this case getName() and setName(name)
   ....
}
```

A simple example of a macro implementation that prints ‘Hello World from MyMacro!’ and returns zero bytes looks like this:

```
package test.plugins.macros;
import kickass.plugins.interf.general.IEngine;
import kickass.plugins.interf.general.IValue;
import kickass.plugins.interf.macro.IMacro;
import kickass.plugins.interf.macro.MacroDefinition;

public class MyMacro implements IMacro {
   MacroDefinition definition;

   public MyMacro() {
      definition = new MacroDefinition();
      definition.setName("MyMacro");
   }

   @Override
   public MacroDefinition getDefinition() {
      return definition;
   }

   @Override
   public byte[] execute(IValue[] parameters, IEngine engine) {
      engine.print("Hello world from mymacro!");
      return new byte[0];
   }
}
```

You execute it as a normal macro:

```
.plugin "test.plugins.macros.MyMacro"
MyMacro()
```

The ‘arguments’ parameter is the arguments parsed to the macro. You can read about these in the 'general communication classes' section. The same goes for the ‘engine’ parameter which is used to do additional communication with the Kick Assembler engine.

17.4. General Communication interfaces

In this section the general interfaces that are used in several plugins are explained. They are all placed in the package 'kickass.plugins.interf.general'. The most important ones are IEngine and IValue. Give them a quick review and return to this chapter when you need info for implementing a particular plugin.

17.4.1. The IEngine Interface

The IEngine interface is the central object when you want to communicate with Kick Assembler. With this you can report errors, print text, create an output stream for outputting a file, etc.

Table 17.1. IEngine Interface

Method	Description
void addError(String message, ISourceRange range);	Adds an error to the engines error list, but continues execution. With this method you can report several errors from your plug in.
byte charToByte(char c);	Converts a char to a petscii byte (upper case).
IMemoryBlock createMemoryBlock(String name, int startAddr, byte[] bytes);	Creates a memory block. Used as result in some plug ins.
void error(String message);	Prints an error message and stops execution. Works like the .error directive. Important! This method will throw an AsmException which you have to pass through any try-catch block used in your code.
void error(String message, ISourceRange);	Like error(string message), buy with a specified position in the code (SourceRange)
File getCurrentDirectory();	Gets the current directory.
File getFile(String filename);	Opens a file with the given filename. The assembler will look for the file as it would look for a source code file. If it isn't present in the current directory, it will look in the library directories. It will return null if the file can't be found.
OutputStream openOutputStream(String name) throws Exception;	Use this to create output from the assembler (like a disk file for a disk writer)
void print(String message);	Prints a message to the screen. Works like the .print directive.
void printNow(String message);	Prints a message to the screen. Works like the .printnow directive.
byte[] stringToBytes(String str);	Converts a string to petscii bytes (Upper case)

17.4.2. The IValue Interface

Objects that implements the interface IValue represents values from Kick Assembler like numbers, strings and booleans. For instance, the arguments given to a macro are given as IValue objects. The IValue interface contains the following methods to extract information from the value:

Table 17.2. IValue Interface

Method	Description
int getInt();	Gets an integer from the value if possible, otherwise it will give an error message.
Double getDouble();	Gets a double from the value if possible, otherwise it will give an error message.
String getString();	Gets a string from the value if possible, otherwise it will give an error message.
Boolean getBoolean();	Gets a Boolean from the value if possible, otherwise it will give an error message.

Method	Description
List<IValue> getList();	Gets at list of values if possible, otherwise it will give an error message. The list implements size(), get(i), isEmpty() and iterator().
Boolean hasIntRepresentation();	Tells if you can get an integer from the value. Every number value can produce an integer. 3.2 will produce 3).
boolean hasDoubleRepresentation();	Tells if you can get a double from the value.
boolean hasStringRepresentation();	Tells if you can get a string from the value.
boolean hasBooleanRepresentation();	Tells if you can get a boolean from the value.
boolean hasListRepresentation();	Tells if you can get a list from the value.

17.4.3. The ISourceRange Interface

The ISourceRange interface represents a position in the source code. An example could be line 17 column 3 to line 17 column 10. These are given to plugins to indicate where it is called from or where certain parameters are defined. The plugin can give them back when reporting errors to indicate what code coursed the error.

Seen from the plugin, the interface is empty:

```
public interface ISourceRange {

}
```

17.4.4. The IMemoryBlock Interface

The IMemoryBlock interface represents a memory block. A block consist of a start address and some byte data. Here are an example of two memory blocks generated by the assembler:

```
*=$1000 "Block 1"
.byte 1,2,3

*=$2000 "Block 2"
lda #1
sta $d020
rts
```

It can either be passed as argument to the plugin or created by the plugin and returned as a result. Use the 'createMemoryBlock' in the IEngine interface to create new memory blocks.

Table 17.3. IMemoryBlock Interface

Method	Description
int getStartAddress()	The start address of the memory block.
byte[] getBytes()	The assembled bytes of the memory block.

17.4.5. The IParameterMap Interface

The IParametersMap interface represent a collection of name-value pairs. The name is a string and the value is of type IValue. These source code parameters are usually defined in square brackets like this:

```
[name="Kevin", age=27, hacker=true]
```

The main methods defined on parameter maps are exists(), getValue(), getSourceRange() and getParameter-Names(). In addition there are some convenience methods for easy retrieval of values of specific types:

Table 17.4. IParameterMap Interface

Method	Description
boolean exist(String paramName)	Tells if a parameter of the given name exists.
boolean getBoolValue(String paramName, boolean defaultValue)	Returns the boolean parameter of the given name. The default is returned in case of an undefined value.
<T extends Enum<?>> T getEnumValue(Class<T> enumeration, String name, T defaultLitteral)	Returns the enum parameter of the given name. The default is returned in case of an undefined value.
int getIntValue(String paramName, int defaultValue)	Returns the int parameter of the given name. The default is returned in case of an undefined value.
Collection<String> getParameterNames()	Returns the names of the defined parameters.
ISourceRange getSourceRange(String paramName)	Returns the position at which this parameter is defined in the source code.
String getStringValue(String paramName, String defaultValue)	Returns the string parameter of the given name. The default is returned in case of an undefined value.
IValue getValue(String paramName)	Returns the value of the parameter with the given name.

17.5. The Plugins

In this section the different plugins are described. Most of them follow a simple pattern: They contain two methods, one for returning a definition for the plugin (name, required parameters, etc.) and one for executing it:

```
interface XYZPlugin extends IPlugin {
   XYZDefinition getDefinition();
   void execute(...);
}
```

The XYZDefinition class simply contains getters and setters for the definition of the plugin, so your getDefinition() method should simply return an XYZDefinition where you have set the fields using the setters (setName("MyPlugin") etc). Many of the definitions only contains a name, but having a definition class makes it easier to extend without breaking backwards compatibility.

You will find that all plugin interfaces extends IPlugin. IPlugin is empty and simply a way of ensuring type safety if you want an object you are sure is a plugin.

17.5.1. Macro Plugins

The interface for a macro looks like this:

```
public interface IMacro extends IPlugin {
   MacroDefinition getDefinition();
   byte[] execute(IValue[] parameters, IEngine engine);
}

public class MacroDefinition {
   // Properties
   private String name;

   // Getters/setters for properties, in this case getName() and setName(name)
   ....
}
```

Macro plugins are described previously in the 'Quick Example' section, so look there for a complete example.

17.5.2. Modifier Plugins

With modifiers you modify the outputted bytes from a section of the code. E.g the following will send the memory block starting at $8080 the the modifier called 'MyModifier' and the returned bytes will be used instead:

```
.modify MyModifier(27) {
    *=$8080
main:
    inc $d020
    jmp main
}
```

You implement a modifier by implementing the following interface. The 'name' in the definition is the modifier name ('MyModifier' in the above example.):

```
public interface IModifier extends IPlugin {
   ModifierDefinition getDefinition();
   byte[] execute(List<IMemoryBlock> memoryBlocks, IValue[] parameters, IEngine
 engine);
}

public class ModifierDefinition {
   private String name;

   // Getters and setters
}
```

Also see the chapter on modifiers.

17.5.3. SegmentModifier plugins

With segment modifiers you can modify the memory block of a segment before it is passed on to its destination. For instance you could implement a packer plugin and have a file packed before it is saved with the command:

```
.file [name="PackedData.prg", segments="Data", modify="MyPacker", marg1=2]
```

A segment modifier is created by implementing a class thats realises the ISegmentModifier plugin:

```
public interface ISegmentModifier extends IPlugin {
   SegmentModifierDefinition getDefinition();
   List<IMemoryBlock> execute(List<IMemoryBlock> memoryBlocks, List<IValue>
 parameters, IEngine engine);
}

public class SegmentModifierDefinition {
   private String name;

   // getters and setters
}
```

See the 'segments' chapter for more about Segment Modifiers.

17.5.4. DiskWriter Plugins

With disk writers you can write disks in a format you decide. Before reading further, read about the the standard disk writer to see what they are able to do. To create a writer you implement a class of the interface IDiskWriter:

```
public interface IDiskWriter extends IPlugin {
   DiskWriterDefinition getDefinition();
   void execute(IDiskData disk, IEngine engine);
}
```

```
public class DiskWriterDefinition {
   private String name;
   private Set<String> allDiskParameters;
   private Set<String> nonOptionalDiskParameters;
   private Set<String> allFileParameters;
   private Set<String> nonOptionalFileParameters;
}
```

Recall the format of the .disk directive to understand the definition properties:

```
.disk WRITERNAME [...DISK PARAMETERS..] {
    [..FILE1 PARAMETERS..],
    [..FILE2 PARAMETERS..],
    [..FILE3 PARAMETERS..],
    ....
}
```

When WRITERNAME matches the name given in the definition the writer is called. Then we have two kinds of parameters: disk and file parameters. For each of these is a set of all possible parameters and a set of non-optional parameters. If a parameter is give that is not included in the allParameters set Kick Assembler will generate an error. The same will happen if a non optional parameter is missing.

The execute method has parameters of two new interfaces:

```
public interface IDiskData {
   IParameterMap getParameters();
   List<IDiskFileData> getFiles();
}

public interface IDiskFileData {
   IParameterMap getParameters();
   List<IMemoryBlock> getMemoryBlocks();
}
```

These represent the given parameters and provides the values and the bytes which should be stored in each file.

When creating the output file, use the IEngine object to open an output stream for storing the bytes. For details, refer to the example project.

17.5.5. Archive Plugins

You can collect more plugins in one archive. This makes it possible to register them with only one plugin directive. To create an archive you implement a class of the IArchive interface:

```
public interface IArchive extends IPlugin {
   public List<IPlugin> getPluginObjects();
}
```

An implementation could look like this:

```
public class MyArchive implements IArchive{
   @Override
   public List<Object> getPluginObjects() {
      List<Object> list = new ArrayList<Object>();
      list.add(new MyMacro());
      list.add(new MyModifyer());
      return list;
   }
}
```

The following plugin directive will then register both MyMacro and MyModifyer.

```
.plugin "test.plugins.archives.MyArchive"
```

17.5.6. AutoIncludeFile Plugins

AutoIncludeFile plugins are used to include source code files in archives. It could be that you want to bundle a source file containing a depack macro together with a segment modifier that packs a segment.

AutoIncludeFile plugins have an interface like all other plugins, but in 99% of all cases you can use the standard implementation included in the KickAssembler jar. Suppose you have a source file (MyAutoInclude.asm) with a macro you want to auto include when importing the archive:

```
//FILE: MyAutoInclude.asm
.macro SetColor(color) {
   lda #color
   sta $d020
}
```

Then you put MyAutoInclude.asm in your jar-file in the package 'include' and add an object of the class AutoIncludeFile to your archive. You archive could look like this:

```
public class MyArchive implements IArchive{

   @Override
   public List<IPlugin> getPluginObjects() {
      ArrayList<IPlugin> plugins = new ArrayList<>();
      plugins.add(new SomePlugin1());
      plugins.add(new SomePlugin2());
      plugins.add(new AutoIncludeFile("MyArcive.jar",getClass(),"/include/
MyAutoInclude.asm"));
      return plugins;
   }
}
```

In the AutoIncludeFile-constructor you give:

1. The jar-name - for use when printing error messages
2. A random 'class'-object from the jar - this is used to open the resource.
3. A path to the resource - the placement inside the jar.

The file will now be compiled with the rest of the source if the archive is imported.

For completeness, here is the IAutoIncludeFile-interface, but as mentioned, you probably wont need it.

```
public interface IAutoIncludeFile extends IPlugin {
   AutoIncludeFileDefinition getDefinition();
   InputStream openStream();
}

public class AutoIncludeFileDefinition {
   private String filePath;
   private String jarName;
}
```

Appendix A. Quick Reference

A.1. Command Line Options

Table A.1. Command Line Options

Option	Example	Description
-afo	-afo	Allows file output outside the output dir.
-aom	-aom	Allow overlapping memory blocks. With this option, overlapping memory blocks will produce a warning instead of an error.
-asminfo	-asminfo all	Turn on exporting of assemble info
-asminfofile	-asminfofile myAsmInfo.txt	Tells where to output the asminfo file.
-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.
-bytedump	-bytebump	Dumps the assembled bytes in the file ByteDump.txt together with the code that generated them.
-bytedumpfile	-bytebumpfile myfile.txt	Same as -bytedump but with an argument specifying the name of the file
-cfgfile	-cfgfile "../../MyConfig.Cfg"	Use additional configuration file (like KickAss.cfg). Supply the file as an absolute path, or a path relative to the source file. You can have as many additional config files as you want.
-debug	-debug	For development use. Adds additional debug info, like stacktraces, to the output.
-debugdump	-debugdump	Dumps an infofile for c64 debugger that maps assembled bytes to locations in the sourcecode.
-define	-define DEBUG	Defines a preprocessor symbol.
-dtv	-dtv	Enables DTV opcodes.
-excludeillegal	-excludeillegal	Exclude the illegal opcodes from the instruction set
-execute	-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.
-executelog	-executelog execlog.txt	If set, this generates a logfile for the output of the program executed using the '-execute' option.

Option	Example	Description
-fillbyte	-fillbyte 255	Sets the byte used to fill the space between memoryblocks in the prg file.
-libdir	-libdir ../stdLib	Defines a library path where the assembler will look when it tries to open external files.
-log	-log logfile.txt	Prints the output of the assembler to a logfile.
-maxaddr	-maxaddr 8191	Sets the upper limit for the memory, default is 65535. Setting a negative value means unlimited memory.
-mbfiles	-mbfiles	One file will be saved for each memory block instead of one big file.
-noeval	-noeval	Parse the sourcecode but exit before evaluation.
-o	-o dots.prg	Sets the output file. Default is the input filename with a '.prg' as suffix.
-odir	-odir out	Sets the output dir. Outputfiles will be output in this dir (or relative to this dir)
-pseudoc3x	-pseudoc3x	Enables semicolon between pseudocommand arguments.
-replacefile	-replacefile c:\source.asm c:\replacement.asm	Replaces a given sourcefile with another everytime it's referred.
-showmem	-showmem	Show a memory map after assembling.
-symbolfile	-symbolfile	Genrates a .sym file with the resolved symbols. The file can be used in other sources with the .import source directive.
-symbolfiledir	-symbolfiledir sources/symbolfiles	Specifies the folder in which the symbolfile is written. If none is given, its written next to the sourcefile.
-time	-time	Displays the assemble time.
-vicesymbols	-vicesymbols	Generates a label file for VICE.
-warningsoff	-warningsoff	Turns off warning messages.
:name=	:x=34 :version=beta2 :path="c:/C64/"	The ':' notation denotes string variables passed to the script. They can be accessed by using the 'cmdLineVars' hashtable which is available from the script.

A.2. Preprocessor Directives

Table A.2. Preprocessor directives

Preprocessor Directives	Example	Description
#define	#define DEBUG	Defines a preprocessor symbol.
#elif	#elif TEST	The combination of an #else and an #if preprocessor directive.

Preprocessor Directives	Example	Description
#else	#else	Used after an #if to start an else block which is executed if the condition is false.
#endif	#endif	Marks the end of an #if/#else block.
#if	#if !DEBUG	Discards the sourcecode after the #if-directive if the condition is false.
#import	#import "file2.asm"	Imports another sourcefile.
#importif	#importif !DEBUG "file2.asm"	Imports another sourcefile if the given expression is evaluated to true.
#importonce	#importonce	Make the assembler skip the current file if it has already been imported.
#undef	#undef DEBUG	Removes the definition of a preprocessor symbol.

A.3. Assembler Directives

Table A.3. Directives

Directive	Example	Description
*	*=$1000	Sets the memory position to a given value.
.align	.align $100	Aligns the memory position with the given value. Ex. '.align $100' at memory position $1234 will set the position to $1300.
.assert	.assert "Test 1",2+2,4	Asserts that two expressions or codeblocks are equal.
.asserterror	.asserterror "Test 2", List().get(27)	Asserts that a given expression or codeblock generates an error.
.break	.break	Puts a breakpoint on the next generated bytes.
.by	.by $01,$02,$03	An alias for '.byte'.
.byte	.byte $01,$02,$03	Outputs bytes.
.const	.const DELAY=7	Defines a constant.
.define	.define width, height {...}	Executes a block of directives in 'functionmode' (faster) to define values.
.disk	.disk [..disk pararamters..] {..fileparameters..}	Creates a d64 image file.
.dw	.dw $12341234	An alias for '.dword'.
.dword	.dword $12341234	Outputs doublewords (4 byte values).
.encoding	.encoding "screencode_upper"	Sets the character encoding.
.enum	.enum {on, off}	Defines a series of constants.
.error	.error "not good!"	Creates an user raised error.
.errorif	.errorif x>10 "not good!"	Creates an user raised error if condition is evaluated to true.

Directive	Example	Description
.eval	.eval x=x+y/2	Evaluates a script expression.
.file	.file [name="myfile.prg" segments="Code, Data"]	Creates a prg or bin file from the given segments.
.filemodify	.filemodify Encrypt(33)	Modify the output of the current source file with the given modifier.
.filenamespace	.filenamespace myspace	Creates a namespace for all the following directives in the current source file.
.fill	.fill 10, i*2	Fills a number of bytes with the value of a given expression.
.for	.for(var i;i<10;i++) {...}	Creates a for loop.
.function	.function area(h,w) {..}	Defines a function.
.if	.if(x>10) {...}	Executes code if the given condition is true.
.import binary	.import binary "Music.bin"	Imports a binary file.
.import c64	.import c64 "Music.c64"	Imports a c64 files. Same as '.import binary', but ignores the two address bytes at the start of the file.
.import source	.import source "MyLib.asm"	Imports another source file. (Depricated, use #import instead)
.import text	.import text "scroll.txt"	Imports a text file.
.importonce	.importonce	Make the assembler skip the current file if it has already been imported. (Depricated, use #importonce instead)
.label	.label color=$d020	Assigns a given expression to a label.
.macro	.macro BasicUpstart() {...}	Defines a macro.
.memblock	.memblock "New block"	Defines a new memoryblock at the current memoryposition.
.modify	.modify Encrypt(27) {...}	Modifies the output of a codeblock using the given modifier.
.namespace	.namespace myspace {..}	Creates a local namespace.
.pc	.pc=$1000	Same as '*='
.plugin	.plugin "plugins.macros.MyMacro"	Tells the assembler to look for a plugin at the given java-package path.
.print	.print "Hello"	Prints a message to the console in the last pass.
.printnow	.printnow "Hello now"	Prints a message to the console immediately.
.pseudocommand	.pseudocommand mov src:tar {...}	Defines a pseudocommand.
.pseudopc	.pseudopc $2000 {...}	Sets the program counter to something else than the actual memory position.
.return	.return 27	Used inside functions to return a value.
.segment	.segment Data "My Data"	Switches to another segment.

Directive	Example	Description
.segmentdef	.segmentdef Data [start=$1000]	Defines a segment.
.segmentout	.segmentout [segments="DRIVE_CODE"]	Output the bytes of an intermediate segment to the current memory-block.
.struct	.struct Point {x,y}	Creates a user defined structure.
.te	.te "hello"	An alias for '.text'.
.text	.text "hello"	Dumps text bytes to memory.
.var	.var x=27	Defines a variable.
.while	.while(i<10) {...}	Creates a while loop.
.wo	.wo $1000,$1012	An alias for '.word'.
.word	.word $1000,$1012	Outputs words (two bytes values),
.zp	.zp { label: .byte 0 ... }	Marks unresolved labels as being in the zeropage.

A.4. Value Types

Table A.4. Value Types

Type	Example	Description
65xxArgument	($10),y	A value that defines an argument given to a mnemnonic.
BinaryFile	LoadBinary("file.bin", "")	A value containing byte data.
Boolean	true	Either true or false.
Char	'x'	A character.
Hashtable	Hashtable()	A value representing a hashtable.
List	List()	A list value.
Matrix	Matrix()	Represents a 4x4 matrix.
Null	null	A null value.
Number	27.4	A floating point number.
OutputFile	createFile("breakpoints.txt")	An value representing an output file.
Picture	LoadPicture("blob.gif")	The contents of a loaded picture.
SidFile	LoadSid("music.sid")	The contents of a sid file.
String	"Hello"	A string value.
Struct	MyStruct(1,2)	Represents a user defined structure.
Vector	Vector(1,2,3)	A 3d vector value.

Appendix B. 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.

B.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.

B.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.

B.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.

B.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. Some examples of invalid value calculations:

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

Appendix C. Going from Version 3.x to 4.0

C.1. The new features

The parser have been rewritten which made some new features possible:

1. You can now use *=$1000 like in good old Turbo Assembler.
2. You can now use soft parenthesis. Kick Assembler will know by the context when it means an indirect adressing mode and when its a normal parenthesis.
3. A preprocessor have been implemented. You can now use the commands #define, #undefine, #if, #else, #elif and #endif (Those who know the C# preprocessor will be familiar with these).
4. There are also preprocessor commands for importing source: #import, #importif, #importonce. #import and #importonce works as the directives known from version 3.x. , but works better together with the preprocessor. #importif supports conditional imports as a simple oneliner.
5. The colon in front of macro and pseudocommand calls are now optional.
6. You can now add an optional ';' after directives and mnemonics. This is usefull if you are use to program languages like C++/Java or C# where these are required.
7. Kick Assembler now report multiple errors in the parsing phase instead of just the first.
8. Kick Assembler can now report syntax elements back to editors. (IN PROGRESS)
9. Kick Assembler can now report syntax errors back to editors, without starting to evaluate the code. (IN PROGRESS)
10. The new parser is faster. The Kick Assembler test suite now assembles in less than half of the time it took when using v3.40.

The scoping/namespace system has been upgraded:

1. Functions, Macros and PseudoCommands are now put in the current namespace when defined. (In 3.x only symbols where scoped)
2. Namespaces can now be reused (Several files can use the same namespace without getting a 'symbol already defined' error).
3. There is now a getNamespace() function that tells the current namespace.
4. Use '@' as prefix when defining a symbol/function/macro/pseudocommand to put it in the root-scope or root-namespace.
5. Use '@' as prefix when referencing a symbol/function/macro/pseudocommand to look it up in the root-scope or root-namespace.
6. NOTICE: There are currently no way of seeing functions/macros/pseudocommands from the outside of a namespace so place your public library functions in the root namespace.
7. Import now always imports to the root scope (Doesn't use the scope at the import call as parent scope)
8. Function/macro/pseudocommand calls now has the definition scope (where the function/macro/pseudocommand is defined) as parent scope during the call. This is consistent with most language like Java, C# etc.

9. All references to a symbol/function/macro/pseudocommand is now resolved in the prepass'. This means you will get errors for misspelled symbols at once. It also means that you can get errors from non-executed code.

10. Resolving symbols in the prepass' gives the same or slightly slower assemble times for performance light sources, but for heavy calculations it is much faster (Example: The fractal2 example from v3.x assembles 38% faster with Kick Assembler 4)

Other news are:

1. There is now a .while directive
2. There is now updated 'quick reference' appendix of options, preprocessor directives, directives and value types.
3. There is now an .encoding directive to switch between petscii/screencode encoding and uppercase/mixedcase.
4. Lines starting with # in KickAss.cfg are now ignored.
5. The source in the manual have been updated
6. The example suite has been rewritten (Its worth a look)
7. A converter to help convert from v3.x to 4.x is included in the distributed zip-file.

C.2. Differences in syntax

There is a small change in the syntax between version 3.x and 4.x, which means that some code might not compile right away - but don't worry, we made a converter to convert sources to the new syntax and have a command line option that will make most code run.

In Kick Assembler 3.x the assembler automatically knows when one command ends and another begins. This means you can write several commands in one line like this:

```
    sei lda #$1b sta $d011 lda #$32 sta $d012
```

In version 4.x you have to separate commands by either a line shift or a semicolon. So in version 4.x the above looks like this:

```
      sei; lda #$1b; sta $d011; lda #$32; sta $d012
```

In general, this is not a problem since you usually put each mnemonic on a separate line. If you want a command to span several lines, use a parenthesis (hard or soft). Since KickAssembler balances the parenthesis sets, only newlines on the outer level will terminate the command so you can write like this:

```
  lda #(
     7 * calculateSomething(a,b)
     + 3 * calculateSomeMore(x,y,z)
)
```

The use of semicolon to terminate commands collide with the old pseudo commands which use the semicolon to separate its arguments. To be compatible with old pseudo commands, use the -pseudo3x option at the command line. You will not be able to write several commands after a pseudocommand call, but your old code will compile. A better option is to convert your code to the new syntax where all semicolons are changed to normal colons. (You can use the converter enclosed in the KickAssembler zip file):

```
// Pseudocommand calls in V3.x
:mov #10 ; data,x

// Pseudocommand calls in V4.x
mov #10 : data,x              // The colon in front is now optional
```

C.3. Difference in behavior

Since all references is now checked prepass, dead code can cause errors. For example, a function that never gets called will now generate an error:

```
.function myFunc1() {
   .var x = unknownSymbol; // Error: Undefined symbol
}
```

If-directives inside functions/defines is now scoped, meaning you can't do like this anymore (This is already the case for .if directives outside functions/defines):

```
.function myFunc1(flag) {
   .if (flag)
      .var message = "flag is true"
   else
      .var message = "flag is false"

   .print message  // Error - 'message' is unknown
}
```

C.4. Converting 3.x sources

To make the transition to from version 3.x to 4 easy, use the converter to convert old source files.

First, take a backup of your source before converting. The source files will be overwritten so its good to have a copy of the original source files. In case there comes updates to the converter, you need the original v3 source code to convert again.

Step one in converting is starting up the converter. This is done by the following command:

```
java -jar KickAss3To4Converter.jar
```

Step two is selecting what to convert. This is done by checking the check boxes in the upper panel. The ones already checked are meant to be converted (You should have a good reason to un check them). The non checked ('Replace .pc with *') are cosmetic changes.

Step three in converting is selecting which source files to convert. To do so, use the 3 buttons:

1. 'Add Files' - Gives you a dialog from which you can pick individual source files.
2. 'Add SourceDir' - Gives you a dialog from which you can add source files of a given type(s) from a source directory and it's subdirectories.
3. 'Remove files' - Removes the selected files of the current file list.

The selected files will appear in the list in the center.

When done, execute the final step by pressing the 'Convert' button, and the conversion will be executed.

The converter will take care of most of the transitions. Currently know issues are:

1. If a command spans more than one line and doesn't contain a kind of parenthesis (soft, hard or curly), you might have to set one as explained in the previous section.