```
//general helper scripts..

//gets hi-byte of 16 bit arguments
.function _16bit_nextArgument(arg) {
    .if (arg.getType() == AT_IMMEDIATE) .return CmdArgument(arg.getType(),
>arg.getValue())
    .if (arg.getType() == AT_IZEROPAGEY || arg.getType() == AT_IZEROPAGEX)
.return arg
    .return CmdArgument(arg.getType(), arg.getValue()+1)
}

//move byte
.pseudocommand mb src;tar {
    lda src
    .if (tar.getType() == AT_INDIRECT) {
        ldy #0
        sta (tar.getValue()),y
    } else {
        sta tar
    }
}

//move word
.pseudocommand mw src;tar;tar2 {
    :mb src;tar
    .var yInced = false
    .if (src.getType() == AT_IZEROPAGEY) {
        iny
        lda tar
        .eval yInced = true
    } else {
        lda _16bit_nextArgument(src)
    }
    .var nextTar
    .if (tar2.getType() != AT_NONE) .eval nextTar = tar2
    else .eval nextTar = _16bit_nextArgument(tar)
    .if (nextTar.getType() == AT_IZEROPAGEY) {
        .if (!yInced) iny
        sta tar
    } else {
        sta nextTar
    }
}

//add word
.pseudocommand aw src;tar {
    :ab src;tar
    .var nextTar = _16bit_nextArgument(tar)
    .if (src.getType() == AT_IZEROPAGEY || tar.getType() == AT_IZEROPAGEY) {
        iny
    }
```

```
    .if (src.getType() == AT_IMMEDIATE && src.getValue() < $80 &&
src.getValue() >= 0
            && tar.getType() != AT_IZEROPAGEY) {
        bcc !+
        inc nextTar
    !:
    } else {
        lda _16bit_nextArgument(src)
        adc nextTar
        sta nextTar
    }
}

//add byte
.pseudocommand ab src;tar {
    clc
    lda src
    adc tar
    sta tar
}

//scripts for code generation...
.var codeOuts   // list of speedcode outputting places that must be updated
on page change
.var offset // keeps track of where in the code segment we are, and can be
used to
        // determine length of the segment
.var codeTarget

.macro startCodeGen(target) {
    .eval codeTarget = target
}

// starts a new segment of speedcode, in this case lineInit/plasmer
.macro newSegment() {
    .eval codeOuts = List()
    .eval offset = 0
    ldy codePnt+0
}

.pseudocommand gc arg1; arg2; arg3; relOffset {
/*  Generates a codegenerator stub for 1-3 bytes of speedcode, based on the
arguments.
    The speedcode generator works like:
    [lda source]
    sta target,y
    Which means that the high byte of the sta's have to be incremented each
time y rolls over.
    For this purpose, the locations of the sta hi-bytes are stored in a list
called codeOuts, which
    can be used for incrementing them when needed.
```

```
    In addition the offset is auto-incremented by 1 for every generated
byte. The offset is located in
    a variable called offset, which can be used afterwards to determine the
length of the code segment.
    Examples:
    ":gc" => no arguments means it will just output the content of the
accumulator =>
    sta target,y
    ":gc #$ea" => output #$ea =>
    lda #$ea
    sta target,y
    ":gc #NOP" => output a nop instruction, which is also $ea => same result
as before. The opcode constants
    are built into KickAss.
    ":gc #LDX_ZP; #$fe" => generate "ldx $fe" =>
    lda #LDX_ZP
    sta target,y
    lda #$fe
    sta target+1,y
    This will increment the offset by 2, where it was only 1 for the
previous examples. Note the # before $fe -
    if this had been left out, the value for the 2nd byte would have been
loaded from $fe.
    ":gc #LDA_IMM; cnt" => generate "lda $xx" where xx is loaded from the
cnt label by the code generator.
    Note the lack of # before cnt =>
    lda #LDA_IMM
    sta target,y
    lda cnt
    sta target+1,y
    All addressing modes can be used for fetching bytes to generate, also
indexed.
    ":gc #LDY_IMM; colors,x" => generate "ldy $xx" where xx is looked up in
the colors table =>
    lda #LDY_IMM
    sta target,y
    lda colors,x
    sta target+1,y
    If the second argument is 16 bits it will result in two generated bytes.
E.g.
    ":gc #INC_ABS; #$d020" => generate "inc $d020" =>
    lda #INC_ABS
    sta target,y
    lda #$20
    sta target+1,y
    lda #$d0
    sta target+2,y
    This currently only works for immediate values, so if the address had to
be looked up you have to
    specify the lo/hi bytes separately:
    ":gc #INC_ABS; adrs+0,x; adrs+1,x" =>
```

```
    lda #INC_ABS
    sta target,y
    lda adrs+0,x
    sta target+1,y
    lda adrs+1,x
    sta target+2,y
    If more advanced methods of calculating bytes are needed, they can be
generated one at a time, like:
    :gc #LDA_IMM
    lda something
    ora somethingElse
    :gc //no args = output accumulator
    Another one...
    :gc #STA_ABS
    lda something
    ora somethingElse
    :gc ; #>address //empty first argument = first output accumulator, then
hi-byte of address.
    Or the relOffset argument can be used for altering bytes ahead/behind
the current offset. If this is set
    the offset will not be auto incremented. In most cases it's probably
easier to use :grc for this though.
    Remember to set the argument that needs to be altered before/after to
empty, in order to avoid redundancy.
    If the code bugs, a good way of debugging is to take a look at the
generated code generator in a monitor.
    The location can be printed at assembly time with .print "here:" +
toHexString(*)
*/
    .eval relOffset = relOffset.getValue()
    .var args = List().add(arg1,arg2,arg3)
    .var progress = 1
    .for (var i=0; i<args.size(); i++) {
        .var arg = args.get(i)
        .if (i==0 || arg.getType() != AT_NONE) {
            .var numBytes = 1
            //16 bit support for 2nd argument...
            .if (i==1 && arg.getValue() >= $100 && arg.getType() ==
AT_IMMEDIATE)
                .eval numBytes = 2
            .for (var b=0; b<numBytes; b++) {
                //only lda if there's a non-empty argument..
                .if (arg.getType() != AT_NONE) {
                    .if (b == 0) lda arg
                    else lda #>arg.getValue()
                }
                sta codeTarget + offset + i + b + relOffset,y
                .eval codeOuts.add(* - 1)
                .eval progress = i + numBytes
            }
        }
```

```
    }
    .if (relOffset==0) .eval offset = offset + progress
}
/*
    Generates a byte (the content of the accumulator) with a relative
offset.
    Can be used together with an empty param for :gc for 3-byte instructions
where the middle one requires
    special handling, e.g. "lda sine,x" with a different lo-byte:
    :gc #LDA_ABSX; ;#>sine    //skips the middle byte
    lda something
    and somethingElse
    :gcr -2            //adds the middle byte
*/
.pseudocommand gcr relOffset {
    :gc ;;;relOffset
}

.macro setCodeOuts(adrs) {
    .for (var i=0; i<adrs.size(); i++)
        sta adrs.get(i)
}
```