Paul Nicholls

6502 Edition

Table of Contents

Introduction

Welcome to **CODE SNIPPETS COOKBOOK** – 6502 Edition written by me, Paul Nicholls (**@syntaxerrorsoft** on twitter). I'm an Indie software developer and budding digital artist who lives in Tasmania, Australia. I love retro computers, especially the Commodore 64 computer.

Where to begin? Well, just before my 12th birthday back in early 1984, my parents purchased a Commodore 64 computer which included a datasette for loading tape games, and a simple programming book, "Spooky Computer Games". They didn't know what they had done, creating a "monster" who couldn't get enough of computers, electronics and similar hobbies.

Fast forward to now, I don't have my original C64 anymore (alas…why did I sell it, smacks head with palm), but a couple of years ago I did get a replacement C64, including the awesome Ultimate 2+ cartridge allowing to load games and programs from virtual folders onto the original hardware for ease of access.

For a while now, I have been writing a platform run-n-jump game for the C64 computer called "**mini MIKE**" and during this time, I have learned many 6502 tricks coding the game.

I wanted to give back to the community by sharing my code snippets (for the 6502 processor), so I'm writing this book. It will have code that should run on all 6502 machines (C64, Atari, NES, SNES, etc.), and will also have code specific to one machine (the C64 computer mainly as this is my main love).

The included code will compile using the great Kick Assembler program that can be found here: http://theweb.dk/KickAssembler/Main.html

I'm going to sell the book very cheap to still make it affordable, and to help keep my retro fires burning.

You can follow along with the code in the book and also view/download the code & example projects from the repository here: https://bitbucket.org/paul_nicholls/code-snippets-cookbook-code/src/master/

Ok, let's get on with it!

Indirect JMP/JSR

Sometimes you might want to jump to a currently unknown address not known at compile time.

One way might be with self-modifying code (see below) which is fine, but this doesn't work if you are in ROM.

```
  jumpPointer: .byte $00,$c0 // lo byte,hi byte of jump address

  lda jumpPointer + 0
  sta jumpAddress + 0
  lda jumpPointer + 1
  sta jumpAddress + 1

  jmp jumpAddress: $ffff
```

Most 6502 processors have an indirect jump opcode like so

```
  jmp (address)
```

where it gets the location to jump to (lo/hi byte) from address + 0 and address + 1 respectively, then jumping to that location. Can be buggy if the lo byte/hi byte **address** word crosses a page boundary causing a crash, so it is sometimes avoided.

Another way is to push the destination address – 1 onto the stack of where you want to jump to, and then use an **rts** instruction to perform the jump.

It is generally 4 bytes smaller, and 1 cycle slower than the indirect jump, but is less buggy.

```
// lo byte,hi byte of jump address - 1
jumpPointer: .byte <jmpAddress-1,>jmpAddress-1

  lda jumpPointer + 1 //high byte of address - 1
  pha
  lda jumpPointer + 0 //low byte of address - 1
  pha
  rts // jump to jmpAddress location
```

An indirect version of **jsr** can be implemented like so where the "indirect jump" needs to be JSR'd to.

```
  jsr doSomeAction
  rts

doSomeAction:
  lda jmpAddress + 1 //high byte of address - 1
  pha
  lda jmpAddress + 0 //low byte of address - 1
```

```
  pha
  rts // jump to jmpAddress location

jmpAddress:
  // do some stuff
  // return back to just after doSomeAction
  rts
```

Jump Tables

The indirect **jmp/jsr** shown previously really shines when you have a bunch of index values that you want to test against and jump to a specific routine matching each value.

You COULD do it this way:

```
  lda index
  cmp #0
  bne !+ // not equal go to next check
  jmp routine0
!:
  cmp #1
  bne !+ // not equal go to next check
  jmp routine1

  …
  cmp #N
  bne !+ // not equal go to next check
  jmp routineN
!:
```

But it would be a maintenance nightmare when adding new routines to the mix, and take up lots of instructions.

A much better way (smaller, faster, and more elegant) is to use a jump table of destination addresses matching the indices being checked against.

Some example code (using the indirect “jsr”) and a separate table for each of the lo and hi bytes of the jump addresses to make it more efficient.

```
  ldx #1 // load index of routine to jump to
  jsr doAction
  rts

doAction:
  lda jumpTable_hi,x
  pha
  lda jumpTable_lo,x
```

```
  pha
  rts // jump to location in stack

// can have up to 256 different routines!!
// addresses must be – 1 from the destination “jump”
// due to using RTS
jumpTable_lo: .byte <routine0-1,<routine1-1,<routine2-1
jumpTable_hi: .byte >routine0-1,>routine1-1,>routine2-1

routine0:
  lda #0
  sta 1024
  rts // return back to the next instruction after “jsr doAction”

routine1:
  lda #1
  sta 1024
  rts // return back to the next instruction after “jsr doAction”

routine2:
  lda #2
  sta 1024
  rts // return back to the next instruction after “jsr doAction”
```

You can get the latest version of the jump tables example project source code from here:
https://bitbucket.org/paul_nicholls/code-snippets-cookbook-code/src/master/src/Jump%20tables/

State Machines

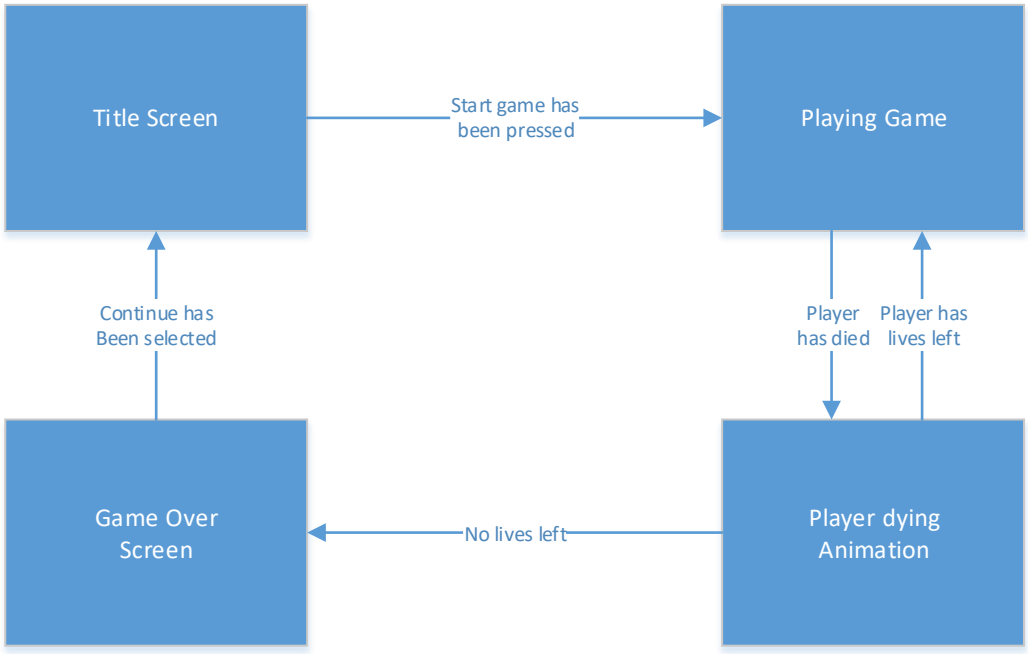
If you are doing any AI work, or even just some programming that requires different states that you need to take care of and switch between, then a state machine can come in very useful.

In a state machine, you have multiple states, but you are only in one state at a time, and when certain conditions are met, you then switch to another state.

In a typical state machine, you would have some routines that get called at different times:

- An "**enter state**" routine – used to initialise or setup required stuff when going to that state.
- An "**exit state**" routine – used to maybe delete or clean up stuff when leaving that state.
- An "**update state**" routine – used to regularly update this state; doing AI, drawing on the screen, etc.

Let's say that you have a game where you have different states, like a "**title screen**", "**playing game**", "**player dying animation**", and "**game over screen**".



Each state with have certain conditions that when met, require switching to another state.

I have created some code that allows one to use a state machine with different states, and to be able to switch between them when necessary, calling the appropriate "enter state", "exit state" code. At the appropriate intervals, you can also call the "update state" routine on the currently active state.

Here is an example bit of code representing a state and it's internal routines like update, enter, exit:

```
// ---------------------------
// STATE_TILESCREEN
// ---------------------------
stateTitlescreen: {
```

```
  onEnter: {
    rts
  }

  onUpdate: {
    rts
  }

  onExit: {
    rts
  }
}
```

Each state will in a real project of course have code added to each of these routines, but that is the bare-bones version of a state ☺

Now, to begin, we will start with defining some constants for each of our states (including invalid states) like so:

```
// state indices
.const STATE_NOSTATE    = -1
.const STATE_TILESCREEN = 0
.const STATE_PLAYING    = 1
.const STATE_DYING      = 2
.const STATE_GAMEOVER   = 3
```

Now we will create a "variable" (like a record in high level languages) to hold our current state, and the routines to change state, call the "**enter state**", "**exit state**", etc.

```
gameState: {
```

Inside **gameState** we will add the current state (initialized to no state):

```
  // current state index
  currentState: .byte STATE_NOSTATE
```

To jump to the appropriate routines; enter, exit, update, we need some jump tables pointing to the appropriate routines (with addresses adjusted by -1). Like previously, each table will have a _lo and _hi part for efficient useage.

Let's start with the "**enter state**" jump table:

```
  // state tables
  onEnterState_lo:
    .byte <stateTitlescreen.onEnter-1,<statePlaying.onEnter-
1,<stateDying.onEnter-1,<stateGameOver.onEnter-1
  onEnterState_hi:
```

```
    .byte >stateTitlescreen.onEnter-1,>statePlaying.onEnter-
1,>stateDying.onEnter-1,>stateGameOver.onEnter-1
```

Similarly, there is a jump table for the “update” routines:

```
  onUpdateState_lo:
    .byte <stateTitlescreen.onUpdate-1,<statePlaying.onUpdate-
1,<stateDying.onUpdate-1,<stateGameOver.onUpdate-1
  onUpdateState_hi:
    .byte >stateTitlescreen.onUpdate-1,>statePlaying.onUpdate-
1,>stateDying.onUpdate-1,>stateGameOver.onUpdate-1
```

And one for the “exit state” routines:

```
  onExitState_lo:
    .byte <stateTitlescreen.onExit-1,<statePlaying.onExit-
1,<stateDying.onExit-1,<stateGameOver.onExit-1
  onExitState_hi:
    .byte >stateTitlescreen.onExit-1,>statePlaying.onExit-
1,>stateDying.onExit-1,>stateGameOver.onExit-1
```

Now we actually need to somehow call the appropriate state’s “onEnter”, “onExit”, and “onUpdate” routines as needed.

Let’s start with the **callOnEnterState** routine:

```
  // call current state's onEnter routine
  callOnEnterState: {
    ldx currentState
    cpx #STATE_NOSTATE
    bne !+
    rts
  !:
    lda onEnterState_hi,x
    pha
    lda onEnterState_lo,x
    pha
    rts
  }
```

This will load the **currentState** index into the **X register** and test for an invalid state (no state), at which point it will exit if found.

If it is a valid state, then the X register will be used as the index offset (**Absolute Indexed**) to load the appropriate **onEnterState** routine address onto the stack in hi/lo format and then call the “**rts**” to “jump” to it. Inside the state, at the end of the routine, another rts will return back to this code which will then exit as normal…nice!

Now for the **onExitState** calling code (almost exact code, but different jump table):

```
  // call current state's onExit routine
  callOnExitState: {
    ldx currentState
    cpx #STATE_NOSTATE
    bne !+
    rts
  !:
    lda onExitState_hi,x
    pha
    lda onExitState_lo,x
    pha
    rts
  }
```

And finally, the **onUpdateState** calling code…very similar again!

```
  // call current state's onUpdate routine
  callOnUpdateState: {
    ldx currentState
    cpx #STATE_NOSTATE
    bne !+
    rts
  !:
    lda onUpdateState_hi,x
    pha
    lda onUpdateState_lo,x
    pha
    rts
  }
```

The next thing part of the puzzle is to actually change state to a new state!
Let's look at the code for that:

```
  // enter new state (if applicable)
  // A reg = new state index
  changeState: {
    cmp currentState
    bne isDifferentState
    // early exit as no change
    rts
```

The code above does this:
If the currentState = the new state (in the A register), then exit right now, otherwise branch to the **isDifferentState** code.

```
  isDifferentState:
    // save new state index
    pha

    lda currentState
    cmp #STATE_NOSTATE
    beq isNoState

    // call onExit routine of existing state (if applicable)
    jsr callOnExitState
```

This code saves the new state index (A register) and checks the **currentState** to see if it is a valid state or not. If it is not, then branch to **isNoState**.

Otherwise call the existing state's "on Exit" routine.

```
  isNoState:
    // restore new state index
    pla
    sta currentState

    // call onEnter routine of new state
    jsr callOnEnterState
    rts
  }
}
```

The rest of the code will restore the new state index and call the relevant "**onEnter**" routine in the state.

Here are all the bare-bones states and their routines (outside of the gameState "record" variable):

```
// states and their routines
// -------------------------
// STATE_TILESCREEN
// -------------------------
stateTitlescreen: {
  onEnter: {
    rts
  }

  onUpdate: {
    rts
  }

  onExit: {
    rts
```

```
  }
}

// ---------------------------
// STATE_PLAYING
// ---------------------------
statePlaying: {
  onEnter: {
    rts
  }

  onUpdate: {
    rts
  }

  onExit: {
    rts
  }
}

// ---------------------------
// STATE_DYING
// ---------------------------
stateDying: {
  onEnter: {
    rts
  }

  onUpdate: {
    rts
  }

  onExit: {
    rts
  }
}

// ---------------------------
// STATE_GAMEOVER
// ---------------------------
stateGameOver: {
  onEnter: {
    rts
  }

  onUpdate: {
    rts
  }
```

```
  onExit: {
    rts
  }
}
```

To make the code more user friendly, I have also create two helper macros, one to change state, and the other to update the current, active state.

```
// helper macros
.macro ChangeState(state) {
  lda #state
  jsr gameState.changeState
}

.macro UpdateState() {
  jsr gameState.callOnUpdateState
}
```

You use them like so:

```
  // test change to initial state with no valid state
  // ie. only calls OnEnter of new state as old state is invalid
  :ChangeState(STATE_TILESCREEN)

  // update active state
  :UpdateState()

  // test changing to different state and calling OnExit
  // of old state and OnEnter of new state
  :ChangeState(STATE_PLAYING)
  // update active state
  :UpdateState()

  // test changing to different state and calling OnExit
  // of old state and OnEnter of new state
  :ChangeState(STATE_DYING)
  // update active state
  :UpdateState()
```

The **UpdateState** macro could be called in a main game loop, or even in an interrupt.

The internal **onUpdate** routine of the active state would handle changing to another state using the **ChangeState**() macro…simple!

You can get the latest version of the state machine code & sample usage project from here:

https://bitbucket.org/paul_nicholls/code-snippets-cookbook-code/src/master/src/State%20Machine/

Events

I'm now going to talk about "Events" where you can setup a routine to be called in approximately N seconds from now (a once off, or retriggerable).

```
#importonce

// maximum number of events; 5 should be plenty!!
.const MAX_EVENTS        = 5

// tables for each part of the event "record"
event_enabled:        .fill MAX_EVENTS,0 // not triggered if not enabled
event_retrigger:      .fill MAX_EVENTS,0 // event will reset automatically
event_counter_lsb:    .fill MAX_EVENTS,0 // counter lsb/msb
event_counter_msb:    .fill MAX_EVENTS,0
event_maxCounter_lsb: .fill MAX_EVENTS,0 // reset counter lsb/msb values to
use
event_maxCounter_msb: .fill MAX_EVENTS,0
event_address_lsb:    .fill MAX_EVENTS,0 // event call address lsb/msb when
triggered
event_address_msb:    .fill MAX_EVENTS,0
```

The code above sets up 5 events maximum (easily changed).
Each event has 5 parts:

- event_enabled – a "Boolean" value; 0 = false, 1 = true. This event won't trigger if false.
- event_retrigger – a "Boolean" value; 0 = false, 1 = true. This event will retrigger automatically if true.
- event_counter_lsb/msb – a 16-bit word that holds the current event counter value counting down. If reaches zero, the event triggers.
- event_maxCounter_lsb/msb – a 16-bit word that holds the maximum event counter value; used when retriggering the event to reset the counter.
- event_address_lsb/msb – a 16-bit word that holds the address of the routine to call when the event is triggered.

The code below defines a useful helper fake instruction pseudo-command to neaten up code when copying byte values around from source to destination.

```
// helper fake instruction to copy a byte of data from
// src to dst;
//   src can be a value or an address, or an instruction operand (ie, $a000,x)
//   dst can be a an address, or an instruction operand (ie, $a000,x)
.pseudocommand copy8 src : dst {
  lda src
  sta dst
}
```

First, we will need some code to find a free event (if any) in the event list when we want to push a new event onto the event list.

The following code does this by scanning the events list, and if it finds a disabled event, it returns the x index value of that event in the accumulator.

Otherwise, it returns 255 as a false “free event not found” value.

```
//------------------------------------------------------
// input:  not used
// output: A & X reg = index of free event index (if not enabled), or 255 if
not found
//------------------------------------------------------
findFreeEvent: {
  ldx #0
loop:
  lda event_enabled,x
  beq foundEvent // this event_enabled = 0 so is free
  inx
  cpx #MAX_EVENTS
  bne loop // not reached MAX_EVENTS so loop back
  lda #255 // free record not found result
  rts
foundEvent:
  txa
  rts
}
```

Next comes the macro to actually push an event onto the event list (fills in a non-used event with the info passed into it).

```
// pushes an event onto the event queue if it can find
// a free slot
//
// seconds       = delay before triggering event.
// eventAddress  = address of routine to call when event is triggered
// reTriggerable = false (0) or true (1); if true, event retriggers every
seconds.
// FPS           = number of times per second it will be triggered; 50 = PAL,
60 = NTSC
.macro PushEvent(seconds,eventAddress,reTriggerable,FPS) {
  .var counterValue = floor(seconds * FPS)

  jsr findFreeEvent
  bmi freeEventNotFound // branch to not found if A = 255 (= -1 in twos-
compliment)

  // must have found a free event so use this one to write data to.
  // copy all data to this event using x as the index
```

```
  :copy8 #1                   : event_enabled,x
  :copy8 #reTriggerable       : event_retrigger,x
  :copy8 #<counterValue       : event_counter_lsb,x
  :copy8 #>counterValue       : event_counter_msb,x
  :copy8 #<counterValue       : event_maxCounter_lsb,x
  :copy8 #>counterValue       : event_maxCounter_msb,x
  :copy8 #<[eventAddress - 1] : event_address_lsb,x
  :copy8 #>[eventAddress - 1] : event_address_msb,x
freeEventNotFound:
}
```

Ok, we might want to clear all events in the event list. The code below does this:

```
// disable all events
clearEvents: {
  ldx #0
loop:
  lda #0
  sta event_enabled,x
  inx
  cpx #MAX_EVENTS
  bne loop
  rts
}
```

Now down to the important part, the event update loop. This should be called at regular intervals like in an interrupt. Let's see some code for this:

```
// updates events that are active, and if counter = 0
// triggers the event address routine
updateEvents: {
  ldx #0
loop:
  lda event_enabled,x
  beq eventNotEnabled

  // event is enabled so check counter = 0
  jsr checkEventCounter
eventNotEnabled:
  inx
  cpx #MAX_EVENTS
  bne loop
  rts
```

That first bit of code will scan through the events list, and if the event is not enabled, will continue on to checking the next event until the end of the list is reached (MAX_EVENTS).

If the current event is enabled, then the event counter will be checked to see if:

a) the event counter has reached zero, thus triggering the event.
b) If the event was triggered, it will then be checked to see if it is needs to be retriggered and it will be done.

```
checkEventCounter:
  lda event_counter_lsb,x
  bne !+           // lsb not zero so update counter
  lda event_counter_msb,x
  beq triggerEvent // msb = 0, so counter = $0000, and event is triggered
  // decrement event counter
  //msb is <> 0 so decrement that by 1
  dec event_counter_msb,x
!:
  // lsb is <> 0 so decrement that by 1
  dec event_counter_lsb,x
  rts
```

The triggerEvent code bellow will disable the event, and then check to see if it is retriggerable. If so, it will reset the counter and re-enable the counter again.

```
triggerEvent:
  // set event to not enabled
  lda #0
  sta event_enabled,x

  lda event_retrigger,x
  beq !+ // not retriggerable so skip next bit
  // is retriggable so reset counter with maxCounter
  :copy8 event_maxCounter_lsb,x : event_counter_lsb,x
  :copy8 event_maxCounter_msb,x : event_counter_msb,x

  // set event to enabled for next update phase
  lda #1
  sta event_enabled,x
!:
```

This code save the x-register, calls the event address of the routine to be triggered, and restores the x-register again.

```
  // save x
  txa
  pha

  jsr callEventAddress
```

```
  //restore x
  pla
  tax
  rts
```

Now finally the actual event routine calling code:

```
callEventAddress:
  // trigger counter event
  lda event_address_msb,x
  pha
  lda event_address_lsb,x
  pha
  rts
}
```

Phew, that was a bit of code, I hope my explanations have helped. You can find the "**events_rtl.asm**" file, and an example usage of the event code (A SPECIFIC C64 MACHINE EXAMPLE) "**events_test.asm**" here:

https://bitbucket.org/paul_nicholls/code-snippets-cookbook-code/src/master/src/Events/

The example code that you can find at the above link updates a single digit on the screen at approximately every second, and updates the border approximately every 0.5 seconds.

Actors (Sprites) and managing them

Now to another topic I hope you will find useful; some techniques to easily update and display up to 8 sprites on the screen at once (ignoring multiplexing) and not having to worry about the sprite's lsb/msb x location on screen.

I'm going be referring to actors as the data that stores each of the sprite's position (x,y), sprite pointer, sprite colour, and whether they are enabled or not.

The sprites are the actual hardware sprites that the each of the actors will map to on a particular system; C64, ATARI, etc.

Now for some code. I'm going to define some tables representing each part of the actor information that we will require; I'm only going to worry about their location, if enabled or not, their colour and sprite pointer (frame), and if they are multi-coloured or not. Other stuff like velocity could be added later.

```
.const MAX_ACTORS      = 8
actor_enabled:     .fill MAX_ACTORS,0

// actor x pos information (lsb/msb)
actor_xlsb:        .fill MAX_ACTORS,0
actor_xmsb:        .fill MAX_ACTORS,0

// actor y pos information (lsb)
actor_ylsb:        .fill MAX_ACTORS,0

// actor color information
actor_multicolor:  .fill MAX_ACTORS,0
actor_color:       .fill MAX_ACTORS,0

// actor frame (sprite pointer) information
actor_frame:       .fill MAX_ACTORS,0
```

Ok, now we need some way of finding a free actor that we can use when spawning an actor into our game/application.

Here is some code that will scan the actor list and find the first free (non-enabled) actor, returning the actor index (0-7) in the x & a register ready to use as the index, or returning 255 in the a register if a free actor is not found.

```
//-------------------------------------------------------------------------
// getFreeActorIndex
//-------------------------------------------------------------------------
// returns 255 in acc if not found,
// otherwise returns x index of free actor
// actor 0 is always the player
//-------------------------------------------------------------------------
getFreeActorIndex: {
  ldx #0 // start at first actor
!:
  lda actor_enabled,x
  beq foundActor // is zero so not used
  inx
  cpx #MAX_ACTORS
  bne !-
  lda #255
  rts
foundActor:
  txa
  rts
}
```

Let's make a **spawnActor** routine to spawn an actor into the actor list (if successful) using variables local to this routine:

```
spawnActor: {
  jsr getFreeActorIndex
  bmi exitSpawnActor // not found as A = 255 (-1)

  // enable actor
  lda #1
  sta actor_enabled,x

  // set actor x pos (lsb/msb)
  lda xPos + 0
  sta actor_xlsb,x

  lda xPos + 1
  sta actor_xmsb,x

  // set actor y pos (lsb only)
  lda yPos
  sta actor_ylsb,x

  // disable actor multicolor
  lda #0
  sta actor_multicolor,x

  // set actor color
  lda color
  sta actor_color,x

  // set actor frame (sprite pointer)
  lda frame
  sta actor_frame,x

exitSpawnActor:
  rts

// local variables for spawnActor
xPos:  .byte 0,0
yPos:  .byte 0
color: .byte 0
frame: .byte 0
}
```

And now for a helper macro to call the above routine with some values:

```
.macro SPAWN_ACTOR(xPos,yPos,color,frame) {
  lda #<xPos
  sta spawnActor.xPos + 0

  lda #>xPos
  sta spawnActor.xPos + 1

  lda #yPos
  sta spawnActor.yPos

  lda #color
  sta spawnActor.color

  lda #frame
  sta spawnActor.frame

  jsr spawnActor
}
```

This would be used like so:

```
  :SPAWN_ACTOR(100,100,1,BALLOON_FRAME)
  :SPAWN_ACTOR(50,50+30,2,BALLOON_FRAME)
  :SPAWN_ACTOR(300,70,3,BALLOON_FRAME)
```

Ok, but this is no fun if we can't actually display the actors on the screen! This means it is time for some code to actually display the actor(s) on screen (if enabled).

I will show a routine only for the C64 as that is the machine I know and love. I will leave as an exercise for the readers to make one for other hardware.

I have a C64 sprites routine I originally found here (https://codebase64.org/doku.php?id=base:moving_sprites&s[]=vic%20registers) that I modified for my needs:

```
// assume default screen address
.const SCREEN_ADDRESS = 1024

vic_copyActorsToHardware: {
  // do sprites 7 downto 0
  ldx #$07 // actor index
  ldy #$0e // sprite register index

loop:
  //write actor y into sprite y register ($d001, $d003, etc...)
```

```
  lda actor_ylsb,x
  sta $d001,y

  //write actor x lsb into sprite x register ($d000, $d002, etc...)
  lda actor_xlsb,x
  sta $d000,y

  lda actor_xmsb,x
  ror                         // rotate actor x msb into carry flag
  rol $d010                   // rotate carry flag into $d010 (sprite x msb bi
t enabled), repeat 8 times and all bits are set

  lda actor_enabled,x
  ror                         // rotate actor enabled lsb into carry flag
  rol $d015                   // rotate carry flag into $d015 (sprite enabled)
, repeat 8 times and all bits are set

  lda actor_multicolor,x
  ror                         // rotate actor muticolor lsb into carry flag
  rol $d01c                   // rotate carry flag into $d01c (sprite multicol
or enabled), repeat 8 times and all bits are set

  // write actor color into sprite color register
  lda actor_color,x
  sta $d027,x

  // sprite pointers; offset from current screen position by 1016
  // write actor sprite pointer (frame) into sprite pointer register
  lda actor_frame,x
  sta SCREEN_ADDRESS + 1016,x

  // decrement y twice as sprite x/y are in staggered pairs
  dey
  dey
  dex
  // not wrapped around to 255 yet?, jump back to loop
  bpl loop
  rts
}
```

Once per frame, you would call the above routine to copy the actor info to the C64's hardware sprite registers, and volia! Easy, peasy sprite handling, at least ignoring velocity and multiplexing!

I have included a sample project for the C64 computer that you can find here: https://bitbucket.org/paul_nicholls/code-snippets-cookbook-code/src/master/src/Actors/

Thanks for reading, I will see you when I do the next topic!