My First Own Demo On

The Atari 2600 Video Computer System

A workshop trying to put together a "hello world"-ish demo in three hours

by Sven Oliver ('SvOlli') Moll

Hackover 2013 - 2013-11-02 - 11:00

Personal background

Got an Atari 2600 in 1982 for X-mas

Sold it two years later to buy a Commodore C=64

Learned 6502 / 6510 assembler there

Bought a 2600 again in the early 90's for nostalgic reasons

Ported 2600 emulator Stella to Sega Dreamcast in 2002 - 2004

Today most coding time is spent on Qt projects

Got interested in the coding for the 2600 in 2011

Motivation for this workshop

- The most retro hardware you can get without spending big money
- Even though it is ancient, there's still much to discover
- Still the most f***ed up hardware I've encountered so far
- Coding is fun, almost like playing a meta-game

Thanks

Thanks to the following sites for providing me with information, supporting me and / or letting me use their content for this talk

http://www.atariage.com/
http://www.biglist.com/lists/stella/
http://www.qotile.net/minidig/
http://www.randomterrain.com/
http://www.console-corner.de/
http://en.wikipedia.org/wiki/Atari_2600

Special big thanks to the folks at AtariAge

Part 1:

What You Need To Start

What You Need To Start

A notebook, PC, Mac, etc. is enough

There's no need for hardware, most time of the development runs inside an emulator

All necessary code is available as source

–> platform independent

So far I've met Windows, MacOS and Linux users

Development "Back Then"

Programming in 1977:

Code assembled on a computer running a proprietary OS

Connected to a special cartridge

When the software crashed, stripes top down would be displayed

For debugging a logic analyzer was used, which could display steps leading to a special condition

What You Need: Software (1)

First of all you'll need an emulator

Probably Stella

Most mature, best maintained

Integrated debugger

Several other features for analyzing the behavior of the hardware (CPU, RIOT, TIA)

Emulation is very good, but not perfect

→ what works in Stella might now work on the 2600

What You Need: Software (2)

For coding you'll need an assembler

Basically there are three choices

– dasm is defacto standard of the 2600 scene

– as65 of cc65

– "all the rest" (acme, nes asm, etc.)

If you don't have reason to pick another assembler specifically, I suggest to go for dasm

For rapid development, Batari Basic might be worth a look

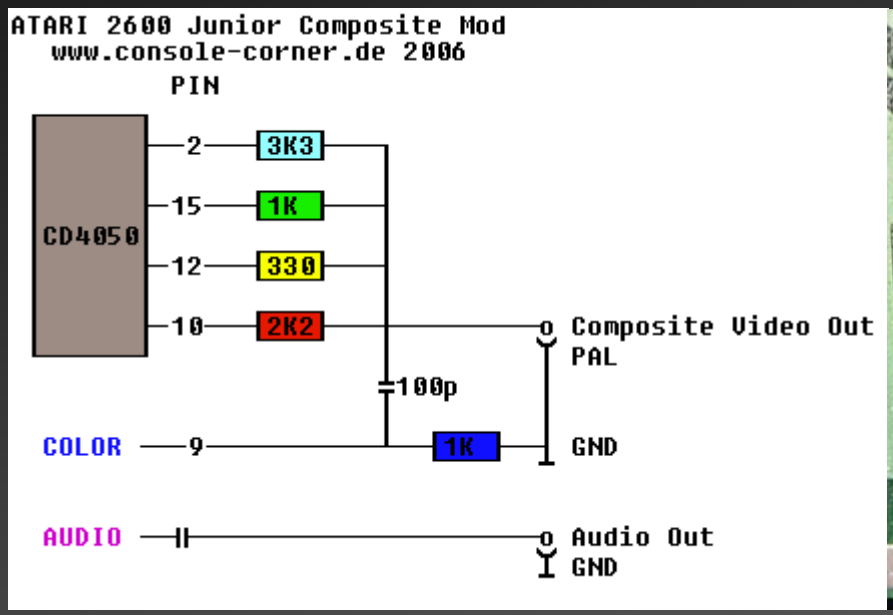
What You Need: Hardware (1)

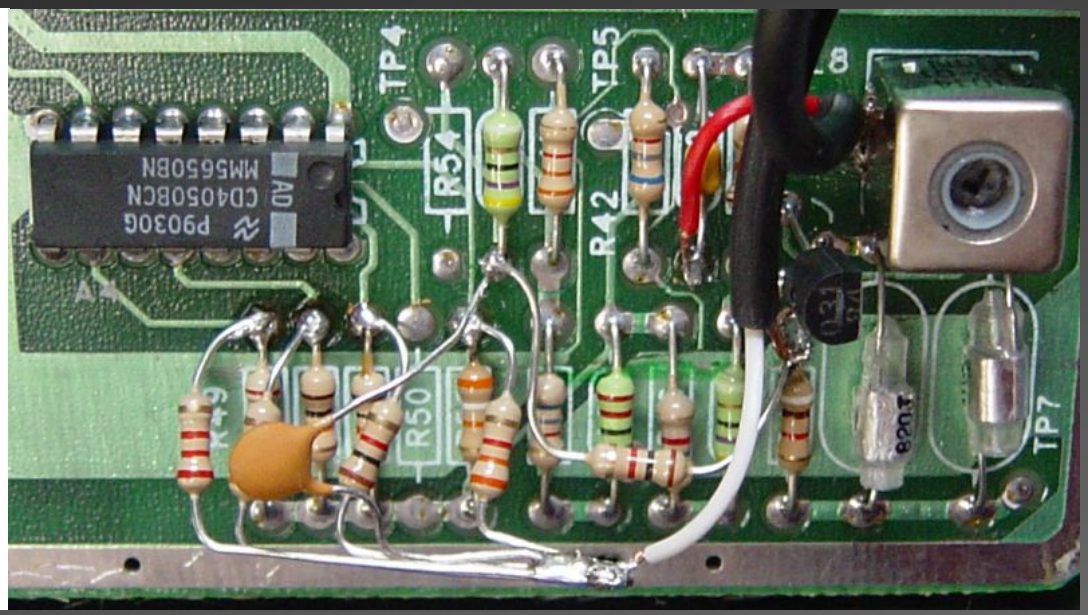
A generic console

For PAL development I suggest looking for a 2600 Jr

It's the easiest to modify with a composite output

http://www.console-corner.de/videomod.html





Images courtesy of www.console-corner.de, used with permission

What You Need: Hardware (2)

Atari 2600 Jr (1984)

Image courtesy of Ewan-Alan, Wikipedia, public domain

What You Need: Hardware (3)

A module you can upload your code to

The defacto-standard is the Harmony Cartridge

Capable of loading ROM data from USB or SD card

What You Need: Hardware (4)

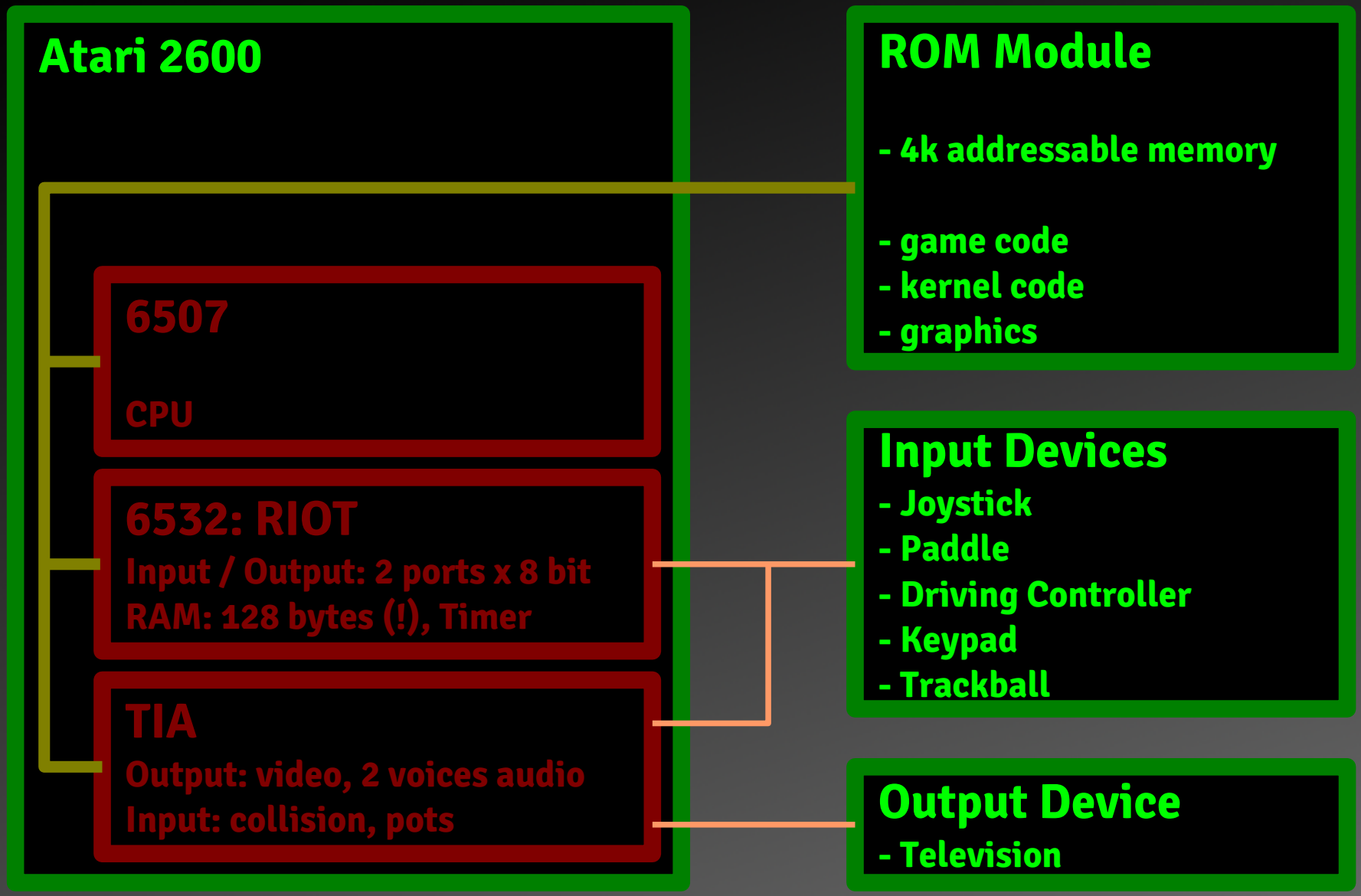
Harmony Cartridge

Image courtesy of Fred Quimby, used with permission

Part 2:

A Look At The Inside

Hardware block diagram



6507: the CPU (1)

The 6507 is a stripped down version of the 6502

Described in depth by Michael Steil on 27c3

Designed by Chuck Peddle, who also worked on the Motorola 6800 team

Clocked at ~1.19MHz

6507: the CPU (2)

Let's compare the 6507 to the 6502:

Smaller chip package (28 pins instead of 40 pins)

What's missing?

3 address lines (64k internal, but only 8k external)

Both interrupt lines are hardwired to +5V internally

1 clock line (phi1), 1 VSS, Sync, S0

3 "n.c." pins ;-)

Even cheaper, popular for embedded applications

6532: RAM, I/O and Timer

Very common companion chip to the 6502 family

128 bytes of RAM

2 I/O ports (8 bit)

- 1 I/O port used for the 5 console switches
- 1 I/O port used for both joysticks
  (only directions, read-write)

Timer that is optionally capable of sending interrupts

(6507 is not capable of receiving interrupts, though)

Memory map (1): overview

External address space of 6507 is 8k

Mirrored 8 times in 64k internal address space

Starting at:

$0000, $2000, $4000, $6000, $8000, $A000, $C000, $E000

$0000 - $0FFF IO, timer and RAM

$1000 - $1FFF ROM (module)

Typically used in two ways:

$0000 - $1FFF

$0000 - $0FFF and $F000 - $FFFF

Memory map (2): ROM

Cartridge port has 24 connectors

Resembling 24 pins of an 32k bit ROM / EPROM

Power: 3 lines: 1x +5V VCC, 2x GND

D0-D7: 8 data lines

A0-A12: 13 address lines

What's missing?

- Chip select: per definition CS is high active A12
- Read / Write: only defined as ROM port (design fail)

Memory map (3): RIOT (1)

Exact mapping: xxx0 xxMx 1NNN NNNN
M: mode (0: RAM 1: I/O+Timer)

RAM: usually accessed at $0080 - $00FF

IO and TIMER: usually accessed at $0280 - $029F

Available 8 times in 8k space, alternating

RAM: $0080, $0180, $0480, $0580, ..., $0C80, $0D80
IO: $0280, $0380, $0680, $0780, ..., $0E80, $0F80

Memory map (4): RIOT (2)

IO-Ports: $0280 (DRA), $0281 (DDRA)
Switches: $0282 (DRB), $0283 (DDRB)

Timer status registers: $0284 - $028C
$0284: read timer (disable interrupt), $028C (enable int.)
$0285: read interrupt flag register (bit 7: timer interrupt)

Interrupt disabled:
$0294 write timer div by 1
$0295 write timer div by 8
$0296 write timer div by 64
$0297 write timer div by 1024

Interrupt enabled:
$029C write timer div by 1
$029D write timer div by 8
$029E write timer div by 64
$029F write timer div by 1024

Memory map (5): RIOT (3)

RAM: 128 bytes

Needed at two locations

– $0080 - $00FF: "variables"
– $0180 - $01FF: stack

Keep in mind that the stack uses a mirror

Quote from development manual:

"The microprocessor stack is normally located from FF on down, and variables are normally located from 80 on up (**hoping the two never meet**)."

Memory map (6): TIA (1)

Exact mapping: xxx0 xxxx 0xNN NNNN

Usually accessed at $0000 - $003F

Available at 32 different positions inside 8k area:

$0000, $0040, $0100, $0140, ..., $0F00, $0F40

"Space" for 64 registers

14 "read only" registers

Mirrored 4 times inside the 64 bytes address space

45 "write only" registers

Memory map (7): TIA (2)

Read registers of the TIA:

CXM0P	CXM1P	CXP0FB	CXP1FB	CXM0FB	CXM1FB
CXBLPF	CXPPMM	INPT0	INPT1	INPT2	INPT3
INPT4	INPT5				

Collision Input

8 registers for collision detection, 6 for input

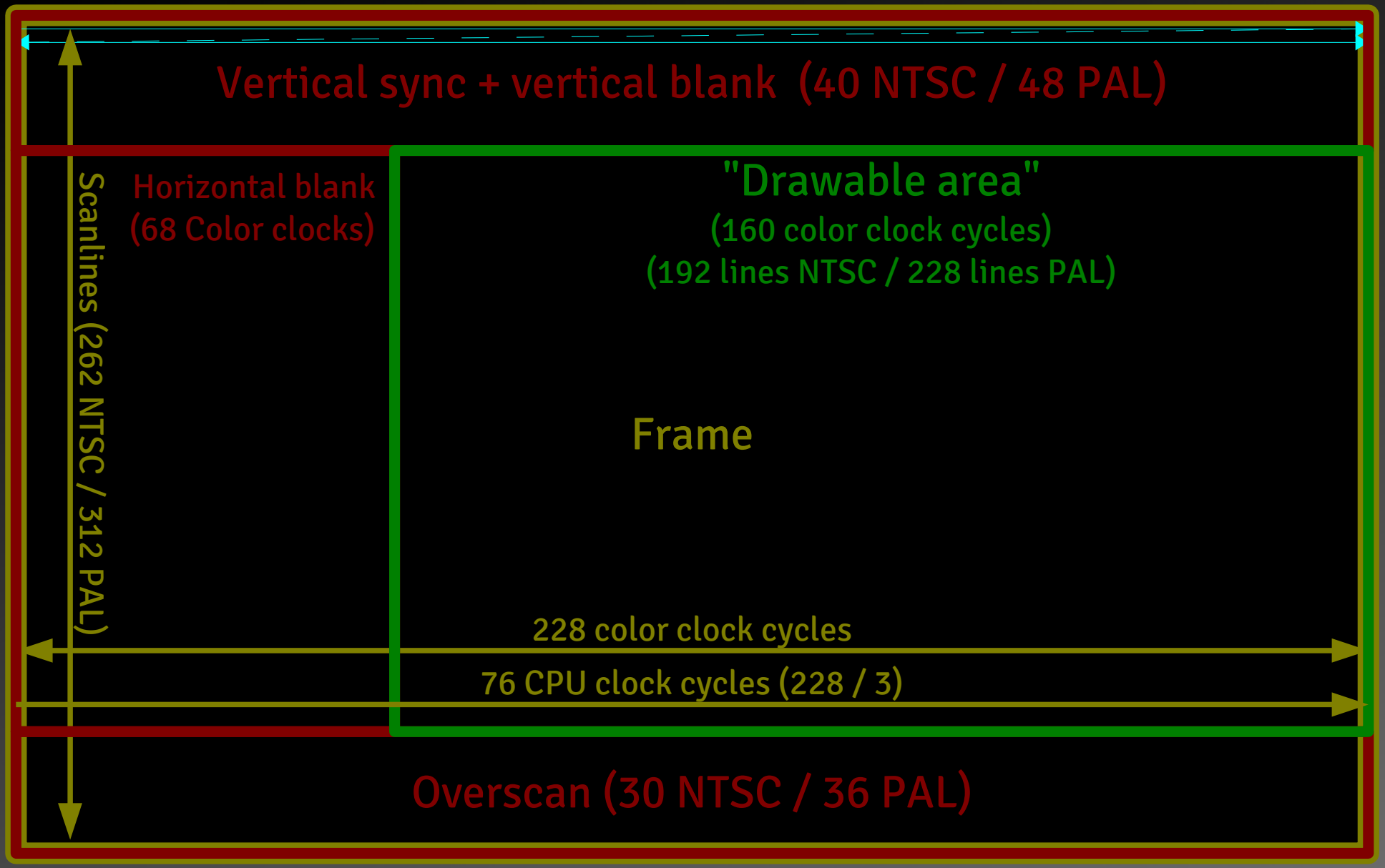
Memory map (8): TIA (3)

Write registers of the TIA:

VSYNC	VBLANK	WSYNC	RSYNC	NUSIZ0	NUSIZ1
COLUP0	COLUP1	COLUPF	COLUBK	CTRLPF	REFP0
REFP1	PF0	PF1	PF2	RESP0	RESP1
RESM0	RESM1	RESBL	AUDC0	AUDC1	AUDF0
AUDF1	AUDV0	AUDV1	GRP0	GRP1	ENAM0
ENAM1	ENABL	HMP0	HMP1	HMM0	HMM1
HMBL	VDELP0	VDELP1	VDELBL	RESMP0	RESMP1
HMOVE	HMCLR	CXCLR		Sync	Graphics

4 registers for syncing, 34 for graphics display

Display


Vertical sync + vertical blank (40 NTSC / 48 PAL)
Horizontal blank
(68 Color clocks)
"Drawable area"
(160 color clock cycles)
(192 lines NTSC / 228 lines PAL)
Scanlines (262 NTSC / 312 PAL)
Frame
228 color clock cycles
76 CPU clock cycles (228 / 3)
Overscan (30 NTSC / 36 PAL)

No Framebuffer

When the Atari 2600 was designed in 1975, RAM was very expensive

To convert the graphics capabilities to a dumb framebuffer you'll need about 30k of 7-bit words

Not only too expensive, but also not addressable by 6507 (8k), and too slow

A completely different approach: program the video chip while the image is displayed

Advantage: cheap and very flexible

Disadvantage: CPU is "occupied" during display

"Racing the beam"

Instead of "running" the graphics frame by frame, the image is drawn line by line

If nothing is changed, the next line is drawn like the one before

There are no registers for Y-components

Example: a player sprite size is 8 bit wide and as high as the screen

You need to tell the TIA what to paint while it is painting! This is called "Racing the beam"

Playfield graphics (1)

Resolution: 40 bits – 4 color clock cycles per bit

Registers responsible for playfield generation:

COLUPF, COLUBK: color

PF0, PF1, PF2: data

How to squeeze this 40 bit resolution into 3 bytes?

CTRLPF: control register

- Bit 0: 1=reflect playfield, 0=repeat playfield
- Bit 1: 1=use player colors, 0=use playfield color
- Bit 2: 1=playfield over sprites, 0=sprites over playfield

Playfield graphics (2)

The data registers in depth:

- PF0: ABCD ----
- PF1: EFGH IJKL
- PF2: MNOP QRST

So the playfield data are only 20 bits that can be

Mirrored: DCBAEFGHIJKLTSRQPONMMNOPQRSTLKJIHGFEABCD

Repeated: DCBAEFGHIJKLTSRQPONMDCBAEFGHIJKLTSRQPONM

Changed: DCBAEFGHIJKLTSRQPONMdcbaefghijkltsrqponm

Note: Intuitive and straight forward to code for, well this isn't

Sprites

The TIA has 5 sprites:

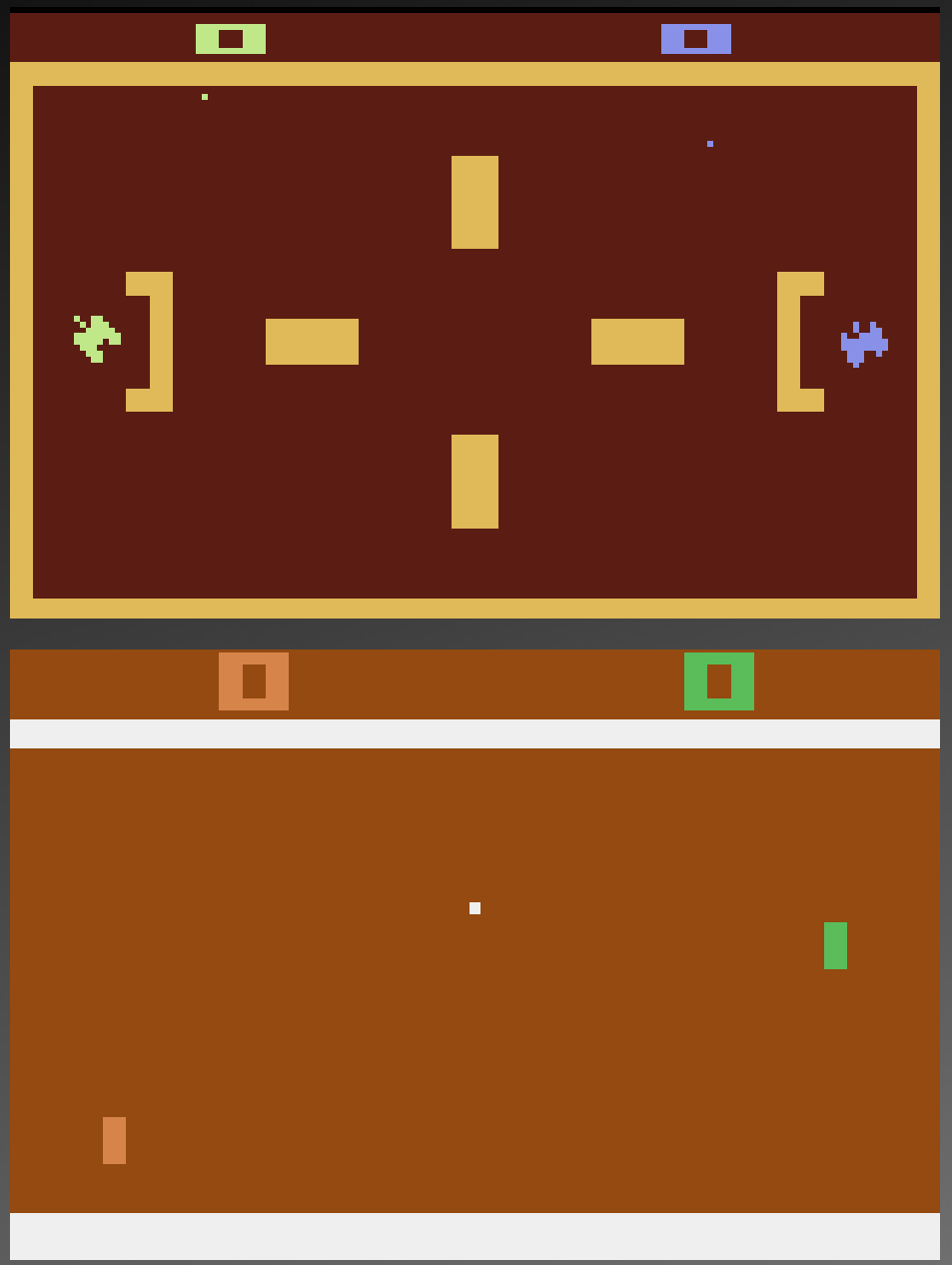
- 2 player sprites (8 bit data)
- 2 missile sprites (1 bit on/off)
- 1 ball sprite (1 bit on/off)

Missile sprite positions can be linked to player positions or positioned independently

Hardware was designed for running

Tank (Combat)

Pong (Video Olympics)



Sprites placement (1)

How are sprites placed on the screen?

Y: enable before beam reaches position

X: more complicated, though

RESP0, RESP1, RESM0, RESM1, RESBL

Reset the sprite position, no value taken

"Reset" has a slightly different interpretation here:

Not reset to position 0, but to current X position of beam

Sprites placement (2)

TIA clock 3 times as fast as CPU clock

Fine-tuning the position:

HMP0, HMP1, HMM0, HMM1, HMBL:

- 4 bit signed motion register
- can move -8 to +7 color clock cycles
- negative moves right, positive left

HMOVE:

- apply motion register settings

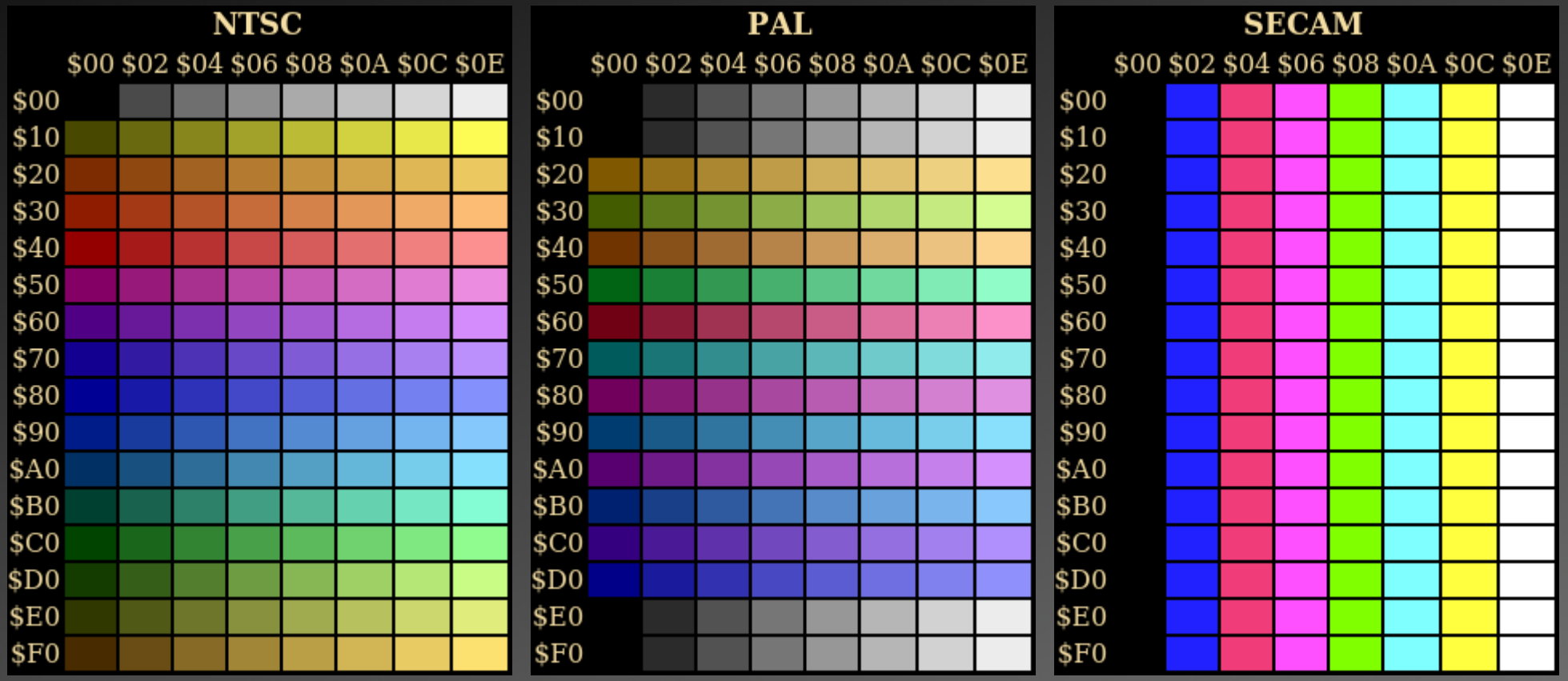
HMCLR:

- clear all HMxx registers at once

Colors (1)

4 Color registers: background, playfield, 2 players

Each color can be picked out of a palette of 128



Colors (2)

COLUBK
– background

COLUPF
– playfield, ball

COLUP0
– player 0, missile 0
– playfield left half (CTRLPF bit 1)

COLUP1
– player 1, missile 1
– playfield right half (CTRLPF bit 1)

NTSC Or PAL?

This question divides into two topics: color and frequency (50 vs 60Hz)

Looking at the color palette, NTSC is better

For demos the frequency of 50Hz is imho better than 60Hz because you've got more rastertime

In games 60Hz gives you a smoother gameplay

I've also seen using the difficulty switches for adjusting color map and frequency in a few games

Keeping In Sync

Since the timing of writing to the registers is essential, it is crucial to know where the beam is

To accomplish this, there are three rules:

1) Count the cycles: of every opcode the time it takes to execute is known
2) Use a write to WSYNC to stop the CPU until the start of a new scanline is reached
3) If you can't predict how long some code will take, start the timer and wait for it to timeout after the work is done → framework does that

Audio

The TIA has 2 voices each having 3 registers

AUDV0, AUDV1: Volume 4 bit

AUDF0, AUDF1: Frequency 5 bit

- Base frequency divided by (AUDFx + 1)

AUDC0, AUDC1: Control 4 bit

- 11 unique settings
- Most of the settings can not be used for music,
  but for sound effects like motor noise, shots, ufos...

Part 3:

A Look At The CPU

6502/6507 Preface

This is just a very incomplete overview of the CPU

- only most important side effects are mentioned

- addressing modes per command is missing

The internet is full of 6502 documentation

- detailed overview of all commands is available at:
  → http://www.oxyron.de/html/opcodes02.html
  (also provided with workshop material)

6502/6507 Architecture

- 8 bit CPU
- 16 bit address bus
- 6 registers
  - accumulator, 8 bit ("A")
  - 2 index registers, 8 bit each ("X" and "Y")
  - processor status, 8 bit ("P")
  - stack pointer, 8 bit ("S")
  - program counter, 16 bit ("PC")

6502/6507 Command-Set

TAX	TAY	TSX	TXS	TXA	TYA	LDA	LDX
LDY	STA	STX	STY	ADC	SBC	AND	ORA
EOR	DEX	INX	DEY	INY	DEC	INC	BIT
SED	CLD	SEC	CLC	SEI	CLI	CLV	CMP
CPX	CPY	ASL	LSR	ROL	ROR	PHA	PLA
PHP	PLP	JMP	JSR	RTS	BRK	RTI	BCC
BCS	BEQ	BNE	BMI	BPL	BVC	BVS	NOP

6502/6507 Addressing Modes (1)

First of all a hint on how to read the following:
- $ indicates a hex value
- # indicates a constant
- () indicates a pointer

A command is one, two or three bytes in size:

- first byte always the command
  → 256 possible, 6502 only "defines" a subset of 151

- second and third byte depend on command

- note that the CPU is little endian

6502/6507 Addressing Modes (2)

implied:

1 byte in size

$1000: NOP → $1000: $EA

6502/6507 Addressing Modes (3)

immediate:

2 bytes in size

$1000: LDA #$2A → $1000: $A9 $2A

A = $2A

6502/6507 Addressing Modes (4)

absolute:

3 bytes in size

$1000: LDA $1234 → $1000: $AD $34 $12

$1234: $2A

A = PEEK($1234) : A = $2A

6502/6507 Addressing Modes (5)

zeropage:

2 bytes in size

$1000: LDA $80 → $1000: $A5 $80

$0080: $2A

A = PEEK($80) : A = $2A

saves one cycle and one byte compared to absolute addressing

6502/6507 Addressing Modes (6)

absolute X indexed:

3 bytes in size

$1000: LDA $1234,X → $BD $34 $12

Read "," as "+": $1234,X → $1234+X

X = $02

$1234: $28 $29 $2A

A = PEEK($1234+X): A = $2A

Note: crossing page boundry "costs" one cycle extra in this case: X > $CC

6502/6507 Addressing Modes (7)

absolute Y indexed:

3 bytes in size

$1000: LDA $1234,Y → $AD $34 $12

Read "," as "+": $1234,Y → $1234+Y

Y = $02

$1234: $28 $29 $2A

A = PEEK($1234+Y): A = $2A

Note: crossing page boundry "costs" one cycle extra in this case: Y > $CC

6502/6507 Addressing Modes (8)

zeropage X indexed:

2 bytes in size

$1000: LDA $81,X → $B5 $81

Read "," as "+": $81,X → $81+X

X = $02

$80: $27 $28 $29 $2A

A = PEEK(($81+X) AND $FF): A = $2A

6502/6507 Addressing Modes (9)

indirect:

3 bytes in size

$1000: JMP ($1234) → $6C $34 $12

Read "()" as a pointer

$1234: $03 $10

GOTO PEEK($1234) + $0100 * PEEK($1235)

Note: does not cross boundry
JMP ($10FF) reads addresses $10FF and $1000

6502/6507 Addressing Modes (10)

X indexed indirect:

2 bytes in size

$1000: LDA ($81,X) → $A1 $81

Read "()" as pointer and "," as "+"

$0080: 2A FF 80 00

X = $02

```
TA = PEEK(($81+X) AND $FF)) + $0100 *
       PEEK(($82+X) AND $FF))
A = PEEK(TA): A = $2A
```

6502/6507 Addressing Modes (11)

indirect Y indexed:

2 bytes in size

$1000: LDA ($81),Y → $A1 $81

Read "()" as pointer and "," as "+"

$0080: FF 80 00 2A

Y = $03

TA = PEEK($81) + $0100 * PEEK($82 AND $FF)
A = PEEK(TA+Y): A = $2A

6502/6507 Addressing Modes (12)

relative

2 bytes in size

$1000: BNE $1000 → $D0 $FE

GOTO "address of following command + operand as signed 8 bit"

Available for conditional branches only

Note: "crossing the page" "costs one cycle extra"

6502/6507 Addressing Modes (13)

- implied: NOP
- immediate: LDA #$2A
- absolute: LDA $1234
- zeropage: LDA $12
- absolute X indexed: LDA $1234,X
- zeropage X indexed: LDA $89,X
- absolute Y indexed: LDA $1234,Y
- indirect: JMP ($1234)
- X indexed indirect: LDA ($89,X)
- indirect Y indexed: LDA ($89),Y
- relative: BNE $ABCD

6502/6507 Commands (1)

Transfer commands

Example: TAX

Read as: "Transfer A to X"

Available as:

TAX, TXA, TAY, TYA, TXS, TSX

6502/6507 Commands (2)

Set/clear processor flags commands

Example: SEC

Read as: "SEt Carry"

Available as:

SEC, CLC, SED, CLD, SEI, CLI, CLV

6502/6507 Commands (3)

Push / pull commands (to / from stack)

Example: PHA

Read as: "PusH Accu to stack"

Available as:

PHA, PLA, PHP, PLP

6502/6507 Commands (4)

Load commands

Example: LDA #$2A

Read as: "LoaD Accu with the value $2A"

Available as:

LDA, LDX, LDY

6502/6507 Commands (5)

Store commands

Example: STA $1234

Read as: "STore Accu to the address $1234"

Available as:

STA, STX, STY

6502/6507 Commands (6)

Binary commands

Example: AND #$2A

Read as: "AND accu with the value $2A"

Example: ORA #$2A

Read as: "OR Accu with the value $2A"

Example: EOR #$2A

Read as: "Exclusive OR accu with the value $2A"

6502/6507 Commands (7)

Arithmetic commands

Example: ADC #$2A

Read as: "ADd with Carry to accu the value $2A"

Example: SBC #$2A

Read as: "SuBtract with Carry from accu
the value $2A"

6502/6507 Commands (8)

Compare commands

Example: CMP #$2A

Read as: "CoMPare the accu the value $2A"

Internally does a subtraction without storing result in accu, just the flags are set

Available as:

CMP, CPX, CPY

6502/6507 Commands (9)

Increment/decrement commands

Example: INX

Read as: "INcrement X register"

Available as:

INX, DEX, INY, DEY, INC, DEC

6502/6507 Commands (9)

Increment/decrement commands

Example: INX

Read as: "INcrement X register"

Available as:

INX, DEX, INY, DEY, INC, DEC

6502/6507 Commands (10)

Bitshift commands

Example: ROL

Read as: "ROtate Left"

Multiplies by 2 and move highest bit to carry and the carry is moved in as the lowest bit

Available as:

ROL, ROR, ASL, LSR
all commands are available using the accu or modifying a memory address

6502/6507 Commands (11)

Conditional branch commands

Example: BNE $1000

Read as: "Branch Not Equal"

Available as:

BEQ, BNE, BPL, BMI, BCC, BCS, BVC, BVS

6502/6507 Commands (12)

Jump commands

Example: JSR $1000

Read as: "Jump to SubRoutine at"

Available as:

JMP, JSR, RTS, BRK, RTI

6502/6507 Commands (13)

Other commands

Example: NOP

Read as: "No OPeration"

Example: BIT $1234

Read as: "test BITs in memory"

Note: often used as 2- or 3-byte NOP

6502/6507 Commands (14)

Illegal commands (better known as illegal opcodes)

As stated only 151 of 256 possible values for a byte are defined as commands

What happens if the CPU wants to "execute" a not defined value?

Most times either a crash of the CPU or execution of a "combined" functionality, e.g. LAX

6502/6507 Command Execution Time

How long does it take for a command to be executed?

It takes 2-7 cycles (8 for some illegal opcodes)

Depends mostly on addressing mode, and if the command is read/modify/write or not

On the same addressing mode LDA uses the same amount of clock cycles as LDX, ADC, ORA, STA, etc.

Part 4:

How To Get Things Done

The First Program

Will most probably look something like this

Doesn't look like much, and still you will be proud when it works!

Why?

Because you've already mastered to overcome your first limitation of the 2600

The First Program

Use some existing code to start your project

– reading code of other people is a good start

– a binary is almost as good as the source code

– a template and an example is provided

For a working ROM you need:

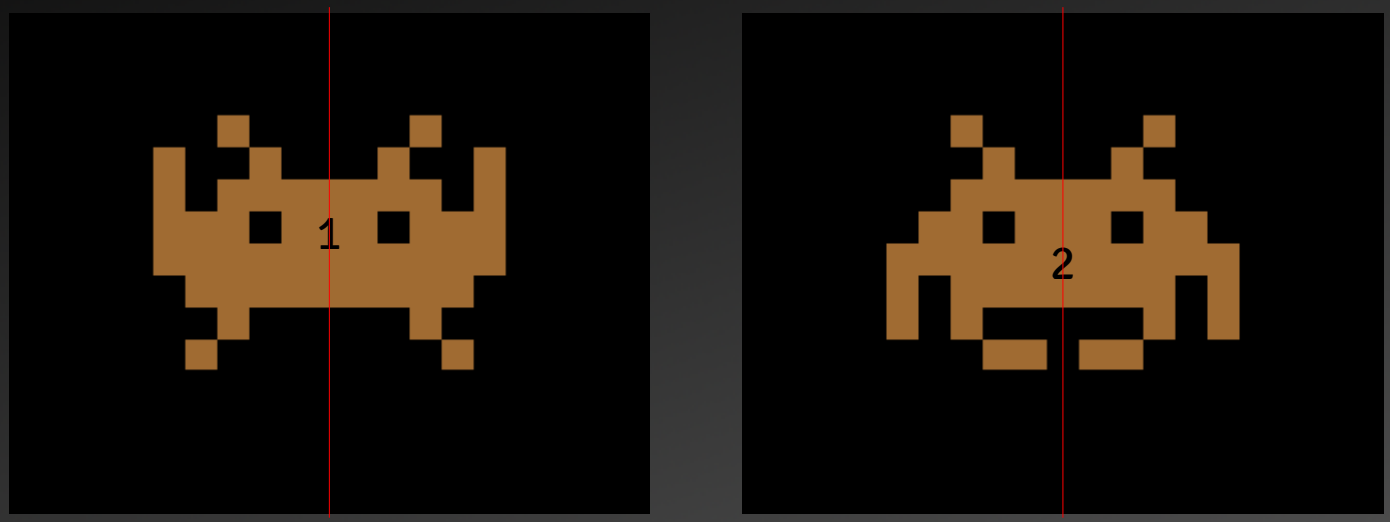
– a reset routine

– some display code (called kernel)

– probably some logic on what to display next

The First Program

Let's take a look at an example:



Ladies and gentlemen start your editors and open "invader.asm" and "framework.asm"