

1 Wersje konsol

Pierwotnym wydaniem konsoli był Family Computer z 1983 roku, w skróconej formie nazywany "Famicom". Famicom oficjalnie nigdy nie został wydany poza Japonią. Nintendo przygotowując się do wydania międzynarodowego swojej konsoli do grania podjęło decyzję o przeprojektowaniu wyglądu sprzętu w celu dostosowania się do zachodnich odbiorców. Wynikiem tych prac był Nintendo Entertainment System, wydany w 1985 roku w USA, rok później na terenie Europy oraz w 1987 roku w pozostałych lokalizacjach w Europie oraz Australii[15].



Rysunek 1: Family Computer wraz z kontrolerami[1]

Rysunek 2: Nintendo Entertainment System NES-001 wraz z kontrolerem[2]

Odświeżone wersje konsol zostały wydane w 1993 roku. Nintendo znacznie zunifikowało wygląd zewnętrznego konsoli, zachowując kompatybilność kartridzy dla poszczególnych regionów.



Rysunek 3: "New Famicom" wraz z kontrolerem[3]

Rysunek 4: Nintendo Entertainment System NES-101 wraz z kontrolerem[4]

Obok oryginalnych sprzętów od Nintendo równocześnie egzystowały konsole podrabiane na masową skalę, w większości były to produkty kompatybilne z japońskim Famicomem. Dokładna historia nie jest znana; zebrane informacje pozwalają na stwierdzenie, iż pierwsze podróbki zaczęły pojawiać się pod koniec lat osiemdziesiątych oraz zostały wyprodukowane na Tajwanie. W latach dziewięćdziesiątych następowała miniutaryzacja sprzętów oraz minimalizacja kosztów, co spowodowało stopniowo powiększane przenoszenie produkcji do Chin oraz drastyczny spadek jakości wykonania konsol. Kreatywność "piratów" doprowadziła do powstania sprzętów będących czymś innym niż zwyczajna konsola:

- Konsole z dwoma gniazdami na kartridże - jedno dla gier z Famicoma, drugie dla gier z NES,
- Konsole z dwoma gniazdami na kartridże - jedno standardowo na zewnątrz obudowy, drugie wewnątrz obudowy z fabrycznie zamontowaną grą. Dzięki temu rozwiązaniu gdy w zewnętrznym gnieździe nie było zamontowanego kartridża to uruchamiał się kartridż wbudowany w konsolę,
- Konsola-klawiatura, udająca komputer - zawsze dołączano kartridż z "programami multimedialnymi" które korzystały z klawiatury. Była tu pewna inspiracja oficjalnym Family Basic,
- Przenośna konsola z wbudowanym wyświetlaczem,
- Sprzęty "Plug 'n' Play" o wymyślnych kształtach, często były one pozbawione gniazda na kartridże, zatem wbudowane gry musiały wystarczyć.



Rysunek 5: Generation NEX - klon z dwoma gniazdami na kartridże, górne jest dla gier Famicom, boczne dla gier NES[5]



Rysunek 6: Daryar DY-400-656 - konsole z wbudowaną grą 400 in 1[6]



Rysunek 7: GLK-2004 - popularna konsola-klawiatura[7]



Rysunek 8: Game Axe Color - jeden z pierwszych handheldów (2000 rok) odwierający gry z Famicoma[8]

Klony konsoli Nintendo są produkowane po dzień dzisiejszy, tyle że częściowo wyszły ze szarej strefy - "Wielkie N" nie ma podstaw prawnych do sądzenia się z firmami gdyż wygasły patenty (TODO: sprawdzić dokładnie jak to wygląda, lecę z głowy z tego co kiedyś czytałem). Konstrukcje są również uwspółczesniane poprzez zastosowanie gniazd HDMI czy obsługę gier z kilku platform jednocześnie.



Rysunek 9: RetroUSB AVS - konsola wysokiej jakości do której stworzenia użyto podzespołów z oryginalnych NES[9]

Rysunek 10: Hyperkin RetroN 5 - sprzęt obsługujący ogromną ilość platform jednocześnie, również kartridże do NES i Famicoma[10]



Rysunek 11: NES 620 Games - nisko-budżetowy, współczesny klon konsoli NES z wbudowanymi grami, bez zewnętrznego gniazda na kartridże[11]

Nieoficjalne sprzęty zyskały popularność w biedniejszych krajach w których Nintendo nie dystrybuowało swoich produktów. "Marki" konsol różni-

ły się między poszczególnymi państwami, niektóre kraje miały nawet "oficjalnych dystrybutorów tychże podróbek. Miały również miejsce sytuacje, gdzie jeden i ten sam model / typ konsoli był sprzedawany pod kilkoma nazwami.[16]



Rysunek 12: Micro Genius IQ-502 - wersja konsoli sygnowana marką producenta[12]



Rysunek 14: Dendy Classic 2 - wersja konsoli Micro Genius IQ-502 przeznaczona na rynek rosyjski, Dendy to marka firmy Steepler.[14]

Rysunek 13: Pegasus IQ-502 - wersja Micro Genius IQ-502 przeznaczona na rynek polski, Pegasus to marka firmy Bobmark.[13]

Wszystkie trzy powyższe sprzęty to w rzeczywistości jeden projekt firmy Micro Genius, różniący się logotypami na konsoli oraz padach.

2 Specyfikacja konsoli

Procesor (CPU): Ricoh 2A03 (NTSC) / Ricoh 2A07 (PAL), ośmiobitowy mikroprocesor będący modyfikacją MOS 6502. Różnice 2A03 / 2A07 względem produktu MOS Technology są następujące[17]:

- Deaktywowany tryb dziesiętny (BCD),
- Wbudowany generator dźwiękowy,
- Obsługa kontrolerów poprzez porty \$4016 oraz \$4017,
- DMA.

Mimo iż jest to 8-bitowy procesor, można zaadresować 64 kilobajty pamięci. To, jak podzielona jest ta przestrzeń adresowa przedstawia poniższa tabela (podane wartości są zapisane w systemie szesnastkowym):

Adresy	Rozmiar	Zastosowanie
\$0000 - \$07FF	\$0800	Wewnętrzna pamięć RAM (dwa kilobajty)
\$0800 - \$1FFF	\$0800	Duble danych spod adresów \$0000 - \$07FF (co dwa kilobajty)
\$2000 - \$2007	\$0008	Rejestry kontrolujące PPU
\$2008 - \$3FFF	\$1FF8	Duble danych spod adresów \$2000 - \$2007 (co osiem bajtów)
\$4000 - \$4017	\$0018	Rejestry kontrolujące APU oraz rejesty wejścia / wyjścia
\$4018 - \$401F	\$0008	Dezaktywowany tryb samotestowania
\$4020 - \$FFFF	\$BFE0	Przestrzeń do dowolnej dyspozycji dla kartridża

Adresy \$4020 - \$FFFF są zaadresowane zgodnie z możliwościami kartridża (patrz Mapper) [18]

Układ graficzny (PPU): Ricoh 2C02, ośmiobitowy układ opracowany przez Nintendo specjalnie dla tej konsoli. Mimo iż jest to 8-bitowy procesor, można zaadresować 16 kilobajtów pamięci. To, jak podzielona jest ta przestrzeń adresowa przedstawia poniższa tabela (podane wartości są zapisane w systemie szesnastkowym):

Adresy	Rozmiar	Zastosowanie
\$0000 - \$0FFF	\$1000	Pattern Table 0
\$1000 - \$1FFF	\$1000	Pattern Table 1
\$2000 - \$23FF	\$0400	Nametable 0
\$2400 - \$27FF	\$0400	Nametable 1
\$2800 - \$2BFF	\$0400	Nametable 2
\$2C00 - \$2FFF	\$0400	Nametable 3
\$3000 - \$3EFF	\$0F00	Dubel danych spod adresów \$2000 - \$2EFF
\$3F00 - \$3F1F	\$0020	Paleta barw
\$3F20 - \$3FFF	\$00E0	Duble palety barw (co 32 bajty)

Dostęp do wyżej wymienionych adresów odbywa się poprzez adresy CPU: \$2006 i \$2007. Gniazdo kartridży: 60-pin (Famicom), 72-pin (NES)

2.1 Rejestry kontrolujące PPU

Adresy \$2000 - \$2007 oraz \$4014 w CPU mają specjalne znaczenie - poprzez nie programista / program może wpływać na działanie układu graficznego konsoli. Zwyczajowe nazwy tych adresów przedstawia poniższa tabela [19]:

Adres hex	Nazwa
\$2000	PPUCTRL
\$2001	PPUMASK
\$2002	PPUSTATUS
\$2003	OAMADDR
\$2004	OAMDATA
\$2005	PPUSCROLL
\$2006	PPUADDR
\$2007	PPUDATA
\$4014	OAMDMA

Nazw tych powszechnie używa się w kodzie programów zamiast adresów bezpośrednich oraz w dyskusjach na temat programowania konsol NES / Famicom. Taką zasadę stosuję również w tym dokumencie.

2.2 Rejestry kontrolujące APU

Adresy \$4000 - \$4013, \$4015 oraz \$4017 w CPU mają specjalne znaczenie - poprzez nie programista / program może wpływać na działanie układu dźwiękowego konsoli. Zwyczajowe nazwy tych adresów przedstawia poniższa tabela [20]:

Adres hex	Nazwa
\$4000	PL1_VOL
\$4001	PL1_SWEEP
\$4002	PL1_LO
\$4003	PL1_HI
\$4004	PL2_VOL
\$4005	PL2_SWEEP
\$4006	PL2_LO
\$4007	PL2_HI
\$4008	TRI_LINEAR
\$400A	TRI_LO
\$400B	TRI_HI
\$400C	NOISE_VOL
\$400E	NOISE_LO
\$400F	NOISE_HI
\$400E	DMC_FREQ
\$400F	DMC_RAW
\$4010	DMC_START
\$4011	DMC_LEN
\$4012	SND_CHN

Nazwy te zostały zaczerpnięte z biblioteki audio do NES / Famicom - FamiTone2 [21]. Używam ich również w kodzie źródłowym programu oraz w tym dokumencie. (TODO: omówienie każdego rejestru, jeszcze nie kodowałem tego to nie znam szczegółów programowych)

3 Projekt

W tej sekcji omówię krok po kroku jak powstawał mój własny projekt programu na platformę NES. Zacznę od pojedynczych, prostych czynności aż po pewnego rodzaju pełnoprawny szkielet takiego projektu zgodny z dobrymi praktykami programowania NES. Następnie zostaną omówione krytyczne fragmenty kodu z punktu widzenia całego projektu oraz własne rozwiązania często spotykanych zagadnień.

3.0 Absolutne minimum

Kod w pełni inicjujący konsolę jest dość rozbudowany, zatem warto zacząć od czegoś łatwiejszego... Niech będzie to coś, co pokaże że jakkolwiek panujemy

nad maszyną - zmiana wartości pojedynczej komórki pamięci to rozsądny pomysł. Stwórzmy plik źródłowy, nazwijmy go *main.s*. Kod ustalający wartość pierwszej komórki RAM na wartość 16 (w systemie szesnastkowym) będzie wyglądał następująco:

```
lda #$16  
sta $0000
```

Mnemonik *lda* ma wiele odmian - ta zastosowana powyżej wczytuje konkretną wartość do wbudowanego w CPU rejestru - akumulatora. Symbol *#\$* oznacza właśnie konkretną wartość liczbową zapisaną szesnastkowo. Mnemonik *sta* kopiuje wartość z akumulatora pod lokację podaną jako operand; tutaj jest to pierwsza komórka pamięci RAM (= o adresie \$0000).

Trzeba zrobić jeszcze jedną rzecz - dopisać za powyższymi poleceniami pętlę nieskończoną:

```
Stop:  
    jmp Stop
```

Trzeba pamiętać o tym iż pod naszym programem nie ma żadnego systemu operacyjnego czy innego środowiska wykonywalnego, zanim nie ma mowy o zakończeniu wykonania programu - bez tej pętli CPU interpretowałby kolejne bajty jako kod, cokolwiek by tam było i wykonał je.

Jednakże jest to o wiele za mało żeby nawet emulator zrobił to, co oczekujemy. Pierwsza sprawa jest związana z ogólnie przyjętym formatem pliku reprezentującym ROM programu na platformę NES. Taki plik ma szesnastobajtowy nagłówek opisujący szczegóły programu. Nagłówek jest potrzebny emulatorom oraz flashcartom do przygotowania adekwatnego środowiska uruchomieniowego. W naszym przypadku nagłówek będzie wyglądał następująco:

```
.byte "NES"  
.byte $1A  
.byte 2  
.byte 0  
.byte %00000000  
.byte %00000000  
.byte 0, 0  
.byte 0, 0, 0, 0, 0, 0
```

Pierwsze trzy bajty ("NES") to magiczna wartość. Kolejny bajt to znak końca linii. Po wyżej opisanych czterech bajtach programy mogą rozpoznać iż mają do czynienia z plikiem w formacie *.NES. Bajt piąty to ilość 16KB

stron pamięci przeznaczonych na kod wykonywalny. Dwa banki wypełniają całą przestrzeń adresową konsoli. Kolejny bajt to ilość 16KB stron pamięci przeznaczonych na grafikę. W tym momencie nie potrzebujemy żadnej grafiki zatem ustawiamy bajt na zero. Właściwie wartość zero ma inny efekt a my skorzystamy z efektu ubocznego tego ale jest to na razie nieistotne - uzyskamy pożądany efekt. Bajty 7 - 10 to flagi - na tę chwilę interesują nas tylko i wyłącznie bajty odpowiadające za Mapper. Mapper 0 jest "podstawowy" przez co odpowiedni do prostych programów, zatem go ustawimy. Warto wspomnieć o tym, jak formowany jest numer mappera: cztery górny bajty pochodzą z czterech górnych bajtów bajta ósmego w nagłówku, cztery dolne bajty pochodzą z czterech górnych bajtów bajta siódmego w nagłówku. (TODO: obrazek lepszy) Funkcjonalność bajtów 9 oraz 10 nie interesuje nas w tej chwili, można je ustawić na 0 i zapomnieć o ich istnieniu. Bajty 11 - 15 to dopełnienie nagłówka. Wartości mogą być dowolne, zwyyczajowo są to zera.

Druga sprawa ma związek z architekturą konsoli. Zawsze po uruchomieniu / zresetowaniu konsoli, pierwszą rzeczą jaką robi procesor to skok bezwzględny do adresu uformowanego z bajtów pod adresami \$FFFC (górny bajt) i \$FFFD (dolny bajt). W tym celu modyfikujemy kod w następujący sposób:

```
_INT_Reset:
    lda #$16
    sta $0000

.word _INT_Reset
```

.word to dyrektywa asemblera działająca jak (.byte), tyle że podajemy wartości dwubajtowe zamiast jednobajtowych. *_INT_Reset* to etykieta, zostanie ona zamieniona na konkretny adres przez linker. Dwukropek definiuje etykietę; samą nazwę etykiety traktuje się jak konkretną wartość szesnastobitową. Nadal zostaje problem powiedzenia kompilatorowi oraz linkerowi że ma wstawić adres wyznaczony przez etykietę *_INT_Reset* dokładnie pod adres \$FFFC w ROM. Inna sprawa to wskazanie kompilatorowi / linkerowi iż nagłówek nie jest częścią kodu lecz ma się znaleźć na początku pliku reprezentującego ROM. Oba problemy rozwiąże dyrektywa asemblera *.segment*. Ukończony pierwszy program będzie wyglądał następująco:

```
.segment "HEADER"
.byte "NES"
.byte $1A
.byte 2
.byte 0
```

```

.byte %00000000
.byte %00000000
.byte 0, 0
.byte 0, 0, 0, 0, 0, 0

.segment "CODE"
_INT_Reset:
    lda #$16
    sta $0000
Stop:
    jmp Stop

.segment "VECTORS"
.word _INT_Reset

```

.segment to dyrektywa mówiąca ”poniższa treść należy do bloku danych zdefiniowanego w pliku konfiguracyjnym linkera podanego jako argument dyrektywy”. Na tę chwilę brzmi to niezrozumiałe, bo jeszcze nawet nie wspominałem czym jest plik konfiguracyjny linkera. Jest to plik ze wskazówkami jak ma być przeprowadzony proces linkowania plików obiektowych. Do pakietu CC65 jest dołączony przykładowy plik dla platformy NES jednakże wymaga wielu zmian by być zgodne z dobrymi praktykami przyjętymi przez społeczność wspólnocie programującą na tą konsolę, zatem nic nam po nim i trzeba samemu rozgryźć jak stworzyć własny. Stwórzmy nowy plik, nazwijmy go *nes.cfg*. W naszym przypadku plik będzie miał następującą treść:

```

MEMORY {
    RAM:      start = $0000,  size = $0800, type = rw, file = "";
    HDR:      start = $0000,  size = $0010, type = ro, file = %0, fill = yes,
    PRG:      start = $8000,  size = $8000, type = ro, file = %0, fill = yes,
}

SEGMENTS {
    BSS:      load = RAM, type = bss;
    HEADER:   load = HDR, type = ro;
    CODE:     load = PRG, type = ro,  start = $8000;
    VECTORS:  load = PRG, type = ro,  start = $FFFC;
}

```

/textit{MEMORY} oraz /textit{SEGMENTS} wyznaczają grupy zasad sterujących linkerem. Etykiety w sekcji /textit{SEGMENTS} opisują dwie rzeczy - mapę pamięci w programie NES za pomocą atrybutu *start* oraz determinuje

kolejność danych w pliku wynikowym dla etykiet z atrybutem `/textittype` o wartości `ro`. Kolejność etykiet mówi jak zostaną ustawione dane w pliku wynikowym wyznaczone przez atrybut `load`. Wartości tam podane odpowiadają etykietom zdefiniowanych w sekcji `MEMORY`. Każda wpis definiuje wielkość obszaru, typ dostępu (odczyt / zapis), w jakim pliku wynikowym się znajdzie obszar, opcjonalne wypełnienie wartościami. Podsumowując: `SEGMENTS` odnosi się do reprezentacji programu na konsoli, `MEMORY` odnosi się do reprezentacji danych w pliku ROM.

Proszę zwrócić uwagę, że nazwy segmentów użyte w pliku źródłowym programu pokrywają się z etykietami w sekcji `SEGMENTS` w pliku konfiguracyjnym. Z tego dla pliku ROM iż:

- na początku pliku pojawi się szesnastobajtowy nagłówek,
- następne szesnaście kilobajtów zajmą dane spod segmentu `CODE`, pozostała wolna przestrzeń zostanie wypełniona zerami
- potem to, co jest pod segmentem `VECTORS` posłuży linkerowi do nadpisania ostatnich 4 bajty segmentu `CODE` ($\$8000 - \$FFFC = \$04$)

A dla konsoli wynika że:

- to co znajduje się za dyrektywą `CODE` w pliku źródłowym zostanie zamapowane na obszar `8000–FFFF` na konsoli NES,
- to co znajduje się za dyrektywą `VECTORS` w pliku źródłowym zostanie zamapowane na obszar `FFFC–FFFF` na konsoli NES, częściowo nadpisując segment `CODE`; jest to jak najbardziej pożądane.

Zanim potwierdzimy powyższe słowa, potrzebujemy stworzyć wynikowy plik ROM. Umieszczamy pliki ca65.exe oraz ld65.exe w tym folderze co nasz kod źródłowy oraz plik konfiguracyjny. Najpierw komplikacja:

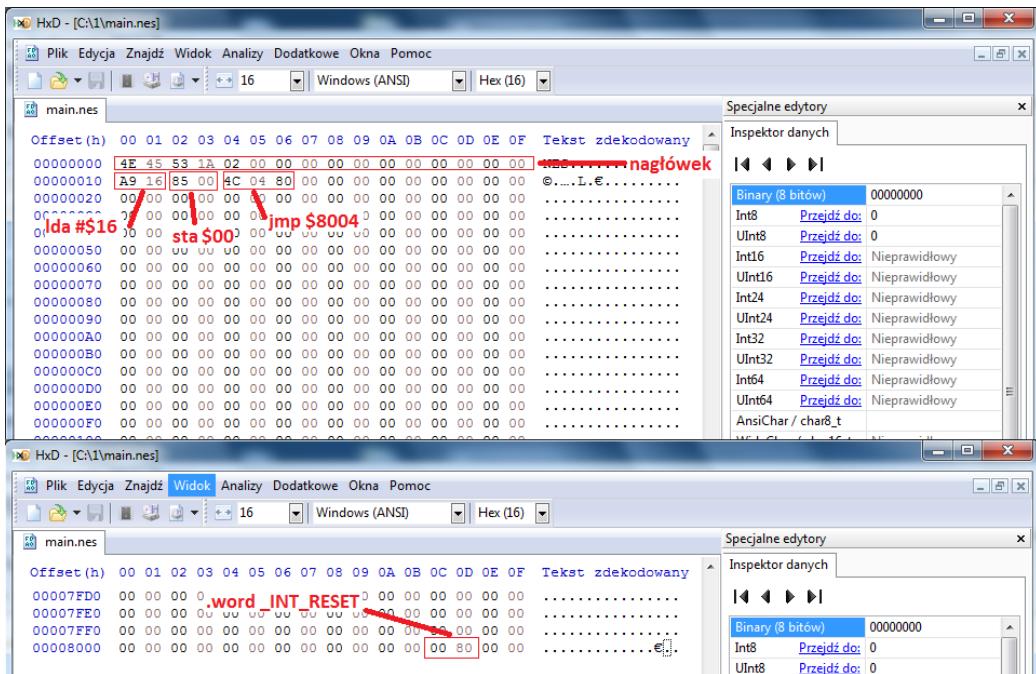
```
ca65 -o main.o main.s
```

Argument `-o` pozwala ustalić wynikową nazwę pliku obiektowego. Na końcu podajemy listę plików źródłowych. Następnie czas na linkowanie:

```
ld65 -t nes -o main.nes main.o
```

Argument `-t` ustala nazwę pliku, pod którą linker znajdzie konfigurację. Co ważne, plik musi mieć rozszerzenie `*.cfg` którego nie podajemy w powyższym poleceniu. Argument `-o` ustala nazwę wynikową pliku ROM. Na końcu jest lista plików obiektowych.

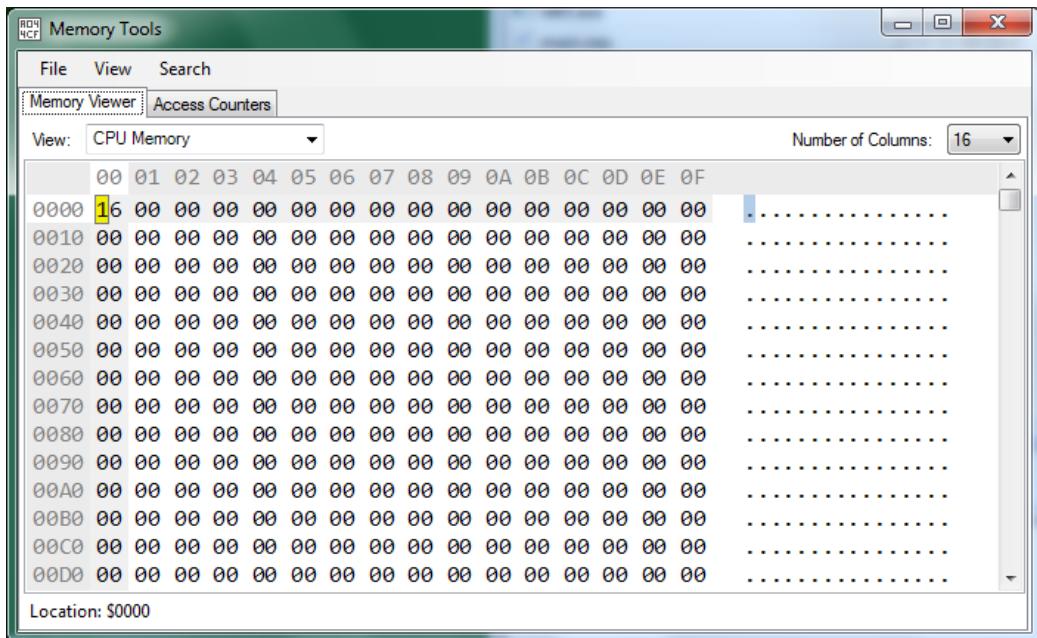
Zobaczmy jak te dwa narzędzia stworzyły plik ROM. Do tego trzeba posłużyć się oprogramowaniem zwanym hexedytorem:



Rysunek 15: Podgląd pliku ROM w programie HxD.

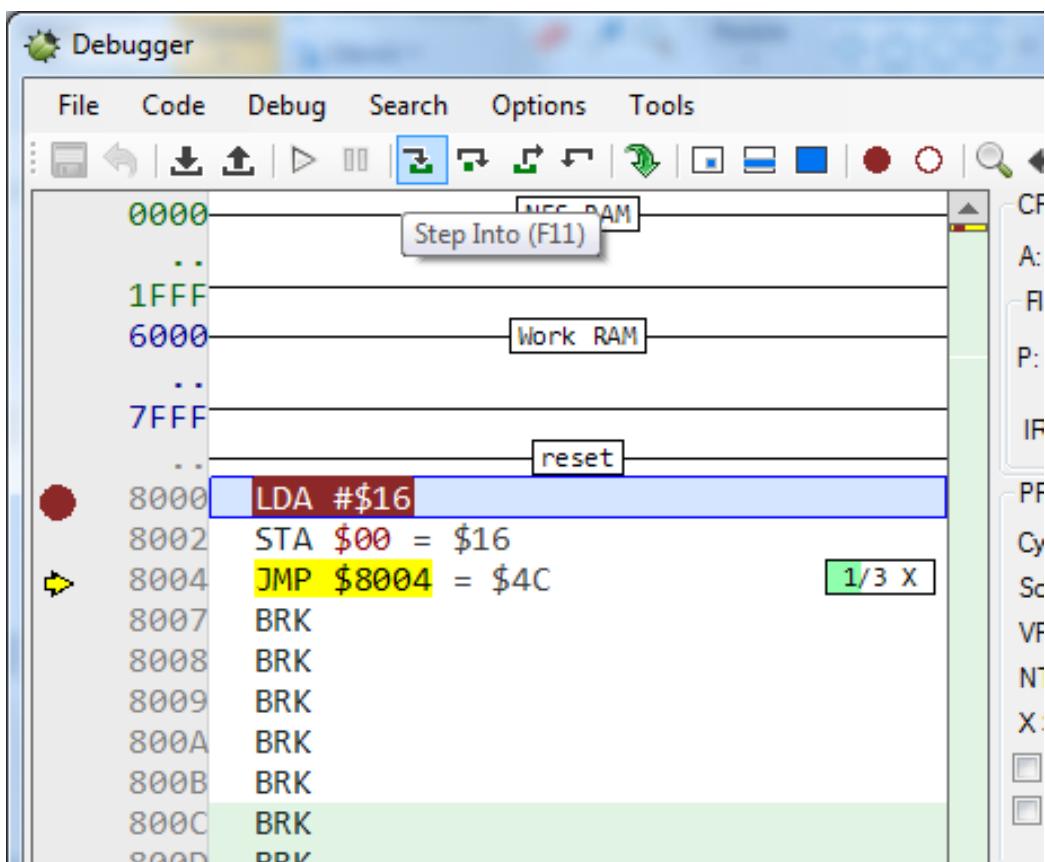
Kompilator inaczej zakodował instrukcję *sta* - związane jest to z tak zwanym Zero Page i nie ma wpływu na poprawność programu. Kiedy przyjrzymy się przesunięciom poszczególnych elementów względem początku pliku zobaczymy, iż wszystko jest ustawione tak jak rozkazaliśmy linkerowi.

Teraz uruchommy ROM w Mesen. Przywita nas jednokolorowy ekran. Aby sprawdzić czy komórka pamięci zawiera pożądaną wartość, musimy wybrać z menu *Debug* opcję *Memory Tools* i upewnić się, że w dropdownie *View* mamy wybraną opcję *CPU Memory*:



Rysunek 16: Podgląd RAM w emulatorze Mesen.

Faktycznie, jest tam wartość szesnaście heksadecymalnie, zatem wszystko działa jak powinno. Jest to dobry przykład na naukę obsługi debuggera. Zatem w oknie głównym Mesena wybieramy z menu *Debug* opcję *Debugger*. Okno debuggera może przytłoczyć ogromem informacji, na początek skupimy się na podglądzie kodu. Postawmy breakpoint na adresie \$8000 (początek kodu) klikając LPM na lewo od numeru adresu. Następnie wyciągamy na wierzch okno z podglądem RAM aby widzieć naszą zmienianą komórkę pamięci a następnie z menu debuggera wybieramy *Debug* opcję *Reset*. Debugger sfokusuje się na adresie \$8000, co oznaczone jest żółtym tłem oraz żółtą strzałką wskazującą kolejną instrukcję do wykonania. Jednakże jest pewien problem - w pamięci już jest ustawiona wartość \$16. Otóż NES nie czyści pamięci podczas resetu - program sam musi o to zadbać.



Rysunek 17: Debugger w emulatorze Mesen.

Mimo tego prześledźmy wykonanie krok po kroku. W tym celu kilka razy kliknijmy przycisk *Step Into*. Otóż po wykonaniu instrukcji *sta* komórka w podglądzie RAM podświetliła się na czerwono - tak właśnie Mesen pokazuje, iż na danej komórce pamięci odbył się zapis. Po dotarciu do instrukcji *jmp* program "stoi" na instrukcji tak jak chcieliśmy pisząc pętlę nieskończoną.

Wprowadźmy poprawkę ustawiającą wartość na zero tuż po resecie, segment *CODE* będzie wyglądał następująco:

```

_INT_Reset:
    lda #$00
    sta $0000
    lda #$16
    sta $0000
Stop:
    jmp Stop

```

Skompilujmy a następnie podejrzyjmy w debuggerze jak ta wersja kodu

się wykonuje. Jeżeli nie wyłączyliśmy emulatora to breakpoint nadal będzie postawiony w pożdanym miejscu. Potrzebujemy przejść programu dwukrotnie - pierwsze przejście ustali wartość w RAM na \$16, a w drugim zobaczymy dwukrotne ustawianie wartości - najpierw na zero, potem ponownie na \$16. Warto również popatrzeć na sekcję *CPU Status* na prawo od naszego kodu w debuggerze a dokładniej pole *A* - to jest właśnie rejestr procesora, akumulator.

Za nami bardzo długie wprowadzenie a właściwie nic wielkiego nie zaprogramowaliśmy. Są to solidne fundamenty do dalszego kodowania, bez nich przedniej czy później wychodzą braki tej podstawowej wiedzy, niezrozumienie bardziej skomplikowanych zagadnień. Narzędzia deweloperskie emulatora to również potężna rzecz którą warto wykorzystywać przy każdej napotkanej wątpliwości / problemie, pozwalając zatrzymać "czarną skrzynkę" jaką niejako jest NES dając podgląd na procesy zachodzące w układach scalonych nawet w mikroskali jaką jest pojedyncza instrukcja procesora.

3.1 Właściwe "Hello World"

NES to konsola mająca wiele pułapek czujących się na programistę któremu brakuje wiedzy technicznej na temat tego sprzętu. Poniżej omówię program który w pełni poprawnie zainicjuje konsolę do działania. Jest on wspólnym opracowaniem wielu ludzi współcześnie zajmujących się tą tematyką [27]. Jest tam trochę działań które można by pominąć gdyż prawidłowe wartości mogą już tam być ustawione, jednakże nie musi być to reguła oraz pozostawiło by to konsolę w stanie niezdefiniowanym narażając na trudne w zdebuggowaniu błędy. Przedstawię ten kod go w kilku fragmentach. Nadal kod będzie dopisywany do pliku main.s, należy to robić na samym początku segmentu *CODE*. Oto pierwszy z nich:

```
reset:  
    sei  
    cld  
    ldx #$40  
    stx $4017  
    ldx #$FF  
    txs  
    inx  
    stx $2000  
    stx $2001  
    stx $4010
```

Opakod *sei* to instrukcja wyłączająca odnotowywanie przez CPU przerwań

niemaskowalnych. To znaczy że procesor nie będzie reagował na przerwania, dzięki czemu nie będzie zakłócał wykonania naszego kodu inicjującego. Przerwania nadal nadal się "wewnętrznie", cyklicznie występują! Instrukcja *cld* wyłącza tryb dziesiętny. Oczywiście wariant procesora wbudowany w NES ma wycięty tryb dziesiętny, jednakże debuggery ogólnie działające na kodzie napisanym pod MOS 6502 opierają się na tym, iż ta konkretna flaga jest ustawiona. Następnie pojawiają się *stx* oraz *ldx* - odpowiedniki instrukcji *sta* oraz *lda* tyle że dla rejestru X. Jest jeszcze analogiczna parą mnemoników dla rejestru Y (*sty*, *ldy*). Wartość #\$40 to właściwie ustawienie bitu numer sześć. Ta wartość jest wysyłana pod port \$4017 CPU w celu wyłączenia przerwań generowanych przez dodatkowe podzespoły rozszerzające możliwości kartridża. Następnie odbywa się inicjacja stosu - wartość \$FF jest ładowana do rejestru X, mnemonik *txs* kopiuje tę wartość z rejestru X do rejestru wskaźnika stosu. Proszę zauważać że to dolna część wskaźnika stosu - góra zostanie ustawiona później. W rejestrze X nadal jest wartość \$FF. Instrukcja *inx* inkrementuje X o jeden, zatem rejestr się "przekręca", ustawiając się na wartość 0. Trzy kolejne zapisy pod adresy \$2000, \$2001 oraz \$4010 to wyłączenie wszelkich flag którymi te trzy rejestrze zarządzają. Najważniejsze z nich to dezaktywacja NMI - znaczy to tyle że przestanie wykonywać się to przerwanie (rejestr \$2000), wyłączenie renderowania ekranu (\$2001), wyłączenie przerwania DMC którego dedykowanym zastosowaniem jest prawidłowe odgrywanie digitalizowanych próbek dźwiękowych (\$4010).

Powyższe instrukcje są powiązane z kolejnym krokiem - trzeba odczekać dwie klatki obrazu aby układ graficzny ustabilizował się oraz zaczął pracować w pełni prawidłowo PPU (TODO: dokładne wskazanie omówienia co tam się dzieje). Mamy wyłączone NMI ale jest jeszcze jedna metoda na odliczenie klatek obrazu. PPU zawsze sygnalizuje moment w którym zakończył rysowanie klatki obrazu (rozpoczęcie wygaszanie pionowe obrazu) poprzez ustawienie najwyższego bitu w rejestrze flag \$2002:

```
vblankwit1:
    bit $2002
    bpl vblankwait1
```

Opis kodu *bit* ma specyficzne działanie: wykonuje operację bitową "akumulator AND operand instrukcji *bit*", następnie "zapomina" wynik za to ustawia flagę Z (flaga zero) jeśli wynik operacji AND był równy zero. Oprócz tego zawsze kopiuje bit szósty operandu do flagi V (flaga przepelnienia), a siódmy do flagi N (flaga wartości ujemnej). W akumulatorze mamy w tej chwili wartość zero - w jego przypadku akurat mamy gwarancję że po uruchomieniu konsoli / resecie zawsze będzie to wartość zero a wcześniej w kodzie nigdzie nie manipulowaliśmy tym rejestrzem. Przykład powinien rozjaśnić:

W rejestrze A mamy wartość zero.

Chcemy wykonać operację \textit{bit} na wartości #\$80 (%10000000 binarnie). Zatem wynik operacji:

```
bit    #$80
będzie następujący:
%00000000
AND %10000000
%00000000
a w rezultacie:
\begin{enumerate}
\item flaga Z zostanie ustawiona,
\item flaga V zostanie wyzerowana,
\item flaga N zostanie ustawiona.
\end{enumerate}
```

Powyższe omówienie instrukcji *bit* to sytuacja tożsama z tą, gdy właśnie PPU dało znak że skończyło rysować klatkę. Instrukcja *tpl* to jeden ze skoków warunkowych - dla *tpl* skok do etykiety podanej jako operand ma miejsce, gdy flaga wartości ujemnej jest wyzerowana. Zatem program będzie "kręcił się w miejscu" dopóki flaga jest ujemna. Przestanie to robić gdy opkod *bit* ustawi flagę N, a to się stanie gdy PPU skończy rysować klatkę oraz ustawi najwyższy bit w rejestrze \$2002. W ten sposób należy rozumieć zaprezentowany kod. Działa to poprawnie także dlatego, iż po odczycie rejestru \$2002 najwyższy bit zostanie wyczyszczony, zatem nie ma mowy o sytuacji gdzie PPU ustawiło tę flagę jakimś cudem wcześniej a teraz jest w trakcie rysowania kolejnej klatki.

Mamy jeszcze całą klatkę obrazu do wykorzystania, zatem zróbmy coś pożytecznego: czyszczenie RAM:

```
clear_memory:
    lda #$00
    sta $0000, x
    sta $0100, x
    sta $0200, x
    sta $0300, x
    sta $0400, x
    sta $0500, x
    sta $0600, x
    sta $0700, x
    inx
    bne clear_memory
```

Na początek warto odnotować że rejestr X zawiera w tej chwili nadal wartość zero. Instrukcja *sta* ma kilka wariantów, powyższy korzysta z jednego z trybów procesora zwanego "Absolute X". Działa to tak, że w celu wyznaczenia adresu pod którym zmodyfikuje się wartość do konkretnego adresu w pierwszym operandzie *sta* dodawana jest wartość rejestrów X, będącego zatem drugim operandem. Na końcu pętli jest inkrementacja tegoż rejestrów oraz instrukcja *bne* - kolejny skok warunkowy do etykiety podanej jako operand mający miejsce, gdy flaga Z ma wartość zero. Zauważmy że pierwsze sprawdzenie rejestrów X odbywa się gdy ma on wartość jeden, zatem flaga Z nie będzie ustawniona dzięki czemu będzie miał skok do podanej etykiety. Tak będzie aż do momentu, gdy X się "przekręci" i wskaże wartość zero. W tym momencie wszystkie komórki RAM (zakres \$0000 - \$07FF) będą zawierać wartości zero. Sam proces zerowania odbywa się tak, że w pierwszym kroku wyzerowane będą komórki \$0000, \$0100, \$0200, ... , \$0700. W drugim: \$0001, \$0101, \$0201, ... , \$0701. W ostatnim, dwudziestu pięćdziesiątym szóstym kroku zerowanie odbędzie się pod adresami \$00FF, \$01FF, \$02FF, ... , \$07FF.

Teraz czas na czekania na koniec drugiej klatki obrazu:

```
vblankwait2:  
    bit $2002  
    bpl vblankwait2
```

Działanie analogiczne do oczekiwania na pierwsze rozpoczęcie wygaszania obrazu.

W tym momencie PPU jest już gotowe do pracy. Wykorzystajmy to wyświetlając jednokolorowy obraz we fioletowym kolorze. Zrobimy to tak:

```
lda #%01100000  
sta $2001
```

To co właściwe ładujemy do rejestrów \$2001 to flagi ustawiające "intensywne odcień" kolorów czerwonego oraz zielonego (TODO: referencja do opisu rejestrów \$2001). Jest to graficzna funkcjonalność PPU, polegającą na tym iż wszystkie kolory w palecie kolorów (TODO: link) zmieniają odcień na bardziej czerwony, zielony, niebieski bądź dowolny miks nich. Nie będziemy wchodzić w ten temat, gdyż jest to niemal nieużywana funkcjonalność a jej praktyczne zastosowanie jest minimalne. Po prostu przyda nam się ona do szybkiego testu czy układ graficzny działa poprawnie. O palecie barw będzie w dalszej części dokumentu.

Zanim skompilujemy program to możemy już usunąć kawałek kodu który napisaliśmy w poprzedniej podsekcji - nie ma potrzeby oddzielnego ustawiania komórki RAM o adresie \$0000 na zero. Finalny kod wygląda następująco:

```

.segment "HEADER"
.byte "NES"
.byte $1A
.byte 2
.byte 0
.byte %00000000
.byte %00000000
.byte 0, 0
.byte 0, 0, 0, 0, 0, 0

.segment "CODE"
_INT_Reset:
    sei
    cld
    ldx    #$40
    stx    $4017
    ldx    #$FF
    txs
    inx
    stx    $2000
    stx    $2001
    stx    $4010

vblankwait1:
    bit    $2002
    bpl    vblankwait1

clear_memory:
    lda    #$00
    sta    $0000, x
    sta    $0100, x
    sta    $0200, x
    sta    $0300, x
    sta    $0400, x
    sta    $0500, x
    sta    $0600, x
    sta    $0700, x
    inx
    bne    clear_memory

vblankwait2:

```

```
bit $2002
bpl vblankwait2

lda #%
```

01100000

```
sta $2001

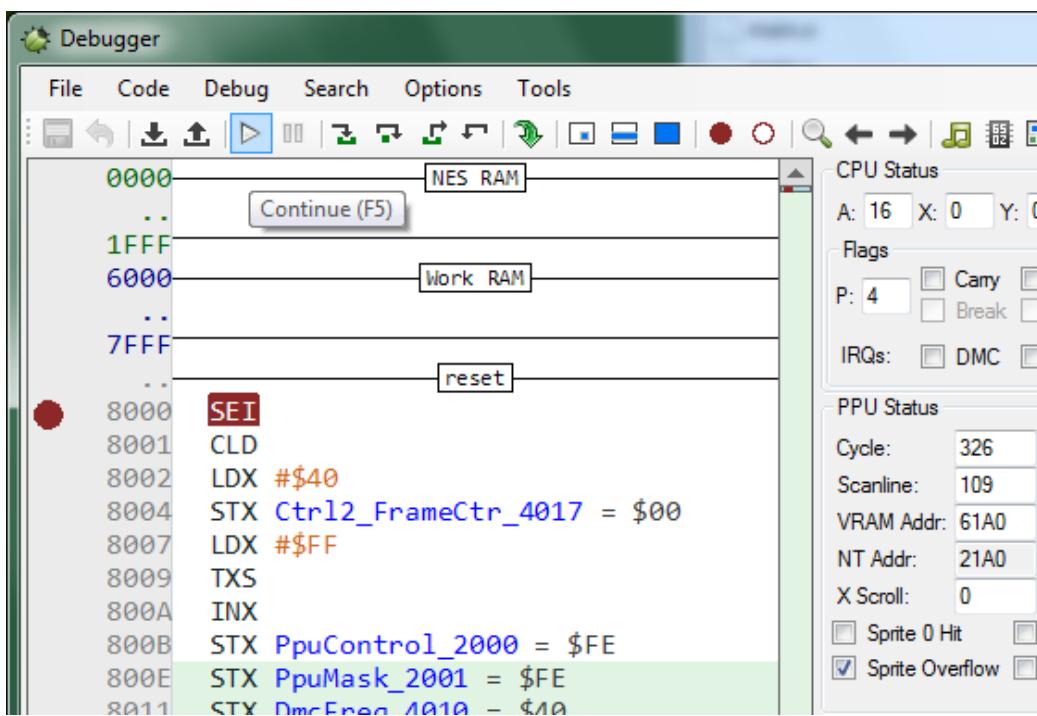
lda #$16
sta $0000
Stop:
jmp Stop

.segment "VECTORS"
.word _INT_Reset
```

Nie ma potrzeby zmiany pliku konfiguracyjnego linkera. Budowę pliku ROM przeprowadzamy tymi samymi komendami co poprzednio:

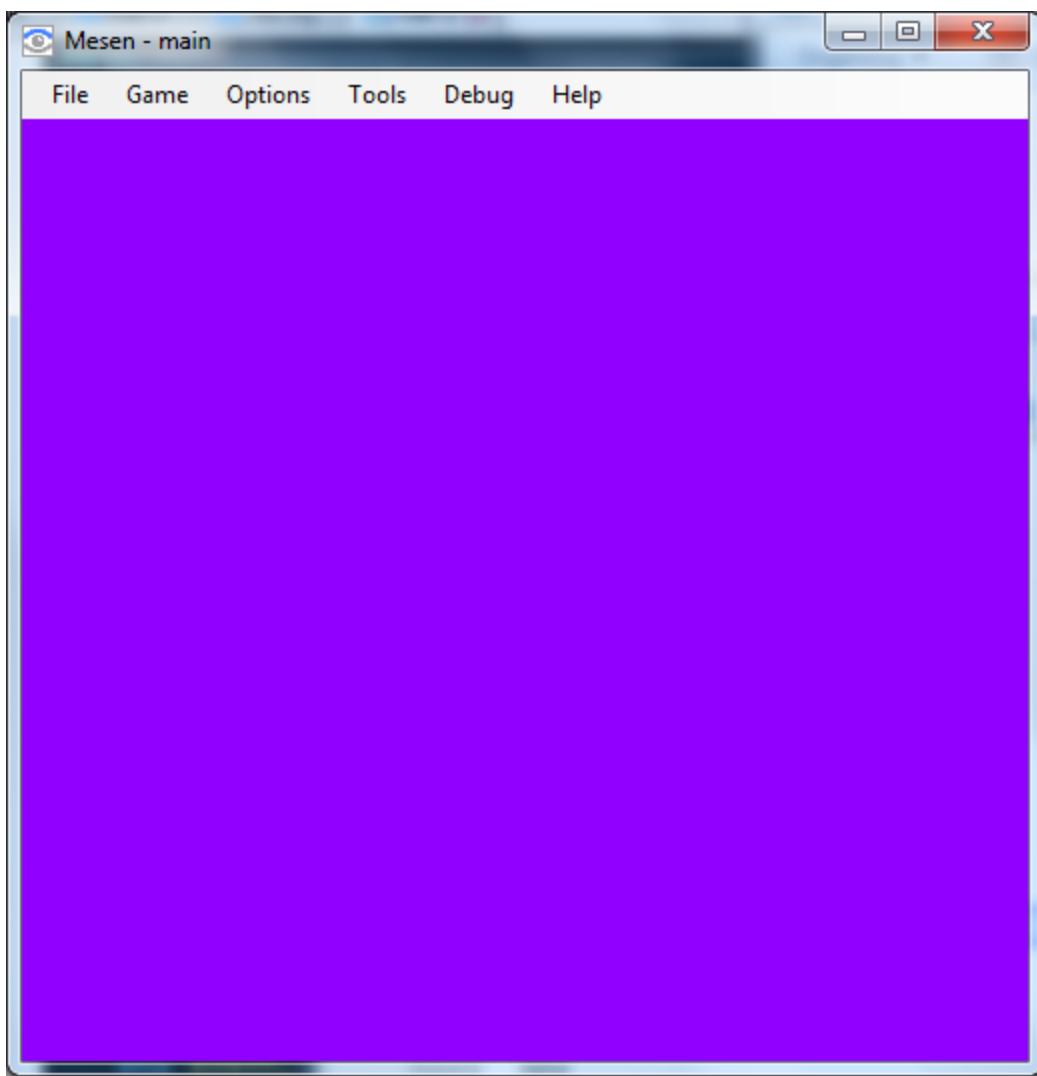
```
ca65 -o main.o main.s
ld65 -t nes -o main.nes main.o
```

Jeśli nie wyłączyliśmy poprzednio breakpointu w debuggerze to po otwarciu świeżo skompilowanego ROM w emulatorze wykonanie programu będzie wstrzymane przed pierwszą instrukcją. W tym celu trzeba wejść w okno debuggera i nacisnąć przycisk *Continue*.



Rysunek 18: Przycisk *Continue* w debuggerze emulatora Mesen.

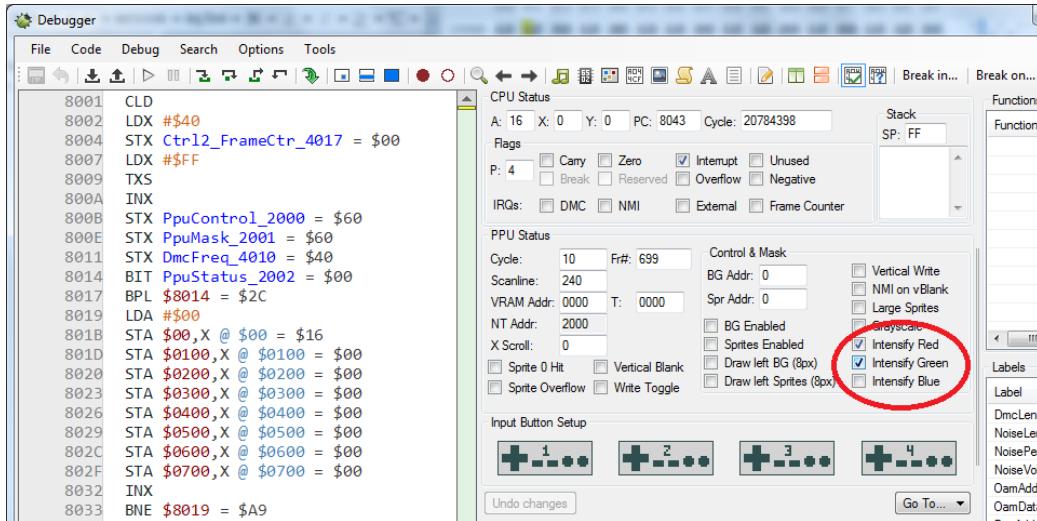
Po udanym procesie budowania naszym oczom powinien ukazać się jednokolorowy, fioletowy ekran:



Rysunek 19: Wynik dotychczas napisanego kodu.

Warto poświęcić chwilę podglądowi w debuggerze tego, jak wykonuje się kod w celu lepszego zrozumienia. Warte odnotowania jest, iż debugger używa specjalnych etykiet dla niektórych rejestrów - są to zwyczajowe nazwy tychże. Po najechaniu na taką etykietę ujrzymy szczegółowy opis danego rejestru. Po prawej stronie okna jest lista wszystkich takich rejestrów w okienku *Labels*, właśnie tam można również edytować treść pokazującą się w dymku. Będzie o tym mowa jeszcze później. Druga sprawa to w sekcji *PPU Status* zaznaczone są checkboxy *Intensify Red* oraz *Intensify Green* - to właśnie bity które ustawiliśmy w rejestrze \$2001. Możemy odznaczyć / zaznaczyć te pola gdy zatrzymamy wykonanie programu. Zmiany będą widoczne po ponownym

uruchomieniu wykonania kodu, polecam poeksperimentować.



Rysunek 20: Podgląd stanu flag ”intensywnych odcieni” w debuggerze.

3.2 Paleta kolorów

Układ graficzny NES ma grupę zdefiniowanych kolorów jakie można uzyskać. Ta grupa liczy sześćdziesiąt cztery różne barwy:

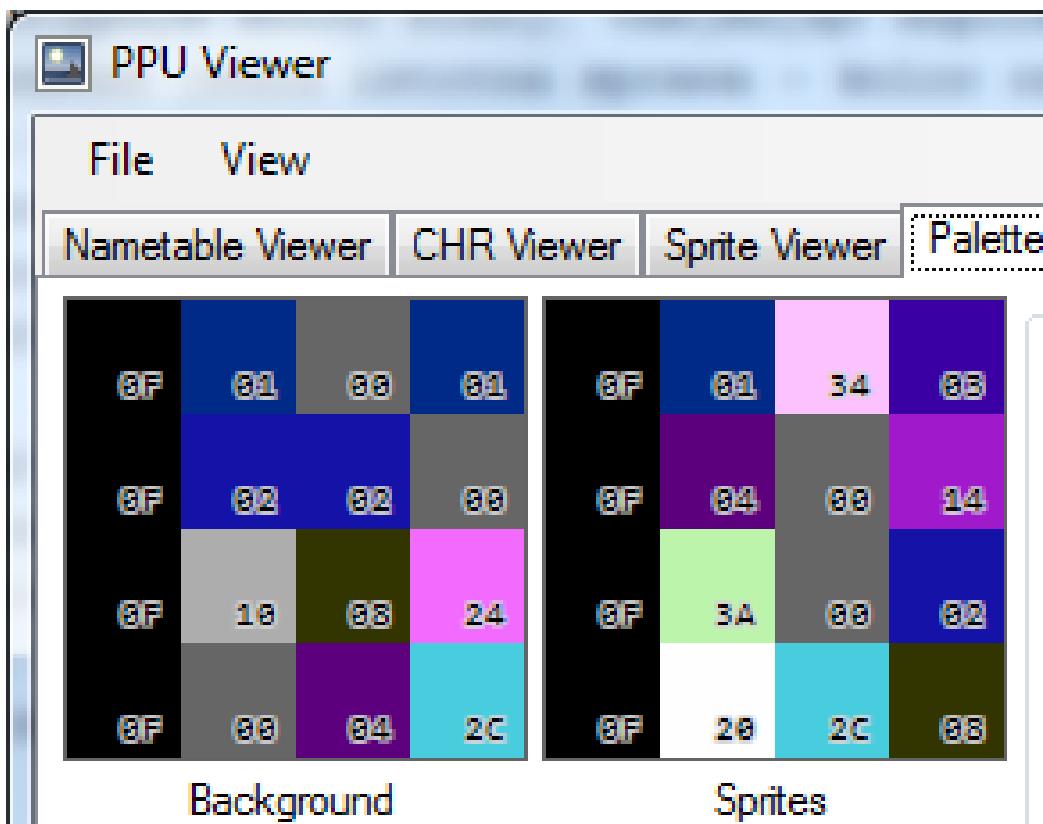


Rysunek 21: Wszystkie możliwe kolory do uzyskania na konsoli NES.

Ułożenie nie jest przypadkowe - na krańcach mamy odcienie szarości, w środku kolory są ułożone w pionie według jasności, w poziomie na podstawie koła barw (TODO: na pewno?). Łatwo zauważać że ilość różnych kolorów jest mniejsza: jest dziesięć identycznych czerni oraz dwa razy występuje kolor biały. Odejmując duplikaty dostajemy 54 unikalne barwy. Jest jeszcze jedna istotna sprawa - kolor oznaczony jako \$0D sprawia problemy niektórym

odbiornikom telewizyjnym które interpretują go jako sygnał synchronizacyjny co powoduje problemy z poprawnym wyświetlanie obrazu, toteż zaleca się go nie stosować. Z tego powodu ten kolor również nazywa się potocznie "ciemniejszym niż czerń".

To też nie jest tak, że możemy używać wszystkich kolorów w danej chwili. PPU ma dedykowany obszar pamięci na ograniczoną ilość kolorów. Podzielony jest on kolory tła (background / BGR) oraz kolory sprite'ów (sprites / SPR). Obie grupy mają po cztery palety. Te kolejne mają cztery kolory. Jednak to nie koniec. Tutaj napotykamy na kolejne ograniczenie - jeden z kolorów jest wspólny dla wszystkich palet a zarazem pełni rolę koloru przezroczystego dla sprite'ów. Przykład:



Rysunek 22: Podgląd obecnie ustawionej palety barw w emulatorze Mesen.

Po lewej stronie są kolory tła, po prawej sprite'ów. Każda paleta zajmuje jeden rzad. Pierwszy kolor w każdej palecie to kolor wspólny, tutaj jest to czerń (oznaczenie \$0F). Palety przyjęło się numerować od zera. Zatem paleta numer 0 dla tła zawiera cztery barwy - czarny (wspólny, \$0F), niebieski (\$01), szary (\$00), niebieski (\$01). Barwy jak najbardziej mogą się

powtarzać w ramach palety. Paleta numer 1 dla tła zawiera kolory: czarny (wspólny, \$0F), granatowy (\$02), granatowy (\$02), szary (\$00). Barwy mogą powtarzać się również między różnymi paletami. Niżej są palety numer 2 oraz 3, skonstruowane analogicznie. Czarny będzie kolorem domyślnie wypełniającym ekran jeśli nie będzie w tym miejscu innej barwy. Palety sprite'ów też mają cztery komórki na kolory. Przeanalizujmy paletę numer zero: czarny (wspólny, \$0F), niebieski (\$01), jasnoróżowy (\$34), fioletowy (\$03). Kolejne palety skonstruowane są analogicznie. W tym przypadku czarny będzie kolorem przezroczystym - piksel z tym kolorem będzie zastąpiony kolorem tła. Zatem właściwie każda tego typu paleta ma tylko trzy unikalne kolory.

Podsumowując - w danej chwili program może mieć do dyspozycji 25 unikalnych kolorów: cztery kolory z palety BGR 0, plus po trzy kolory z palet BGR 1 - 3 (powtarza się jeden kolor występujący w palecie zero) łącznie dziewięć, plus po trzy kolory z palet SPR 0 - 3.

Skorzystajmy z przyswojonej wiedzy i ustalmy paletę kolorów w naszym programie. Przyda się ona do dalszych zagadnień. Najpierw opracujemy tablice z wartościami pożądanych kolorów:

```
palette:
.byte $0f, $01, $21, $31 ; BGR 0
.byte $0f, $02, $22, $32 ; BGR 1
.byte $0f, $03, $23, $33 ; BGR 2
.byte $0f, $04, $24, $34 ; BGR 3
.byte $0f, $05, $25, $35 ; SPR 0
.byte $0f, $06, $26, $36 ; SPR 1
.byte $0f, $07, $27, $37 ; SPR 2
.byte $0f, $08, $28, $38 ; SPR 3
```

Umieszczamy ją za pętlą nieskończoną *Stop*. Symbol średnika oznacza komentarz - to co napiszemy po średniku nie jest traktowane jak kod. Tutaj użyliśmy komentara do opisu które dane odpowiadają której palecie barw. To co jest po etykiecie *palette* traktujemy jako jeden ciągły obszar. Podział na wiersze został zastosowany dla czytelności. Zatem mamy do wczytania 32 wartości. Jak to zrobić? Musimy napisać odpowiednią pętlę, jednakże najpierw wpiszmy te wartości do pamięci RAM pod adresy \$0010 - \$002F - podmienimy to na wysyłanie danych do PPU za chwilę. Poniższy kod umieszczamy po pętli *vblankwait2* ale przed konfiguracją rejestru \$2001:

```
idx #$00
_loop_Palette:
lda palette, x
sta $0010, x
```

```

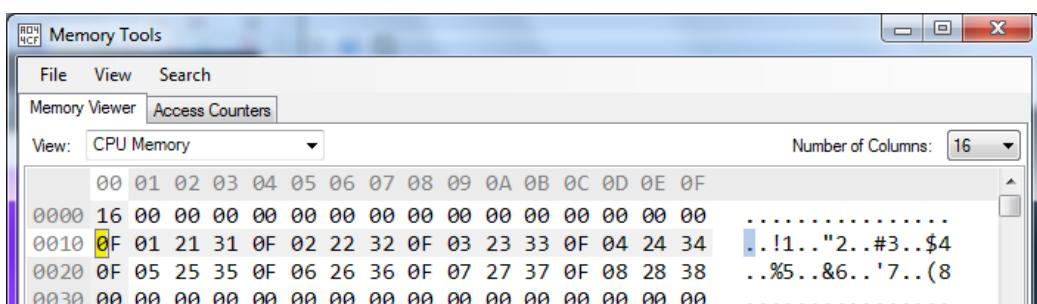
inx
cpx #$20
bne _loop_Palette

```

Przed rozpoczęciem pętli w rejestrze X umieszczamy początkową wartość licznika. Następnie mamy pętle która zajmuje się przetwarzaniem danych. Mnemonik *lda* również ma tryb "absolute X" który już widzieliśmy w wykonaniu instrukcji *sta* przy czyszczeniu RAM. Zatem kod *lda palette*, *x* załadowuje bajt danych spod adresu (etykieta (*palette*) + wartość w rejestrze X). Następnie wrzucamy tę wartość do RAM, inkrementujemy X. Kolejna instrukcja, *cpx* porównuje wartość znajdująca się w rejestrze X z wartością podaną jako operand. Istnieją również analogiczne operacje do porównywania dla pozostałych rejestrów *cmp* dla rejestrów A, *cpy* dla rejestrów Y. Rezultatem porównania jest ustalenie odpowiednich flag procesora. Tutaj porównanie jest z wartością `#$20` 32 dziesiętnie czyli wielkością naszych danych, czyli po tylu bajtach chcemy zakończyć przetwarzanie. Żeby to zapewnić po porównaniu stawiamy skok warunkowy, tutaj jest to *bne*. Dlaczego? Właściwie warunek końca pętli można ująć jako *rejestr X == #\$20* i tylko w tym jednym przypadku zostanie ustalona flaga procesora Z; w innych przypadkach będzie ona wyzerowana, a co za tym idzie *bne* będzie skakał na początek pętli zawsze z wyjątkiem sytuacji w której właśnie chcemy zakończyć pętli - wtedy *bne* "przepuści" wykonywanie programu dalej.

Powyższy kod to pewien wzorzec na przetwarzanie zbiorów danych, które ma często miejsce w kodzie.

Czas na komplikacje oraz uruchomienie. Po załadowaniu ROMu do emulatora i wejściu do podglądu pamięci powinniśmy ujrzeć następujący widok:



Rysunek 23: Podgląd RAM w emulatorze Mesen z przetworzonymi danymi.

Udało się załadować poprawnie przetworzyć dane, zatem czas na przesłanie ich do PPU.

Przypominam iż program komunikuje się bezpośrednio wyłącznie z CPU, zatem nie możemy użyć instrukcji *sta* i tylko zmienić adresu docelowego.

CPU ma dedykowane adresy do komunikacji z PPU i ich właśnie użyjemy. Najpierw przygotowujemy PPU powiadamiając poprzez CPU o adresie do którego chcemy uzyskać dostęp, umieszczając poniższy kod po pętli *vblankwait2* a przed pętlą przetwarzającą dane:

```
lda $2002
lda #$3f
sta $2006
lda #$00
sta $2006
```

Odczyt rejestru \$2002 służy do zresetowania adresu wewnętrznego PPU. Układ graficzny cały czas wykonuje jakieś operacje używając właśnie tego wewnętrznego adresu a my musimy mu "przeszkodzić" przerywając domyślną pracę. Następnie ładowamy górną część adresu początku palety tła (#\$3f) i wysyłamy do adresu \$2006 CPU. Ten z kolei przekazuje go PPU którego zadaniem jest ustawić tę wartość w swoim wewnętrznym adresie. Kolejna para *lda* / *sta* przesyła dolny bajt adresu początku palety tła. Adres \$3F00 to pierwsza wartość palety BGR 0.

Kolejna czynność to faktyczne przesyłanie danych. Wymaga to małej zmiany w pętli, podmienienia instrukcji *sta* na następującą:

```
sta $2007
```

Instrukcja ta robi dwie rzeczy: poprzez pośrednika - CPU wysyła zawartość rejestru A pod wcześniej ustalony adres w PPU, a samo PPU zwiększa wewnętrzny adres o jeden.

Zatem cała pętla będzie wyglądała tak:

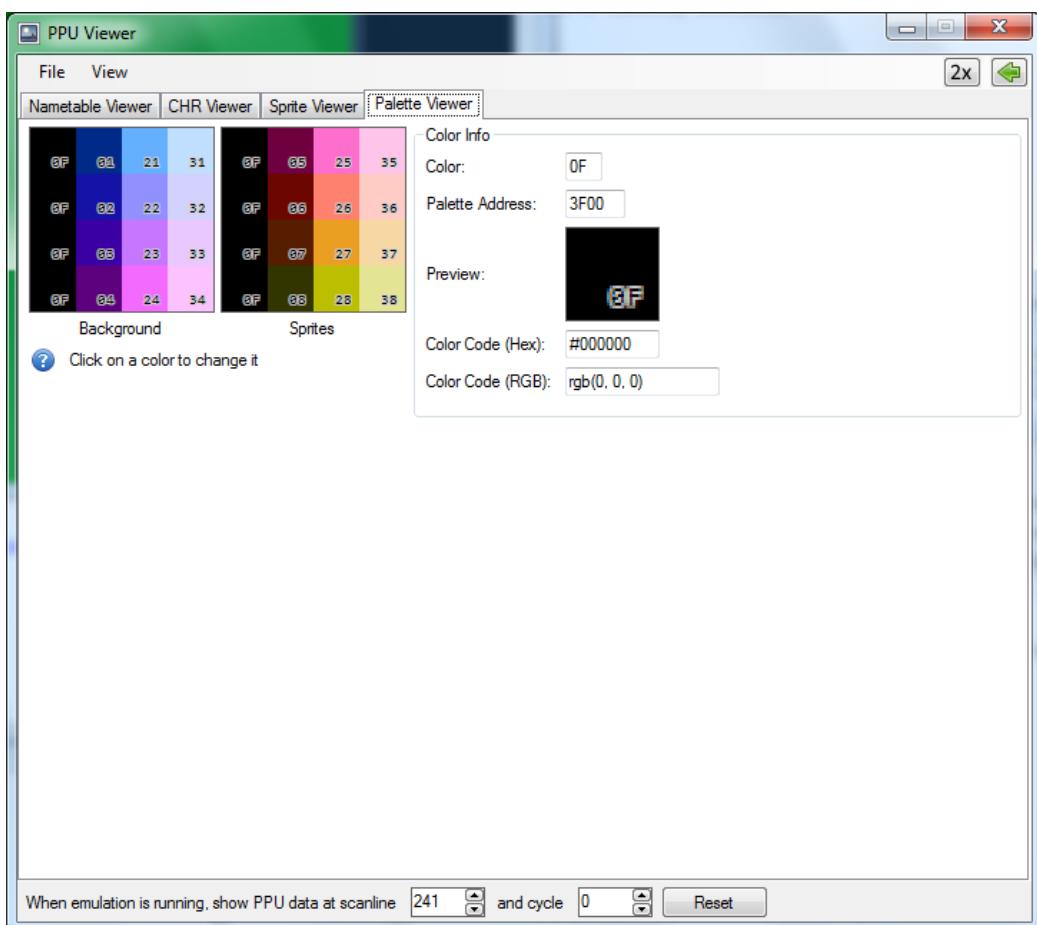
```
_loop_Palette:
lda palette, x
sta $2007
inx
cpx #$20
bne _loop_Palette
```

Rozwijając wątek ta pętla prześle zawartość tablicy *palette* do PPU bajt po bajcie. W pierwszym kroku pierwsza wartość z tablicy wyląduje pod adresem \$3F00, w drugim druga wartość znajdzie się pod adresem \$3F01, ... , w ostatnim, trzydziestym drugim kroku ostatnia wartość z tablicy zostanie umieszczona pod adresem \$3F1F.

PPU przechowuje palety tła pod adresami \$3F00 - \$3F0F a adresu spróto'ów pod adresami \$3F10 - \$3F1F. W powyższych kodzie korzystamy z faktu

”ciągłości” tego obszaru - nie musimy osobno ustawać adresu na początek obszaru palet sprite’ów, przesyłając za jednym zamachem wszystkie dane.

Czas na przetestowanie zmian. Po uruchomieniu emulatora pierwszą sprawą mogącą rzucić się w oczy jest inny kolor tła w oknie głównym w przypadku gdy domyślnie emulator miał ustawione inny ”współdzielony” kolor. W celu sprawdzenia czy palety zostały pomyślnie ustawione, należy przejść do menu *debug -> PPU Viewer* a następnie do zakładki *Palette Viewer*.



Rysunek 24: Podgląd ustalonej przez program palety kolorów w emulatorze Mesen.

Na obrazku powyżej wygląda na to, że wszystko działa jak należy. To okno pozwala również na ustawienie innych kolorów w trakcie działania, wystarczy kliknąć w odpowiednią komórkę a następnie wybrać pożądany kolor; zmiana nastąpi natychmiastowo. Przydatna sprawa przy eksperymentowaniu.

- 3.3 Grafika - Sprite'y**
- 3.4 Grafika - Tło**
- 3.5 Synchronizacja pętli głównej programu z NMI**
- 3.6 Rozpoznawanie regionu konsoli**
- 3.7 Drugi ekran z tłem, przewijanie ekranu**
- 3.8 Podprogramy / procedury**
- 3.9 Liczenie czasu**
- 3.10 Animacja grafiki**
- 3.11 Podział projektu na pliki**
- 3.12 Obsługa kontrolera**
- 3.13 Implementacja trybów programu**

4 Informacje o platformie

- 4.1 Zero Page**
- 4.2 NMI**
- 4.3 PPU**
- 4.4 Mapper**
- 4.5 Pattern Table**
- 4.6 Nametable**
- 4.7 Paleta barw**

5 Narzędzia

5.1 CA65

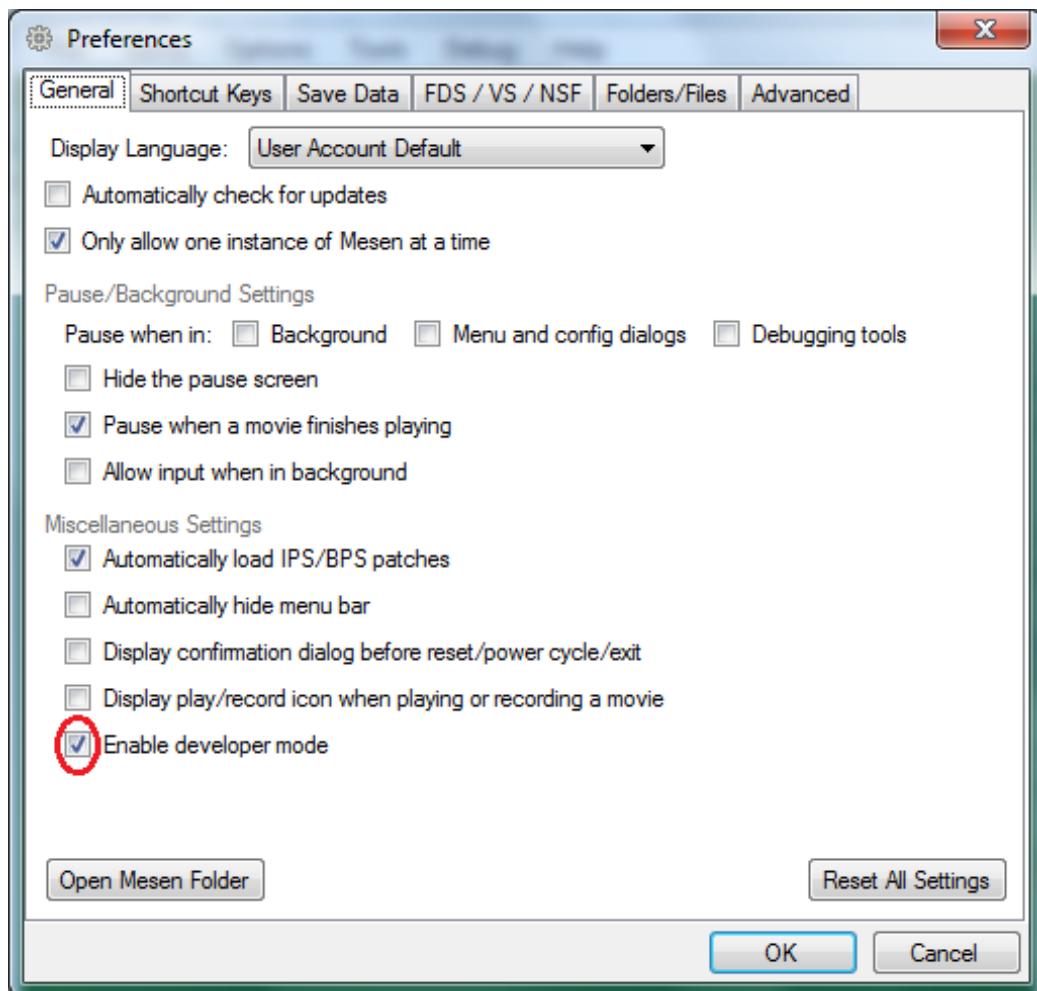
Strona projektu: <https://cc65.github.io/> CA65 jest częścią większego pakietu zwanego CC65. Jest on zbiorem narzędzi do kompilacji kodu napisanego w je-

zyku C na sprzętę z procesorem MOS 6502. Za to CA65 jest asemblerem pod ten sam procesor. W przypadku NES język C stanowi problem wydajnościowy, gdyż plik wynikowy który powstaje po komplikacji kodu w C jest znacznie wolniejszy niż odpowiednik asemblerowy; warto pamiętać iż ta konsola jest wielokrotnie mniej wydajna niż współczesne sprzęty. Stąd moja decyzja o pisaniu kodu tylko i wyłącznie w asemblerze.

(UWAGI: zestaw kompilator plus linker zasługuje na większe wyróżnienie od niżej opisanych programów pomocniczych do grafiki czy muzyki; do przemyślenia miejsce w dokumencie)

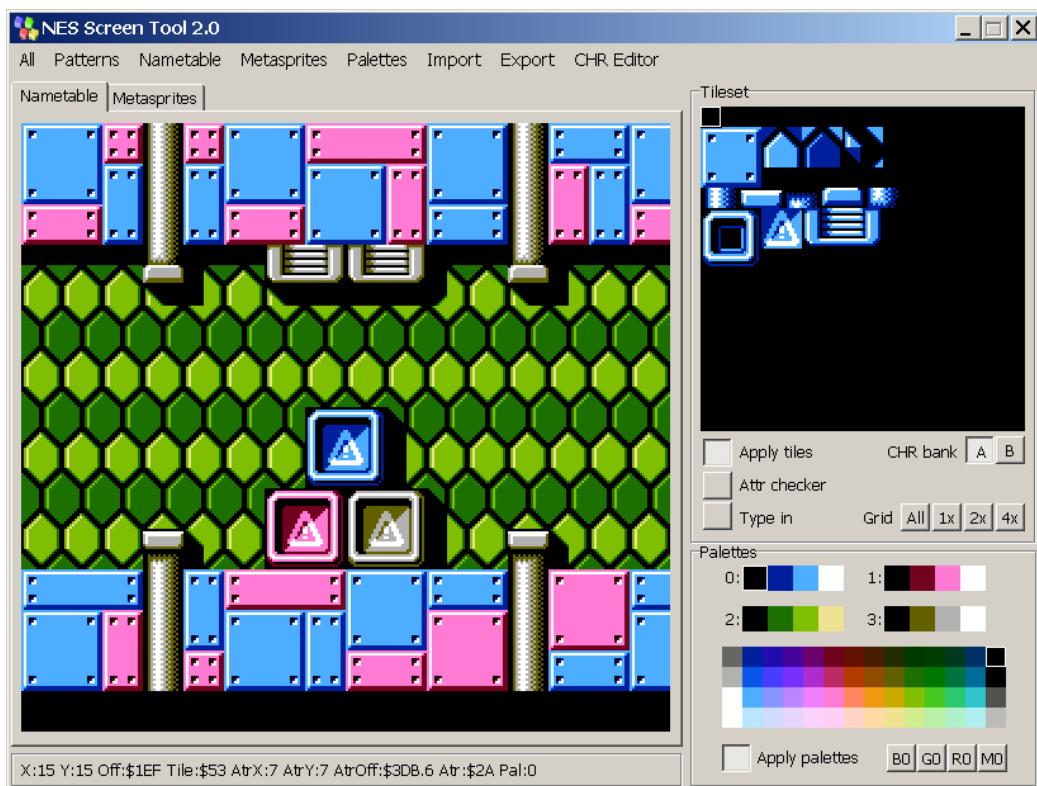
5.2 Mesen

Emulator znacząco wspiera proces tworzenia oprogramowania pod NES, gdyż zawiera wiele narzędzi pozwalających na zbadanie zachowania naszego programu. Aby mieć dostęp do narzędzi deweloperskich, musimy w menu *Options -> Preferences* zaznaczyć checkbox "Enable developer mode". Od tej pory w menu jest widoczna nowa pozycja - *debug*.



Rysunek 25: Przełącznik opcji deweloperskich w emulatorze Mesen.

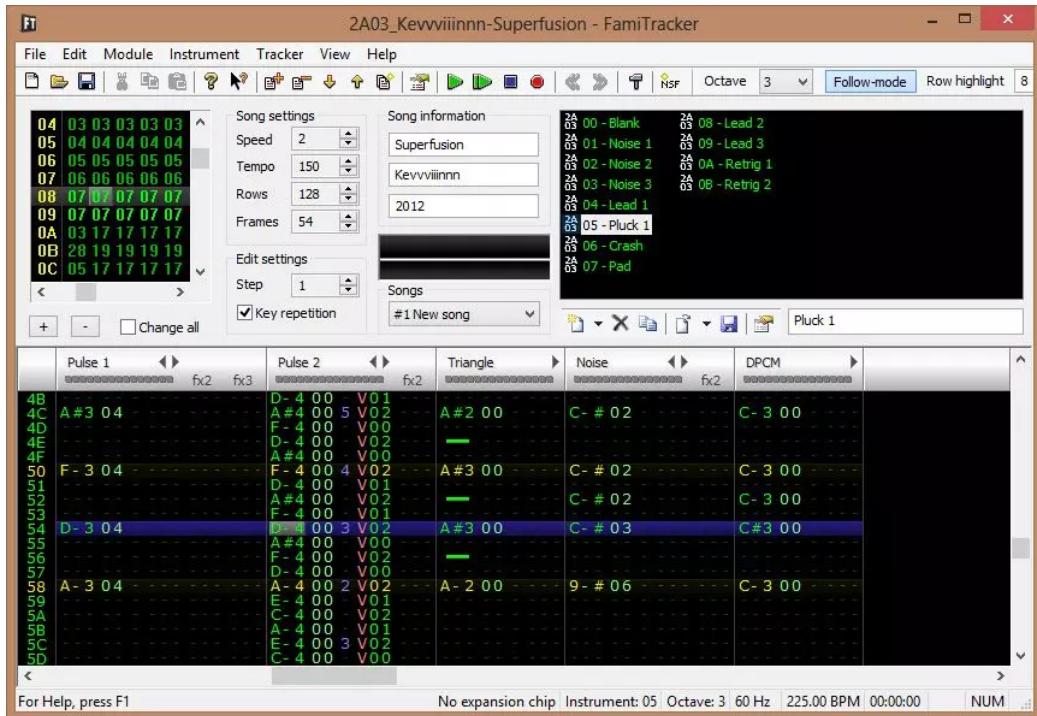
5.3 NES Screen Tool



Rysunek 26: Ekran główny aplikacji [23]

NES Screen Tool to program wspomagający tworzenie grafiki zgodnej z platformą NES / Famicom. Aplikacja pozwala na szczegółowe tworzenie grafik per pixel jak i bardziej ogólne działania jak eksport przygotowanych danych do formatu zrozumiałego przez konsolę, gotowego do zastosowania w kodzie programu. Strona domowa [22].

5.4 FamiTracker



Rysunek 27: Ekran główny aplikacji [25]

FamiTracker to program pozwalający na tworzenie muzyki w stu procentach kompatybilnej z platformami NES / Famicom. Aplikacja należy do rodziny trackerów - programów komputerowych w których komponuje się muzykę wykorzystując połączenie zapisu nutowego oraz poleceń sterujących efektami [26]. Strona domowa [24].

Literatura

- [1] <https://upload.wikimedia.org/wikipedia/commons/0/06/Nintendo-Famicom-Console-Set-FL.jpg>
- [2] <https://upload.wikimedia.org/wikipedia/commons/8/82/NES-Console-Set.jpg>
- [3] https://upload.wikimedia.org/wikipedia/commons/f/f1/New_Famicom.jpg
- [4] <https://upload.wikimedia.org/wikipedia/commons/e/e0/NES-101-Console-Set.jpg>

- [5] <https://levelupvideogames.com/used-messiah-generation-next-468309594161/>
- [6] <http://forum.pegasus-gry.com/index.php?topic=2315.0>
- [7] <https://arhn.eu/2018/04/historia-pegasusa-z-klawiatura-glk-2004-et-al/>
- [8] <https://www.ign.com/articles/2000/06/10/game-axe-color>
- [9] <https://upload.wikimedia.org/wikipedia/commons/7/7a/RetroUSB-AVS-Console-wController-FL.jpg>
- [10] https://cdn10.bigcommerce.com/s-2l8ru96d/products/223/images/1135/ret5bk_54484.1567
- [11] <https://www.amazon.com/Controllers-Children-Birthday-Childhood-Memories/dp/B083RBYRYN>
- [12] <http://www.retrogamingmuseum.com/the-collection/micro-genius-nes-clone>
- [13] <https://archiwum.allegro.pl/oferta/konsola-pegasus-iq-502-pady-pudelko-plomba-i7530854084.html>
- [14] <https://youla.ru/moskva/hobbi-razvlecheniya/konsoli-igry/dendy-classic-2-novaia-5b6c9ae1cf689a85567f44a6>
- [15] https://en.wikipedia.org/wiki/Nintendo_Entertainment_System
- [16] https://en.wikipedia.org/wiki/Nintendo_Entertainment_System_hardware_clone
- [17] Patrick Diskin: *Nintendo Entertainment System Documentation*, 2.1 2A03 Overview, <http://www.nesdev.com/NESDoc.pdf>
- [18] http://wiki.nesdev.com/w/index.php/CPU_memory_map
- [19] http://wiki.nesdev.com/w/index.php/PPU_registers#Summary
- [20] <http://wiki.nesdev.com/w/index.php/APU#Specification>
- [21] <http://forums.nesdev.com/viewtopic.php?t=7329>
- [22] <https://shiru.untergrund.net/software.shtml>
- [23] <https://shiru.untergrund.net/pic/nesst.png>
- [24] <http://famitracker.com/>

- [25] <https://blog.uptodown.com/wp-content/uploads/Famitracker-pianola.jpg.webp>
- [26] [https://pl.wikipedia.org/wiki/Tracker_\(oprogramowanie_muzyczne\)](https://pl.wikipedia.org/wiki/Tracker_(oprogramowanie_muzyczne))
- [27] http://wiki.nesdev.com/w/index.php/Init_code
- [28] <http://wiki.nesdev.com/w/index.php/File:Savtool-swatches.png>