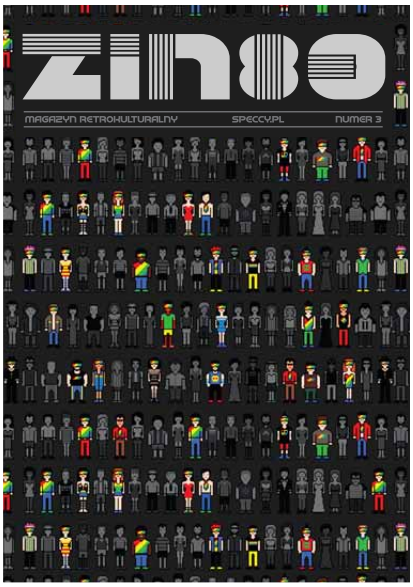
SPECCY.PL

ZINE

MAGAZYN RETROKULTURALNY

NUMER 3



MAGAZYN
UŻYTKOWNIKÓW
MIKROKOMPUTERÓW
Z PROCESOREM Z80

WYDAWANY PRZEZ
SERWIS
SPECCY.PL

REDAKCJA:

REDAKCJA:
TYGRYS – REDNACZ

AUTORZY ARTYKUŁÓW:
CHICADII
DALTHON
JT
MAT
MCKLAUD
SACHY
SIR DAVID
TOOLOUD
TYGRYS

OKŁADKA:
SLAYER

PROJEKT LOGOTYPU:
DKL

KOREKTA:
ENDER, MADDY, MAT/ESI

SKŁAD:
FAUST (HTTPS://T2E.PL)

KONTAKT:
TYGRYS @SPECCY.PL

WWW:
HTTPS://SPECCY.PL

3 . . 2 . . 1 . .

To miał być specjalny numer, wydany na speccy.pl party 2020.1. COVID-19 sprawił, że party zostało przeniesione na inny termin. Kolejną konsekwencją wirusa był termin wydania pisma – już nie sztywny termin a… „kiedy się uda". I oto masz przed sobą trzeci numer Zin80. Do trzech razy sztuka, jak to mówią. Osobiście mam nadzieję, że jeszcze będzie mi dane nie raz pisać wstępniaki, choć przyznam się, że podchodzę do tematu po raz dziesiąty.

Speccy.pl Party 2020.1 zostało przeniesione na jesień, choć nie ma jeszcze ustalonej ponownie daty imprezy – wszystko zależy od sytuacji epidemiologicznej. To też powód, dla którego nie obwieszczam, że kolejny numer ukaże się podczas party. Chciałbym aby tak było! Zwłaszcza, że gotowa jest specjalna „partyjna" okładka przygotowana przez Slayera. Z przyczyn oczywistych musiała więc powstać nowa okładka – i tym razem Slayer nie odmówił, czego efekt możemy podziwiać stronę wcześniej :)

Z wydaniem tego numeru zaczynają się dla mnie prace nad kolejnym. Mam nadzieję, że „do 3 razy sztuka" sprawi, że ten numer będzie interesujący. A co w nim? Na łamach numeru debiutują kolejni nowi autorzy – McKlaud, trojacek i tooloud piszący ciekawe teksty o sprzęcie, oraz, w przypadku tooloud – o polskich grach przygodowych. Stali autorzy również nie zawiedli – Chicadii opisał kolejne gry oraz programy użytkowe, Dalthon w swoim krótkim tekście przedstawił 8 najciekawszych jego zdaniem intr 256 bajtowych na ZX Spectrum, a Sir David odkrywa przed nami kolejne tajemnice BASICa dla SAMa Coupé.

Dla osób chcących poznać asembler, przygotowaliśmy aż trzy teksty! Sachy kontynuuje swój cykl o robieniu dem na ZX Spectrum. Mat, po 8 latach, w końcu napisał trzecią część o oldskulowych efektach, zaś ja – kolejny tekst o podstawach asm dla osób, które znają BASIC. Nad korektą czuwały tym razem 3 osoby, dzięki którym po raz kolejny ilość błędów dość mocno zmalała – mam nadzieję, że ich wysiłki widać ;] Na koniec, po raz kolejny, świetną robotę wykonał faust, nocami „wlewając" teksty i poprawiając ich skład w nieskończoność.

A co w kolejnym numerze? To wszystko zależy od Was – potencjalnych autorów tekstów. Mam nadzieję że zagoszczą komputery CPC oraz MSX, a może też jakieś inne „egzotyczne". Wszystko zależy od czytelników, którzy mogą stać się autorami ;)

W NUMERZE:

MAŁE JEST PIĘKNE, CZYLI 256 BAJTÓW

(EDYCJA 2019)

DALTHON/JOKER

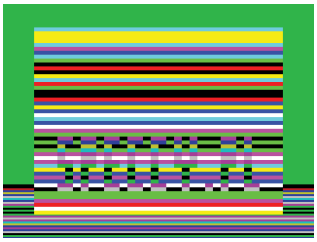
PIERWSZE 256-BAJTOWE (DOBRZE CZYTACIE: „-BAJTOWE”!) INTRA NA ZX SPECTRUM POJAWIŁY SIĘ DOŚĆ NIEDAWNO, BO DOPIERO W 2005 ROKU. NA INNYCH PLATFORMACH (CZYT. PC) BYŁY ZNANE JUŻ OD PONAD DEKADY, BO WYDAWAŁO SIĘ ŻE TEORETYCZNIE W TAKIEJ ILOŚCI KODU, CIĘŻKO COŚ NA Z80 ZROBIĆ. PIERWSZYM, KTÓRY PRAKTYCZNIE POKAZAŁ, ŻE JEDNAK MOŻNA, BYŁ GASMAN A POTEM JUŻ POLECIAŁO.

Na początku były to pojedyńcze produkcje, które pojawiały się wyłącznie na corocznym raww.orgy party, organizowanym przez Icaboda. Gdy imprezy Brytyjczyka zabrakło w 2011 roku, nie było odpowiednich bodźców by powstawały nowe intra. Sytuacja poprawiła się (i to dość znacznie) w 2014 roku, gdy praktycznie na każdym party organizowanym za naszą wschodnią granicą pojawiała się taka konkurencja.

Tyle jeśli chodzi o odległą historię. Spoglądając na tę bliższą – rok 2019 przedstawiał się całkiem nieźle. Powstało 20 nowych produkcji z czego aż 12 stworzonych przez naszych rodzimych koderów! Prawdziwym tytanem pracy okazał się MoNsTeR który to na speccy.pl party 2019.1 popełnił aż 6 zróżnicowanych interek. Co prawda nie udało mu się zająć miejsca na podium, ale zasłużone oklaski za pracowitość otrzymał.

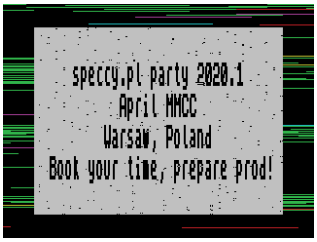
Poniżej subiektywny wybór ośmiu prac (działające na oryginalnych ZXach, więc produkcji wymagających rosyjskich wynalazków tutaj nie znajdziecie), które polecam obejrzeć, by przekonać się jakie cuda (i czary) można zmieścić w zaledwie 256b. Opisy może i lakoniczne, ale mają one Was zachęcić do odpalenia intra, a nie zanudzać szczegółami technicznymi czy też moimi wątpliwymi umiejętnościami pisarskimi. ■

KOLEJNOŚĆ ALFABETYCZNA (WSPAK) BY NIKOGO NIE FAWORYZOWAĆ.



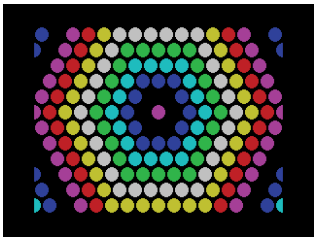
Voodoo Pink Phonex

Szalejące kolorowe paski (nie tylko na ekranie, ale ipoza nim) oraz nie mniej kolorowe napisy z nazwą party, na którym intro zostało wystawione. Ilość kolorów na bajt intra powyżej normy!



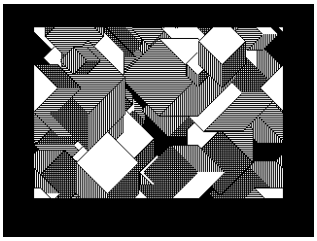
SPECCY.PL PARTY 2020.1 INVITATION Tygrys/Speccy.pl

Z chaosu wizualnego i dźwiękowego pomału wyłania się zaproszenie na nową odsłonę rodzimej imprezy dla fanów sprzętów SIR CLIVE'A SINCLAIRA. Tekst na ekranie to 75 bajtów, więc na sam efekt jeszcze mniej miejsca!



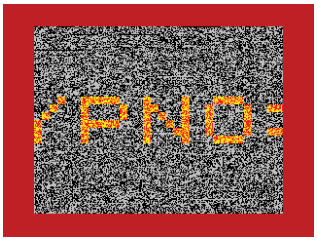
OpArt gOblinish

Ciekawy sposób wykorzystania atrybutów, by uzyskać z kolorowych kulek efekt... tunelu? Studni? Ciężko stwierdzić, ale wygląda to bardzo uspokajająco. Jakiś dźwięk byłby niezłym dopełnieniem, ale najwyraźniej zabrakło miejsca.



OH NO! MORE BOXES DALTHON/JOKER

Praca, której jestem autorem – proceduralnie generowane pudełka 3D oróżnych wymiarach ikolorach. Szkoda, że samo ich rysowanie nie jest zbyt szybkie, ale nie wymagajmy cudów od takiej programistycznej pchełki

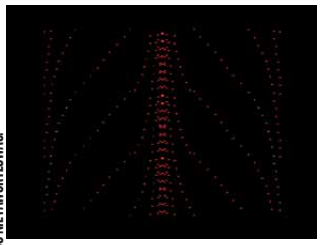


Hypnosis - MoNsTeR/GDC

Szybko przebiegający scroll, rytmicznie zmieniający się „biały szum” na ekranie a do tego pulsacyjny dźwięk i mamy przepis, jak zahipnotyzować oglądającego. Czujcie się ostrzeżeni. oglądanie tego maleństwa wciąga!

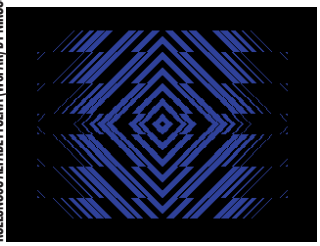
CAFe eXperience Artifact - Fyrex & Kael/ Mayhem

Pionowy zoom grafiki (tak, tak – grafiki w256b!) z odpowiednio dobranym dźwiękiem wydaje się czymś prostym. Macie rację! Wydaje się. Ciekawy pomysł w doskonałym wykonaniu.



Brain Waves - Dox/Joker

Tańczące kolorowe pixele, które nikogo nie pozostawią obojętnym! Ale to nie wszystko. Dochodzi do tego dźwięk, który perfekcyjnie oddaje to, co dzieje się na ekranie – słyszy się co widzi lub widzi się co słyszy.



Abbatone - TomCat & G.o.D. & ern0/Abaddon

Osiem dość zróżnicowanych wersji naprawdę niesamowitego efektu połączonych z minimalistyczną muzyką, która zmienia się w trakcie przejść pomiędzy kolejnymi fazami. Ta produkcja, to kolejny przykład, że granica tego co jest możliwe, a co nie, stale się przesuwa. Bezapelacyjnie najlepsze 256b intro roku 2019!

KILKA SŁÓW O ZX

MAMY POŁOWĘ LUTEGO 2020 ROKU, KTÓRY TO POZA ŚWIATOWĄ EPIDEMIĄ KORONAWIRUSA SARS-COV-2 PRZYNIÓSŁ, TAKŻE ODROBINĘ NADZIEI W ŚWIECIE RETRO KOMPUTERÓW. PO DŁUGICH PERTURBACJACH I PRAWIE DWULETNIM OPÓŹNIENIU, W RĘCE UŻYTKOWNIKÓW ZACZĄŁ TRAFIAĆ W PEŁNI KOMERCYJNY PRODUKT POD NAZWĄ ZX SPECTRUM NEXT. W TYM ARTYKULE NIE BĘDĘ ROZWODZIŁ SIĘ NAD DZIEJAMI ZBIÓRKI PIENIĘDZY, PRZEDPŁAT I OCZEKIWANIEM NA TEN KICKSTARTER. NALEŻY JEDYNIE WSPOMNIEĆ, IŻ PRODUKT DOCZEKAŁ SIĘ REALIZACJI, A NIE UMARŁ, JAK NIEGDYŚ VEGA+.

MCKLAUD

CZYM JEST ZX SPECTRUM NEXT?

Jest on komputerem zbudowanym na implementacji (symulacji układów logicznych) w układzie FPGA *(ang. field-programmable gate array)*, w którym nie ma fizycznego procesora Z80 czy układu dźwiękowego AY-3-8910. Elementy te zostały „wbudowane" w FPGA, podobnie jak zostało to zrobione w **MiST** czy **ZX Uno/DOS**.

W przypadku **Nexta** mogą nasunąć się następujące pytania:
→ Skoro są wcześniejsze implementacje **ZX Spectrum** w FPGA, to co wyróżnia **Nexta** spośród nich?
→ Skoro **ZX Omni 128HQ** jest dostępny prawie od ręki, po co kolejny klon **ZX Spectrum**?

Nie wiem, czy będę w stanie odpowiedzieć wyczerpująco na te pytania. Należy jednak pamiętać, że **MiST** powstał jako maszyna wieloplatformowa, w której ZX Spectrum jest jednym z dostępnych „rdzeni" wybieranych przez użytkownika. **ZX Uno** natomiast jest niewiele większe od karty kredytowej i także stało się kolejnym kombajnem z dostępem do bogatej biblioteki symulowanych komputerów. Oba są toporne w wyglądzie (jak na mój gust), zamknięte w przemysłowe lub drukowane obudowy oraz niewiele różnią się w obcowaniu z „ciastami owocowymi" tj. **Raspberry Pi (RPi)**. Brak im „feelingu", czy „duszy". Zwał jak zwał.

Na rynku brakowało **ZX Spectrum** z prawdziwego zdarzenia, z fizyczną klawiaturą, złączem rozszerzeń dla dużej rodziny dostępnych interfejsów i ze wszystkim co najlepsze z lat świetności Speccy we wnętrzu.

A co z **ZX Omni 128HQ** z Retro-Radionics? Tak, jest ono dostępne od ponad dwóch lat, jest dobrym komputerem zbudowanym bez układów wielkiej skali integracji. **Omni** zostało zamknięte w obudowę od „gumiaka", otrzymało bateryjki, opcjonalny ekran LCD i można nosić je w plecaku jak latopa. Jednak **Omni** nie

SPECYFIKACJA TECHNICZNA

ŹRÓDŁO: WWW.SPECNEXT.COM:

PROCESOR: Z80 W FPGA (SPARTAN 6, XC6SLX16) Z TRYBAMI: NORMAL I TURBO,

PAMIĘĆ: 1 MB RAM Z MOŻLIWOŚCIĄ ROZBUDOWY DO 2 MB

UKŁAD GRAFICZNY: 256X192 Z 256 KOLORAMI Z PALETY 512, TRYBY ULA+ I TIMEXA 8+1 ORAZ SPRZĘTOWE DUSZKI I SPRZĘTOWY SCROLL

WYJŚCIA WIDEO: RGB, HDMI I VGA

PAMIĘĆ MASOWA: KARTA SD Z PROTOKOŁEM ZGODNYM Z DIVMMC

DŹWIĘK: 3 X UKŁAD AY-3-8912 W FPGA Z WYJŚCIEM STEREO

PORTY JOYSTICKA: 2 X DB9 ZGODNE Z PROTOKOŁAMI CURSOR, KEMPSTON I IF2

PORT PS/2: KLAWIATURA ZEWNĘTRZNA ORAZ MYSZ W TRYBIE EMULACJI KEMPSTON

WEJŚCIE/WYJŚCIE AUDIO NA MAGNETOFON

ZŁĄCZE KRAWĘDZIOWE: ZGODNE Z ZX SPECTRUM

URZĄDZENIA OPCJONALNE: AKCELERATOR NA RASPBERRY PI ZERO, UKŁAD ZEGARA CZASU RZECZYWISTEGO (RTC), WEWNĘTRZNY GŁOŚNICZEK, MODUŁ WI-FI (ESP-01).

SPECTRUM NEXT

wyszło poza możliwości sprzętowe oryginalnego **ZX Spectrum Plus 128K**. **ZX Omni 128HQ** wywodzi się wprost z klona o nazwie **Harlequin 128K**, ma swoje zalety, ale z jakością obudowy i jej spasowaniem bywa różnie. Posiadałem je przez jakiś czas, wraz ze sporą liczbą innych własnoręcznie zbudowanych klonów, ale nadal brakowało mi w nim tego czegoś. Jest i działa. **Omni** jest kompletne, ale w sumie takie **ZX Spectrum** sam mogę zbudować i nic mu nie będzie brakowało. Jak na produkt komercyjny miałem niedosyt powiewu świeżości, nowinek technicznych, wyjścia poza schemat **ZX Spectrum**, powielany w każdym zakątku świata (może poza Chinami i Japonią) od ponad 35 lat. Proszę nie brać mojego zdania za przytyk w stronę Dona „*Superfo*", całego sztabu „hiszpańskiej inkwizycji" i firmy RetroRadionics – osób, które wykonały kawał dobrej inżynierskiej roboty przy **Harlequinie** i **ZX Omni**.

Na tym polu **ZX Spectrum Next** wypada naprawdę imponująco, ponieważ poza nowoczesną implementacją całego serca Speccy i zgodności z istniejącymi interfejsami, projektanci dołożyli kilka nowości i smaczków. W projekt zaangażowały się znane osoby w świecie Spectrum np. nieżyjący już Rick Dickinson, Garry Lancaster, Jim Bagley oraz firma Sky PLC, będąca właścicielem praw autorskich do produktów ze znakiem **ZX Spectrum** oraz ROMu.

W solidnym i kolorowym pudełku z komputerem dostarczony jest zasilacz 9 V z wtyczkami do każdego typu gniazdka, drukowana instrukcja obsługi w języku angielskim i karta SD z firmwarem.

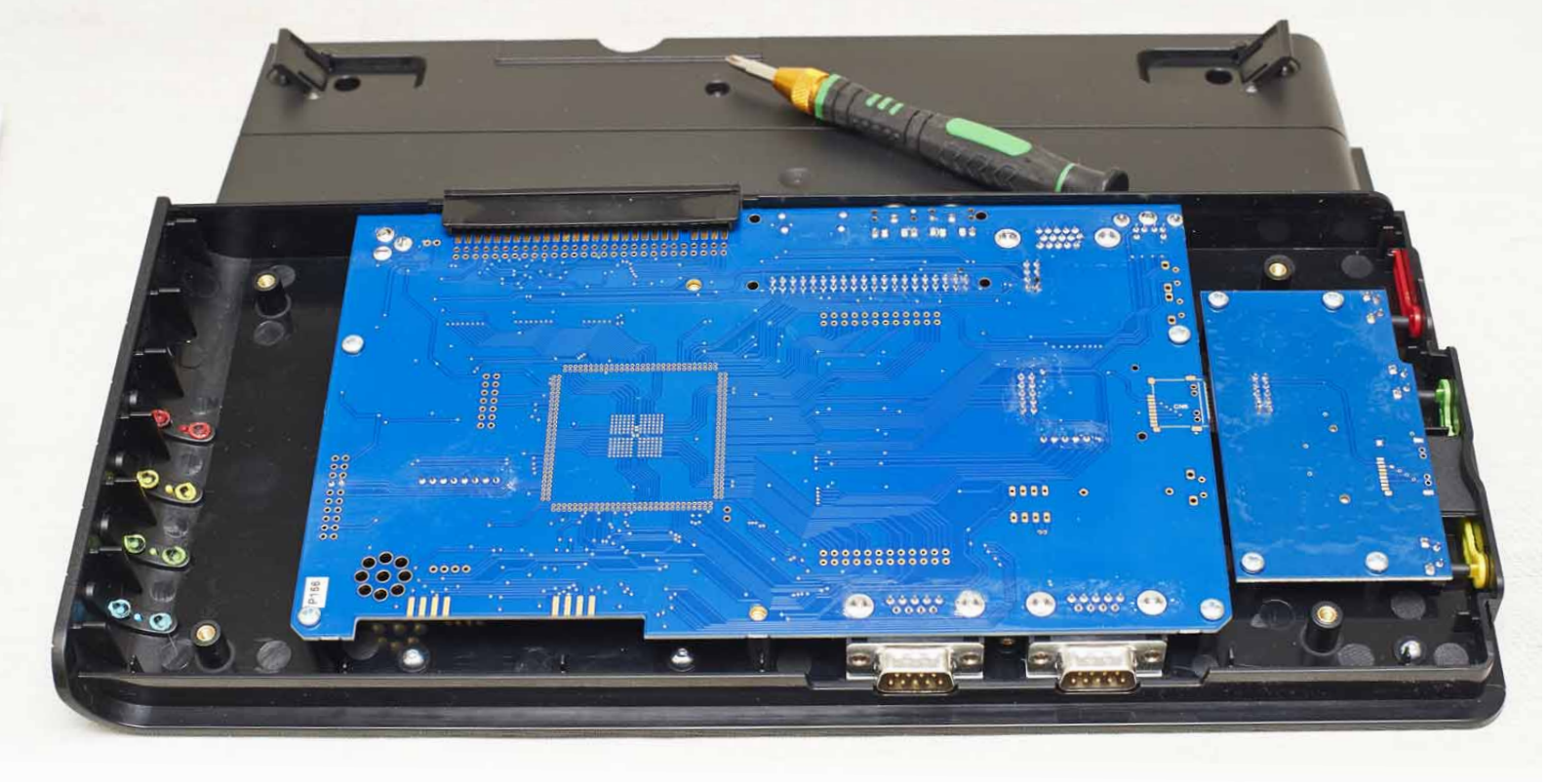
OBUDOWA

Nie ma co bić piany, obudowa **Nexta** sprawia bardzo dobre wrażenie swoją prostotą, funkcjonalnością oraz pięknem. Za jej projektem stoi Rick Dickinson, projektant obudowy **ZX81** i **ZX Spectrum**. Osobiście do zakupu **Nexta** skłoniła mnie tylko obudowa i klawiatura, a tak naprawdę jej wizualizacje pokazane w momencie zapisów na ten komputerek. *„De gustibus non disputandum est"* jak mawiali starożytni i jej urok należy ocenić samemu.

Jest kilka rzeczy, które osobiście nie podobają mi się w obudowie, np. błyszczący tylny i lewy panel czy wypolerowane na wysoki połysk części klawiszy klawiatury. O ile lewy panel będzie tylko ciągle umazany i brudny, o tyle tylna część obudowy dość szybko zostanie porysowana wtyczkami. Ponadto błyszczące części obudowy noszą ślady procesu technologicznego z wtryskarek. Są na nich drobne zniekształcenia i nierówności.

Po zamontowaniu Raspberry Pi Zero, jego gniazdo HDMI „straszy" w otworze bez maskownicy i nijak nie pasuje do reszty obudowy. Cóż, jakaś niedoróbka musiała się znaleźć.

Sama klawiatura sprawia wrażenie solidnej, klawisze mają niewielki skok, działają pewnie i wygodnie. Odczucia przy pisaniu są podobne to zwykłej klawiatury laptopowej. Jest jednak jedna rzecz związana z klawiaturą, która mi nie pasuje. Wszystkie napisy na klawiszach są malowane i niestety grubość farby jest wyczuwalna pod



opuszkami palców. Ciekawe jaka będzie trwałość tych napisów, ale tylko dłuższe używanie **Nexta** może na to pytanie odpowiedzieć. O wypolerowanych powierzchniach klawiszy wspomniałem już powyżej i uważam, że są one wadą estetyczną. Klawiatura zawsze będzie sprawiała wrażenie brudnej, ze smugami. Utrzymanie jej w czystości jest niemożliwe.

Nie należy sugerować się zdjęciami i filmikami z sieci, recenzjami youtuberów czy innej maści „znawców tematu", bo **Nexta** trzeba doświadczyć namacalnie, aby wyrobić sobie o nim zdanie.

PORNO-FLACZKI W HI-RES

Wersja **Nexta,** którą kupiłem jest wersją w podstawowej konfiguracji, z 1MB RAM, bez zegara czasu rzeczywistego (RTC), akceleratora RPi, głośniczka i modułu WiFi. Elementy te dołożyłem we własnym zakresie.

Płyta główna wygląda schludnie, klawiatura ma 3 „wąsy" do podłączenia, „płetwa" ze złączem karty SD jest łatwo demontowalna, a reszta została pokazana na zdjęciach.

TAB. 1. ZESTAWIENIE KOMPUTERÓW

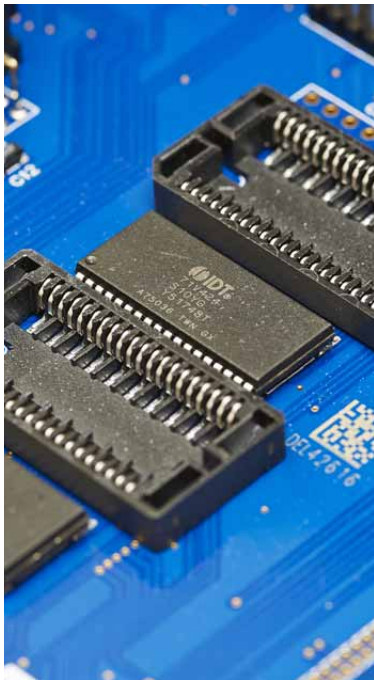
						
ELEMENT	**MIST**	**ZX UNO**	**ZXDOS**	**ZX SPECTRUM NEXT**	**ZX OMNI 128HQ**	**JUST SPECCY 128K**
FIZYCZNY PROCESOR	BRAK	BRAK	BRAK	BRAK	ZILOG Z80A	ZILOG Z80A
PAMIĘĆ RAM	SDRAM 8MB X 8BIT 4MB X 16BIT	SRAM 512KB - 2MB	SRAM 512KB + SDRAM	SRAM 1 - 2MB	SRAM 128KB	DRAM 128KB
UKŁADY DUŻEJ SKALI INTEGRACJI	CYCLONE III EP3C25 (FPGA)	SPARTAN 6 XC6SLX9 (FPGA)	SPARTAN 6 XC6SLX16 (FPGA)	SPARTAN 6 XC6SLX16 (FPGA)	BRAK	DEDYKOWANE ULA I PCF
UKŁAD DŹWIĘKOWY	W FPGA	W FPGA	W FPGA	W FPGA (3 X AY)	AY-3-8910	AY-3-8912
PAMIĘĆ MASOWA	KARTA SD	KARTA SD (DIVMMC)	KARTA SD (DIVMMC)	KARTA SD (DIVMMC)	KARTA SD (DIVMMC)	KARTA SD (DIVMMC)
SYS. ZŁĄCZE ROZSZERZEŃ	BRAK	WŁASNE	WŁASNE	ZGODNE Z ZX SPECTRUM	ZGODNE Z ZX SPECTRUM	ZGODNE Z ZX SPECTRUM
KLAWIATURA	ZEW. USB	ZEW. PS/2	ZEW.PS/2	TAK	TAK	BRAK
OBUDOWA	TAK	TAK	TAK	TAK	TAK	BRAK

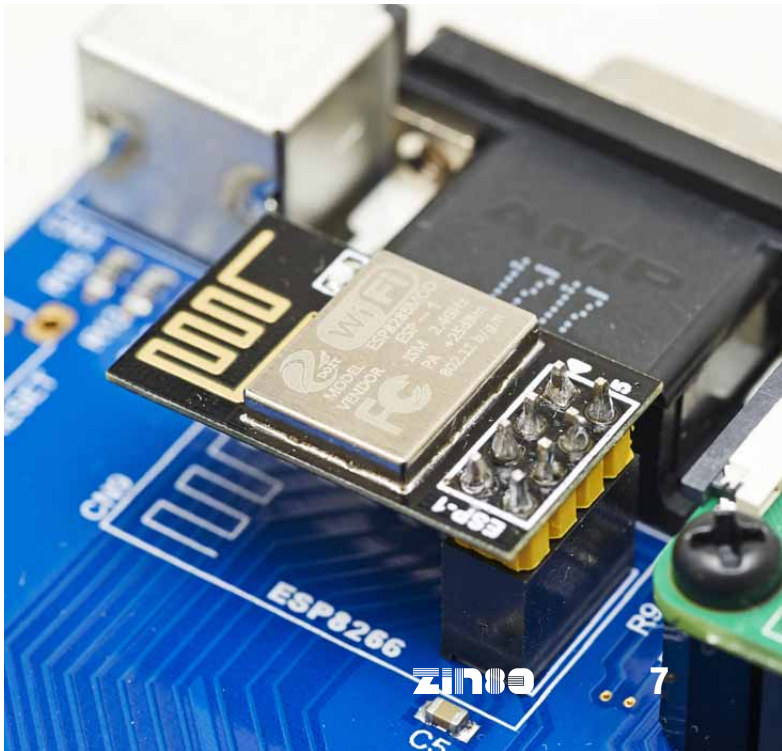
Niestety konstruktorzy popełnili błąd w układzie zasilania, stosując „standardową" polaryzację zasilania z plusem na pinie. W świecie Speccy, od zarania dziejów było i jest odwrotnie, **„plus"** jest na zewnątrz, a **„minus"** na pinie w gnieździe. Ponadto w obwodzie zasilania nie ma żadnego zabezpieczenia przed odwrotną polaryzacją napięcia wejściowego, w postaci mostka Graetza albo diody prostowniczej. Przez nieuwagę lub przyzwyczajenie można odesłać drogocenny komputer do krainy wiecznych ośmiu bitów.

Innym niedopatrzeniem jest złącze kołkowe (męskie) na akcelerator RPi, które jest wlutowane w płytę główną. Standardowo RPi Zero z fabrycznie zamontowanym złączem GPIO, ma je w wersji męskiej. Do samodzielnego uszlachetnienia **Nexta** należy zaopatrzyć się w „malinkę" bez złącza GPIO, zakupić złącze żeńskie 2x20 pinów i przylutować je do Raspberry przed jego montażem.

O ile sam montaż modułu Wi-Fi (typu ESP-01) i akceleratora nie wymaga dodatkowych narzędzi poza śrubokrętem, o tyle rozszerzenie o RTC i głośnik może nastręczać trudności komuś nieobytemu z lutownicą. Do zegara czasu rzeczywistego, poza układem DS1307, potrzebne są także dodatkowe elementy, tj. postawka DIP-8, kwarc zegarowy o częstotliwości 32.768 kHz, uchwyt na baterię i jedna pastylka CR2032. Głośniczek o średnicy 15 mm można przylutować wprost do płyty głównej, bo nie ma na niej złącza kołkowego do jego podpięcia.

Tyle w telegraficznym skrócie o tym co dobre, a co złe we wnętrzu **ZX Spectrum Next**.





PODSUMOWANIE

Ktoś może zadać mi pytanie o stosunek wartości do ceny. W 2017 roku na „kickstarterze" za **Nexta** zapłaciłem 185 funtów i ta kwota obejmowała wysyłkę. Obecna oficjalna cena tej konfiguracji wynosi ponad 250 funtów / 285 euro, a na aukcjach internetowych cena osiąga pułap 500 funtów. Z drugiej strony za **MiSTa** trzeba zapłacić 210 euro. W przypadku **ZX Omni 128HQ** musimy liczyć się z wydatkiem rzędu 135 euro, za **ZX Uno** w obudowie ok 100 euro, a **Just Speccy 128K** jest dostępne za 150 euro. W przypadku **Omni** dostaniemy pełnowartościowy produkt gotowy do użytku, do **ZX Uno** będziemy potrzebowali klawiaturę na PS/2, a **Just Speccy** przyjedzie w postaci samej płyty głównej. Na tle tej konkurencji cena za **Nexta** wydaje się niewygórowana, choć do niskich nie należy. Jeśli miałbym ponownie wybierać, skłaniałbym się w stronę **Nexta**. ■

c.d.n.

(W następnym odcinku historia ZX Spectrum Nexta)

Shadow of The Beast

ZX Spectrum Next
(tech demo)

RETROPROJEKTY NA NOWO,

CZYLI CZY MOŻNA WYPRODUKOWAĆ INTERFEJS DO RETROKOMPUTERA NA PODSTAWIE MATERIAŁÓW SKANOWANYCH?

RETROMANIA OPANOWAŁA ŚWIAT 40-LATKÓW NICZYM WIRUS, A CENY SPRZĘTU SZYBUJĄ W GÓRĘ BEZ OPAMIĘTANIA. NIE WSZYSTKIE INTERFEJSY SĄ DOSTĘPNE, NIEKTÓRE SĄ DROGIE LUB BARDZO DROGIE, A CZĘŚĆ Z NICH ZAGINĘŁA W KOLEJCE DZIEJÓW. PONADTO SPORA CZĘŚĆ Z POMYSŁÓW I PROJEKTÓW Z LAT ŚWIETNOŚCI 8-BITÓW NIGDY NIE DOCZEKAŁA SIĘ SERYJNEJ PRODUKCJI ALBO BYŁA TYLKO PUBLIKOWANA NA ŁAMACH PERIODYKÓW O KOMPUTERACH LUB ELEKTRONICE DLA HOBBYSTÓW. JEDNYM Z TAKICH INTERFEJSÓW BYŁ MODUŁ KOLORÓW DO JUPITERA ACE. ZACZNIJMY JEDNAK OD POCZĄTKU, CZYLI CZYM BYŁ JUPITER ACE.

JUPITER ACE

Jupiter ACE był brytyjskim komputerem ośmiobitowym z procesorem Z80, zbudowanym bez użycia układów o dużej skali integracji. Został zaprojektowany i zbudowany przez dwóch byłych współpracowników Sir Clive'a Sinclaira, Richarda Altwassera i Stevena Vickersa.

W Sinclair Research Ltd. byli oni zaangażowani w przygotowanie i rozwój ZX81 oraz ZX82 znanego pod oficjalną nazwą ZX Spectrum. Na początku 1982 r. obaj zakończyli współpracę z Sinclairem i założyli własną firmę pod nazwą Jupiter Cantab Ltd. Od podstaw zaprojektowali, zbudowali i wdrożyli do produkcji mikrokomputer o nazwie Jupiter ACE. Cały proces trwał mniej niż 9 miesięcy i sprzęt oficjalnie ujrzał światło dzienne 22 września 1982 r. z ceną w sprzedaży 90 funtów. Komputer nie zdobył serc użytkowników z kilku powodów, głównym był czarno-biały obraz, złożony ze znaków semigrafiki, dowolnie programowalnej przez użytkownika, w dobie wszechobecnych kolorowych 8-bitowych komputerów konkurencji. Dla ówczesnego użytkownika brak kolorów i brak trybu graficznego z prawdziwego zdarzenia był krokiem wstecz. Jupiterowi nie pomógł także wbudowany kompilator języka FORTH, zamiast popularnego BASICa. Ponadto komputer posiadał tylko 3KB pamięci RAM, kiedy konkurencja oferowała jako standard 48KB lub 64KB w podobnej cenie.

```
vlist
FORTH UFLOAT INT FNEGATE F/ F* F
+ F- LOAD BVERIFY VERIFY BLOAD B
SAVE SAVE LIST EDIT FORGET REDEF
INE EXIT ." ( [ +LOOP LOOP DO UN
TIL REPEAT BEGIN THEN ELSE WHILE
 IF ] LEAVE J I' I DEFINITIONS V
OCABULARY IMMEDIATE RUNS> DOES>
COMPILER CALL DEFINER ASCII LITE
RAL CONSTANT VARIABLE ALLOT C, ,
 CREATE : DECIMAL MIN MAX XOR AN
D OR 2- 1- 2+ 1+ D+ - + DNEGATE
NEGATE U/MOD */ * MOD / */MOD /M
OD U* D< U< < > = 0> 0< 0= ABS O
UT IN INKEY BEEP PLOT AT F. EMIT
 CR SPACES SPACE HOLD CLS # #S U
. . SIGN #> <# TYPE ROLL PICK OV
ER ROT ?DUP R> >R ! @ C! C@ SWAP
 DROP DUP SLOW FAST INVIS VIS CO
NVERT NUMBER EXECUTE FIND VLIST
WORD RETYPE QUERY LINE ; PAD BAS
E CURRENT CONTEXT HERE ABORT QUI
T  OK
 ■
```

ZRZUT EKRANU Z VLIST

Pomimo swojej prostoty w budowie oraz założonych ograniczeń sprzętowych Jupiter ACE posiadał dwa złącza rozszerzeń, jedno zwane złączem pamięci/systemowym i drugie zwane złączem graficznym/drukarki. Pierwsze złącze krawędziowe pozwalało na podpinanie dedykowanych modułów pamięci 16KB lub peryferiów od ZX81 poprzez dodatkowe „międzymordzie". Konstruktorzy komputera wiedzieli, że nie będą w stanie konkurować z ilością interfejsów dostępnych dla schodzącego z rynku ZX81, a tworzenie nowych jest ekonomicznie nieuzasadnione. Do Jupitera napisano specjalne sterowniki, aby korzystać z drukarki termicznej ZX Printer, a firma Memotech wypuściła dedykowaną klawiaturę zewnętrzną na bazie swojej klawiatury do ZX81.

Drugie gniazdo rozszerzeń nigdy nie doczekało się oficjalnie drukarki pracującej na tym porcie, ani modułu kolorów zapowiadanego

EKRAN EIGHTY ONE Z WŁĄCZONYM ETI I OBRAZEM TESTOWYM

przez Jupiter Cantab. Dlaczego? Z jednego prozaicznego powodu – firma zawiesiła sprzedaż komputerów we wrześniu 1983 r. W 1984 r. Jupiter Cantab został wykupiony przez przez Boldfield Computing Ltd. Części lub kity do budowy komputera Jupiter ACE były dostępne w sprzedaży do końca 1984 r. Sytuację próbowano ratować poprzez wypuszczenie na rynek amerykański Jupitera ACE 4000, który był lekko ulepszonym wariantem wersji podstawowej, bez większych zmian konstrukcyjnych. Następnym krokiem była wyprzedaż stanów magazynowych w zestawie z „plecaczkiem" 16KB RAM pod nazwą Jupiter ACE 16+ za ok. 30 funtów.

Mogłoby się wydawać, że wraz z końcem 1984 r. historia Jupitera dobiegła końca, ale tak nie było. Komputer sprzedał się w ok. 8 tysiącach sztuk, trafił do sporej rzeszy entuzjastów i ze względu na zgodność na poziomie sygnałów na złączu krawędziowym z ZX81 i szybkość obliczeń wykonywanych w FORTH, na łamach różnych czasopism do mniej więcej 1987 r. pojawiały się interfejsy dla tego komputera. W kwietniu 1984 r. w wydaniu brytyjskim *Electronics Today International* został opublikowany obszerny artykuł pt. „*Adding colour to the ACE*".

Artykuł traktował o tym w jaki sposób Jupiter ACE generuje obraz oraz w jaki sposób zbudować interfejs kolorów do tego komputera. Zamieszczono w nim schemat aplikacyjny z dokładnym opisem, zdjęcie gotowego interfejsu, projekt płytki, instrukcje montażu oraz przykłady wykorzystania interfejsu w programach w FORTH. Z tego co mi


PROJECT

ADDING COLOUR TO THE ACE

Black and white is boring, so add a little colour (or a lot, if you want) to your computer's life with this project from John Wike. Additional material by Phil Walker.

The two facilities that really enhance the output of a microcomputer are sound and colour. The Jupiter Ace already has sound of sorts and this project provides a means of adding colour to its monochrome display. Eight colours including white and black are available, any of which may be selected as foreground (ink) or background (paper). The circuit is active from switch-on and requires no special software for monochrome operation. Thus programs may be listed, edited and run without the need to keep swapping over aerial leads or operating systems. The board uses the smaller edge connector and contains a RAM (addressed in parallel with the Ace video RAM) to hold the colour information for each of the 768 locations on the screen.

paper. The attribute number is obtained by adding the ink value to the paper value and adding 128 (80h).

Colour	Ink	Paper
Black	0	0
Blue	1	16 (10h)
Red	2	32 (20h)
Purple	3	48 (30h)
Green	4	64 (40h)
Cyan	5	80 (50h)
Yellow	6	96 (60h)
White	7	112 (70h)

Attributes

Each character written on the screen will have associated with it the current attribute describing its ink and paper colours. To print anything with different colours the current attribute must first be changed. This is easy to implement and for doing so is given

About The Ace

The Jupiter Ace video display consists of 24 rows of 32 characters, each of which is selected by storing the required code at the appropriate location in the video RAM. There are 128 characters or shapes (plus their inverses) available and these are re-definable by the user.

At the rear of the comp

EDITORIAL COMMENT

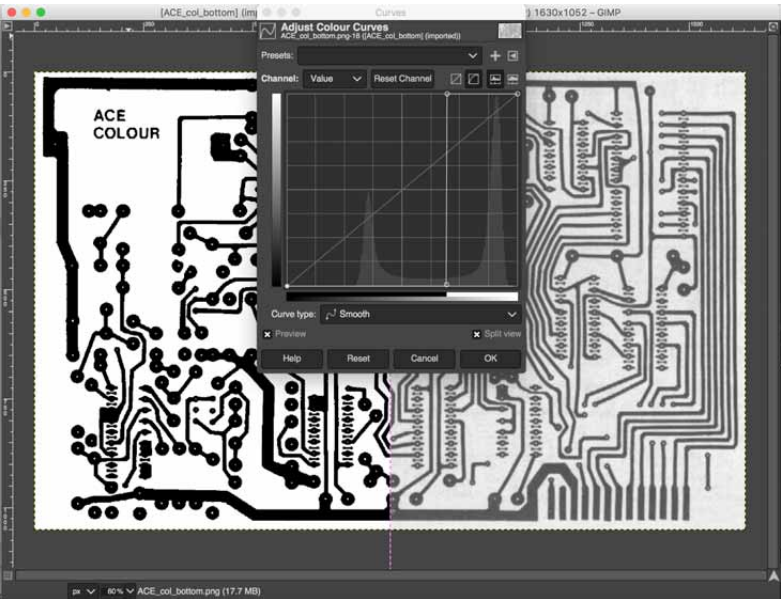
It may seem a little strange to publish a project for the Jupiter Ace months after the manufac- into liquidation. a number of and,

wiadomo, do czasów teraźniejszych nie przetrwał ani jeden egzemplarz tego interfejsu, ale zawsze jest jakieś drugie dno. Tym razem są nim emulatory Jupitera ACE, a konkretnie emulator *Eighty One* (EO), który potrafi emulować interfejs ETI.

Ostatnie półtora roku, w tym budowa repliki jak najbardziej zbliżonej do oryginału oraz zaprojektowanie i uruchomienie kilku dodatkowych zabawek do Jupitera, pchnęły mnie w stronę tego ciekawego interfejsu. Nosiłem się z zamiarem jego budowy od dłuższego czasu i zrobiłem kilka podejść do tematu, wliczając w to przeprojektowanie interfejsu, z wykorzystaniem współcześnie dostępnych układów lub układów w wersji do montażu powierzchniowego. Ostatecznie wybór padł na zbudowanie interfejsu w takiej formie, w jakiej został on zaprojektowany w 1984 r.

OD SKANU DO LAMINATU

Tyle wystarczy na przydługawy wstęp. Przejdźmy teraz do projektowania płytek na podstawie fragmentarycznych informacji lub dokumentacji dostępnej w dość kiepskiej jakości w formie elektronicznej. Skan magazynu i projekt ETI Colour można znaleźć w archiwum poświęconym Jupiterowi ACE (www.jupiter-ace.co.uk). Dostępne tam materiały wydają się wystarczające, aby wykonać płytkę w domu np.

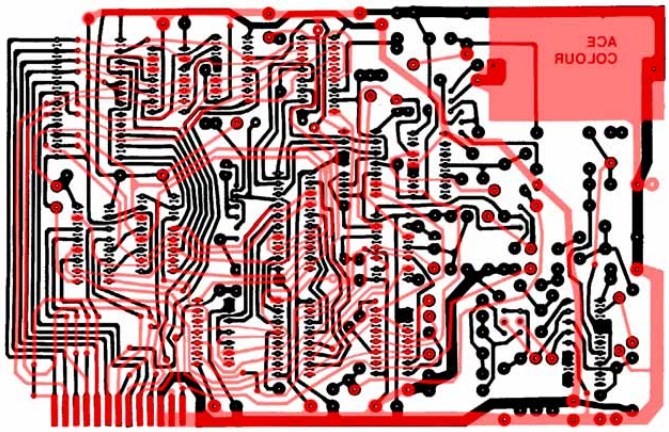


GIMP I PRZEJŚCIE Z OBRAZKA SZARO-SZAREGO NA CZ-B

metodą termotransferu. Jednak posiadając doświadczenie zdobyte przy uruchamianiu repliki Jupitera, zbudowanej na podstawie projektu PCB do samodzielnego wykonania, oraz dużą dozę ostrożności w podejściu do takich materiałów, nie rzuciłem się na „produkcję" z tego co jest. Przytoczony tu proces przypomina ślęczenie nad manuskryptem przez średniowiecznego mnicha benedyktyńskiego, z dostępem do nowych programów i technologii.

WALKA Z SZAROŚCIAMI

Skan z magazynu wydrukowany na kartce A4 w skali 1:1 nie nadawał się do jakiegokolwiek użytku. Obrazki te mają zbyt mały kontrast i są zszarzałe. Wykonanie samodzielnie płytki na ich podstawie metodą termotransferu byłoby stratą czasu i materiałów. W takim przypadku postanowiłem wrócić do sprawdzonych metod projektowych, czyli ołówka, papieru i nożyczek. Wydrukowałem obie matryce płytek na kalce technicznej. Po nałożeniu obu wydruków na siebie okazało się, że każdy z nich jest w innej skali oraz posiada zniekształcenia geometryczne, w wyniku czego, tylko niektóre przelotki i punkty lutownicze się pokrywają. No dobrze, skoro mam te obrazki w formie zdigitalizowanej, w postaci pliku rastrowego o średniej



KOLOROWA SKŁADANKA

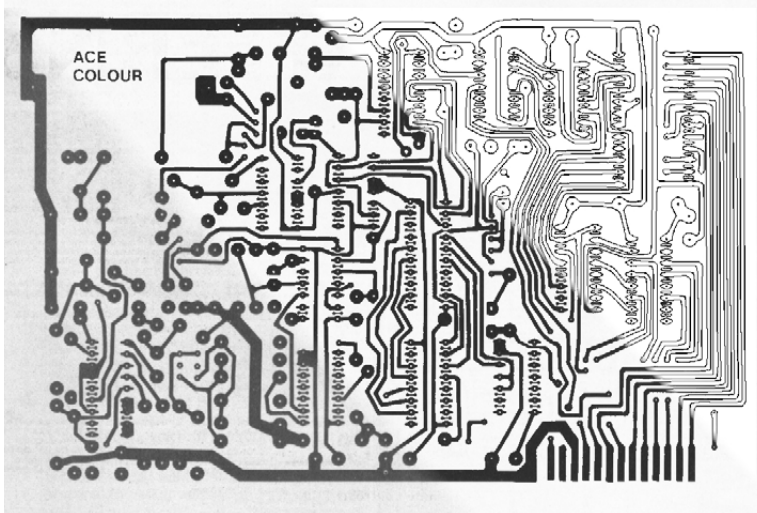
rozdzielczości, mogę nimi operować do woli. Z racji tego, że znam PhotoShopa i Gimpa, oba obrazki przekształciłem do wersji czarno-białej z ostrym przejściem pomiędzy bielą a czernią.

Oba obrazki w tej formie zapisałem na dysku, tak na wszelki wypadek. Następnie stworzyłem nowy rysunek, w którym obrazu spodu płytki użyłem jako obrazu podstawowego (po uprzednim jego odbiciu lustrzanym, tak jakbym patrzył na mozaikę z góry przez szklaną płytkę), na który nałożyłem drugi obrazek górnej mozaiki ścieżek, w formie dodatkowej warstwy na obrazku. Na górnej warstwie kolor tła (biały) oznaczyłem jako przezroczysty, a kolor ścieżek (czarny), zamieniłem na inny (w tym przypadku czerwony) o przezroczystości ok 75%. Górną warstwę potraktowałem narzędziem do korekcji perspektywy, powyciągałem za rogi tak, aby pola lutownicze, oraz przewierty pasowały pomiędzy sobą i taką formę obrazka zapisałem na dysku. Obrazek z kolorowymi warstwami po wydrukowaniu idealnie nadawał się do celów porównawczych ze schematem, do wyłapywania zwarć w skanach i do dalszych prac nad projektem.

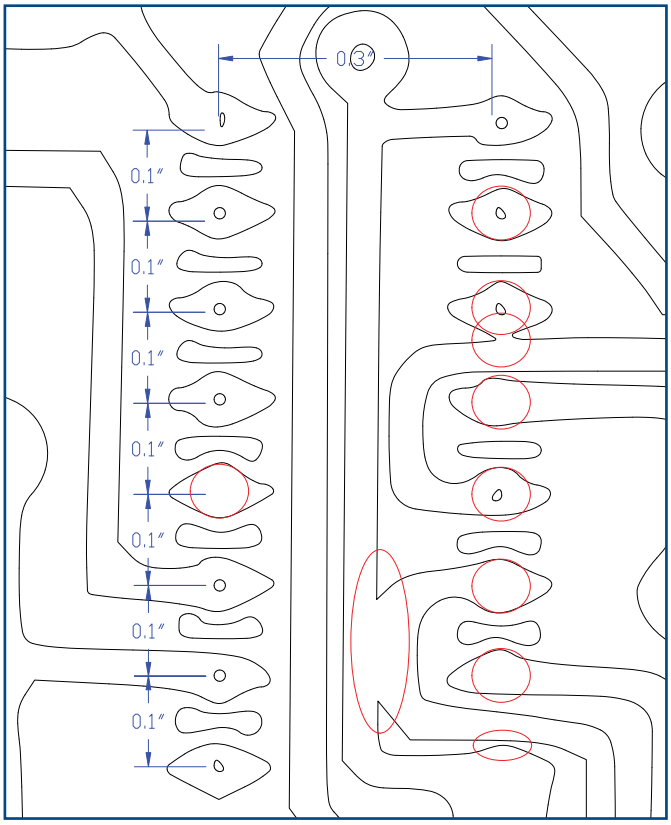
Praca z tak prostymi plikami graficznymi nie jest procesem bardzo czasochłonnym i w jedną godzinę można osiągnąć rezultaty więcej niż zadowalające. Oczywiście jeśli ktoś jest biegły w programach graficznych, taka operacja zajmuje mu mniej niż 15 minut i tyle mi ona zajęła. Nadal miałem tylko plik rastrowy, a do produkcji PCB potrzebne były pliki wektorowe, oddzielne dla każdej strony płytki, dla warstw miedzi, warstw maski, napisów, plik dla maszyny CNC z mapą wierceń i średnicami otworów, oraz plik z obrysem kształtu płytki, tzw. gerbery. W najprostszej postaci dla płytki dwustronnej potrzebne jest 8 plików.

AUTO-TRACER?

Ktoś może zapytać, po co tyle zachodu, skoro można dowolne skany i obrazki przepuścić przez automatyczne wyszukiwanie krawędzi lub narzędzie typu autotracer i od razu otrzymać plik wektorowy



OD SKANU PO WEKTORY



PLIK WEKTOROWY Z POMYŁKAMI AUTOTRACERA

w formacie DXF lub podobnym. Posiadając pliki wektorowe można by innym narzędziem przekonwertować je wprost do gerberów, tylko dla dwóch warstw miedzi i wysłać do produkcji.

Cała zabawa zajęłaby nie więcej niż 30 minut. Niby genialne w swojej prostocie, jednak obarczone dużą dozą niepewności i niedokładności w końcowym produkcie. Autotracery sprawdzają się świetnie, są szybkie, jednak późniejsza edycja setek czy tysięcy drobnych wektorów w programach CADowskich zajmuje więcej czasu, niż stworzenie projektu od podstaw, włączając przerysowanie schematu.

Na przykładowym pliku z automatu, oznaczyłem kilka podstawowych błędów i dla jednego układu scalonego jest ich całkiem sporo. Część z nich jest bardzo łatwa do wychwycenia, np. brak przewiertów czy zwarcia pomiędzy sąsiadującymi ścieżkami. Inne wymagają więcej uwagi i sprawdzenia położenia otworów montażowych. Większość otworów nie ma kształtu koła, są one wielokątami lub elipsami. Odstępy między poszczególnymi pinami układu nie zostały zachowane oraz nie są ułożone w osiach.

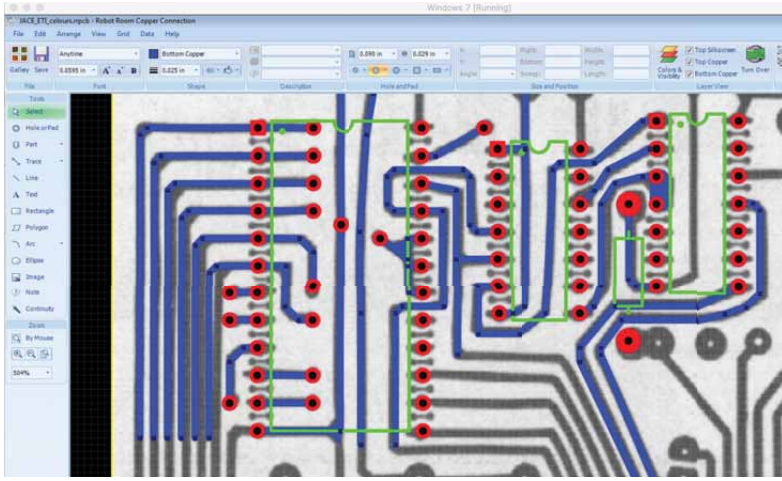
W takim podejściu trudno jest też sprawdzić zachowanie minimalnych odstępów między ścieżkami, wymaganych przez firmy produkujące płytki i ten proces trzeba wykonać „na piechotę".

MALUJEMY MOZAIKĘ

Z pomocą przychodzą nam dedykowane programy do „malowania" płytek. Co prawda nie wszystkie pakiety programowe pozwalają na dowolne tworzenie mozaiki bez schematu, ale są takie – jak na przykład dobrze znany *DipTrace*. Osobiście nie jestem fanem tego pakietu, pomimo jego niezaprzeczalnych zalet. Na co dzień w projektach hobbystycznych używam *KiCAD*a. Jestem z „klanu ogryzka" i filozofia obsługi programów z Windowsa pod Wine jest dla mnie niewygodna. Z mojego doświadczenia ograniczeniem *DipTrace* jest brak możliwości prowadzenia ścieżek pod dowolnym kątem. Ścieżki w *DipTrace* mogą być łamane tylko pod kątem 45°. Co prawda można dodawać do ścieżek łuki, ale to nie było to, czego szukałem.

Istnieją inne pakiety, które pozwalają na pełną dowolność w tym względzie, posiadają biblioteki typowych elementów elektronicznych jak rezystory, kondensatory, złącza, przelotki, podstawki pod układy scalone itp. Pakiety te dbają o ciągłość ścieżek i generują pliki produkcyjne. Jednym z takich pakietów jest *Robot Room Copper Connection*. Jest to pełnowymiarowy program typu CAD, dedykowany do malowania mozaik. Jego poprzednia wersja była dość lekka (ok. 4.5MB na dysku) i jej właśnie używam. W ostatnim czasie program został wykupiony i powiązany z jedną z fabryk obwodów drukowanych na Dalekim Wschodzie. Używam starszej wersji tego programu w środowisku maszyny wirtualnej z Windows 7. Obsługi programu nauczyłem się w jeden wieczór. Program jest bardzo intuicyjny i w sumie prowadzi użytkownika za rękę.

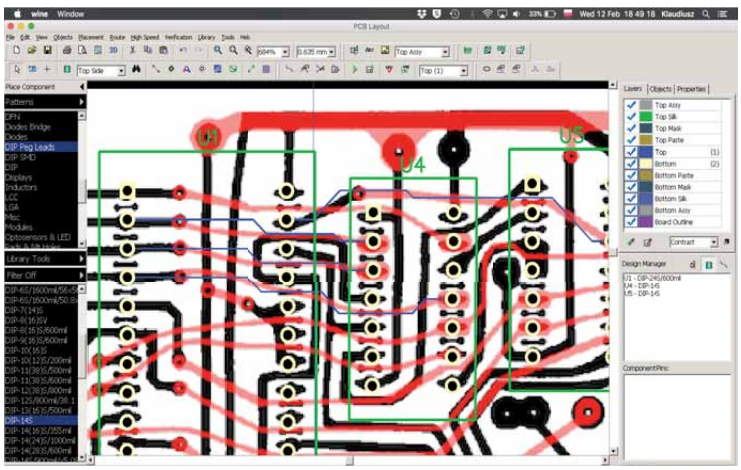
Co robimy w takim programie aby wyprodukować płytkę o jak najlepszej jakości? Po pierwsze otwieramy nowy projekt i jako tło



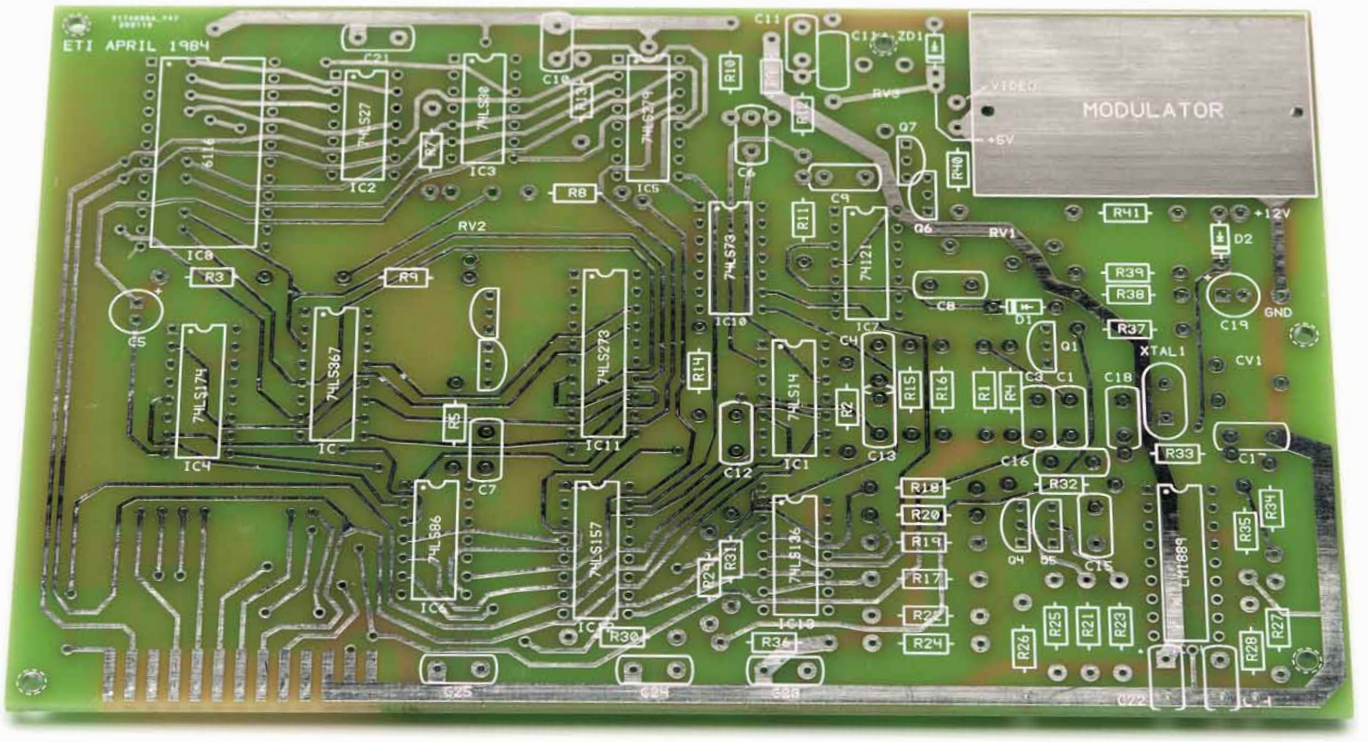
OBSZAR ROBOCZY W COPPER CONNECTIONS

podpinamy obrazek z mozaiką ścieżek, które chcielibyśmy wymalować np. na górnej warstwie PCB. Na podpięty obraz wrzucamy dwa, trzy komponenty typu podstawki pod układy scalone (im więcej nóżek i im szersze tym lepiej) i przeskalowujemy podkład tak, aby otwory pod nóżki zgadzały się z tymi w podstawkach. Najwygodniej zrobić skalowanie podkładu dla dwóch podstawek umieszczonych po przekątnej płytki. Jeśli ten etap mamy za sobą, rozmieszczamy przelotki, dobieramy ich średnice i średnice przewiertów. Dodajemy z biblioteki elementy takie jak rezystory, kondensatory, tranzystory i cokolwiek innego, co jest nam potrzebne. Po ułożeniu ich w docelowych miejscach oraz opisaniu zgodnie z ich przeznaczeniem, zaczynamy „klikanie" ścieżek od punktu do punktu, z zachowaniem oryginalnego ich przebiegu, kształtu i szerokości.

Program zadba za nas o ciągłość ścieżek, możemy dodawać i usuwać punkty załamań, punkty pośrednie lub zmieniać szerokość ścieżki w locie. Po zakończeniu jednej warstwy, powtarzamy proces dla drugiej, w tym wypadku dla spodniej warstwy miedzi. Na koniec zostaje dodać warstwę z obrysem płytki oraz informacje o tym, że jest to replika, a nie produkt



BITMAPA W DIPTRACE



GOTOWA PŁYTKA Z FABRYKI

oryginalny. To, czy ta ostatnia kwestia jest istotna, pozostawiam do rozstrzygnięcia czytelnikowi.

W taki sposób przerysowałem projekt z ETI w kilka wieczorów. Praca żmudna, ale odstresowująca.

CZAS NA SPRAWDZENIE

Po zakończeniu malowania należy projekt odłożyć na półkę na przynajmniej 24h i zająć się czymś innym. Osobiscie polecam hodowlę jedwabników albo obejrzenie tureckiego serialu, aby nasz mózg zapomniał o ścieżkach i warstwach. Nie należy spieszyć się z zamawianiem produkcji. Lepiej trzy razy sprawdzić rzeczy oczywiste, aby wyłapać jak najwięcej błędów, które mogą okazać się trudne do usunięcia przy uruchomieniu układu.

Copper Connections posiada kilka wbudowanych i bardzo pomocnych narzędzi do tego celu. Najczęściej używanym przeze mnie jest sprawdzanie ciągłości połączeń oraz czy aby na pewno do danej ścieżki nie podłączyłem jeszcze czegoś, co tam nie powinno być. Przy przerysowywaniu interfejsu ETI „zwarły" mi się w kilkunastu miejscach masa i linia +5V. W ferworze walki nie obrysowałem złącza krawędziowego na spodniej stronie płytki, w wyniku czego fabryka pokryłaby je maską. Warto sprawdzić też warstwę opisu elementów, czy są one poprawne, co ułatwia późniejszy montaż. W pakiecie są dostępne zautomatyzowane narzędzia, pokazujące miejsca zbliżeń ścieżek na mniejsze odległości niż wymagania procesu produkcyjnego, przewierty nie będące w osiach, punkty lutownicze bez podłączeń itp.

PRODUKCJA I URUCHOMIENIE

W jaki sposób i gdzie zamawiać płytki, nie jest istotne z punktu widzenia tego artykułu. Gerbery wygenerowane z *Copper Connection* wysłałem do fabryki w Chinach, zapłaciłem i czekałem na przesyłkę. W tym czasie mogłem skompletować brakujące elementy i nerwowo obgryzać paznokcie, patrząc na postęp produkcji i dostawy. W końcu listonosz przyniósł pudełko z „niespodzianką" i gdy je otwierałem moja radość była dużo większa, niż przy rozpakowywaniu prezentu spod choinki. Na widok kawałka laminatu cieszyłem się jest dziecko z nowego resoraka czy współcześnie smartfona. Ręce same zaczęły pocić się i drżeć, tętno przyspieszyło, serce waliło jak w stanie przedzawałowym, bo dostałem swoje dzieło życia. Rzuciłem wszystko co robiłem, rodzina zaczęła utyskiwać, że tracę kontakt z rzeczywistością, zaniedbałem sprawy codzienne, ponieważ żądza „spawania" była tak wielka, jak zew krwi u rekina, który wyczuł ranną fokę w wodzie. Jednak nadal należy zachować zimną krew, a lutownicę odłożyć na bok, bo na „pieczyste" przyjdzie czas. Na początek wziąłem lupę w rękę i sprawdziłem obie strony płytki, czy przypadkiem nie strzeliłem samobója. Następnie poszedł w ruch omomierz i sprawdziłem ciągłość linii zasilających i masy, a także, czy nie ma zwarć między nimi.

Wszystko wydawało się poprawne, nadszedł więc czas, na to co tygrysy lubią najbardziej (Tygrys, wybacz), czyli „smarkanie" elementów. Przy takich eksperymentach zaczynam od wlutowania i uruchomienia sekcji zasilającej, o ile taka jest. W moim projekcie była ona bardzo ograniczona i zacząłem od zasilania +12V, oraz zasilania +5V części analogowej. Nie miałem problemów z odpaleniem obu.

Następnie wlutowałem pozostałe elementy bierne, tranzystory, podstawki pod układy scalone i złącze krawędziowe. Tak na marginesie uważam, że jeśli ktoś ma odrobinę pojęcia o elektronice, nie musi trzymać się elementów w 100% zgodnie z listą materiałową zamieszczoną w tym artykule. W przypadku tego interfejsu projektanci założyli, że kondensatory odsprzęgające będą miały pojemności: 10 nF, 100 nF, 220 nF i 470 nF, przy czym te ostatnie powinny być spolaryzowane i tantalowe. W tej kwestii poszedłem na unifikację i wszędzie zainstalowałem kondensatory ceramiczne o pojemności 220 nF.

Przyszedł czas na podłączenie całości (bez układów logicznych) do komputera. Na szczęście nic nie wybuchło, nie zapaliło się, strat w sprzęcie i ludziach brak, a komputer działa poprawnie, tak jak działał poprzednio. Na tym etapie można go wyłączyć i zamontować wszystkie układy logiczne. Po ponownym podaniu zasilania do komputera i karty ETI organoleptycznie sprawdziłem, czy przypadkiem któreś z elementów nie grzeją się nadmiernie. Pozostało przystąpić do uruchomienia i zestrojenia układu zgodnie z procedurą zamieszczoną w artykule, co też uczyniłem.

Z uruchomieniem było trochę pracy, ale w ostatecznym rozrachunku układ ruszył, na ekranie zobaczyłem zielony kursor na czarnym tle. Wpisałem krótki program testowy, generujący kolorowe paski na ekranie et voilà, wszystko działa jak należy. „Panie Prezydencie! Melduję wykonania zadania!"

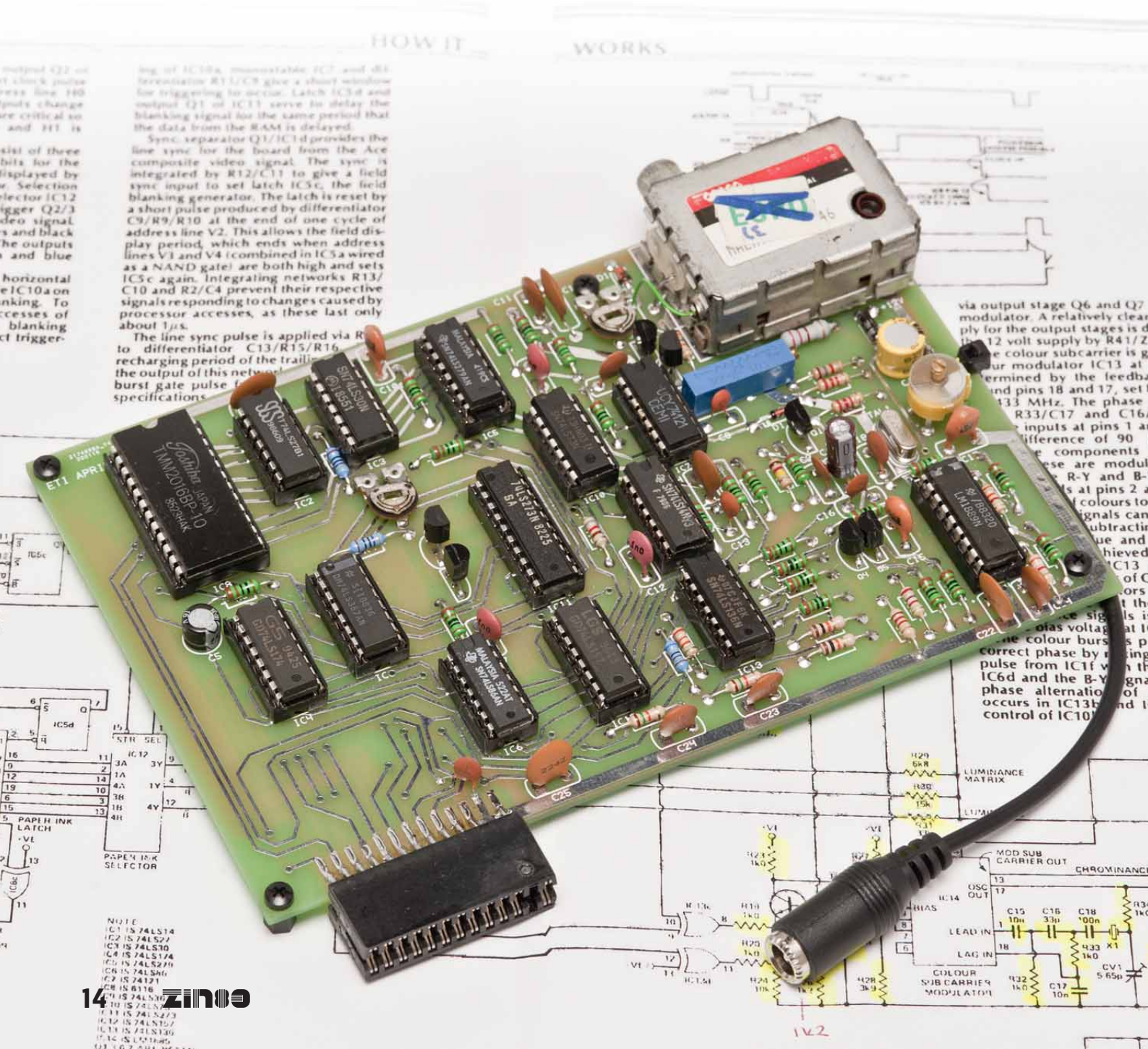
CZASOPRZESTRZEŃ I PODSUMOWANIE

Przerysowanie projektu ETI Colour zajęło mi kilka wieczorów, ale w ten czas wliczyłem też czas, który poświęciłem na opisy elementów, sprawdzenie ciągłości ścieżek i zwarć w newralgicznych miejscach, jak linie zasilające, czy pola masy.

Dodatkowo jeden wieczór spędziłem na przygotowaniu plików rastrowych i studiowaniu po raz setny schematu oraz jeszcze jeden na oglądanie i porównywanie gerberów z moim dwukolorowym rysunkiem ścieżek. Błędy? Jakie błędy?! Owszem, nie uniknąłem kilku potknięć. Na przykład w opisie rezystorów na płytce dwa razy występuje R9 i brakuje R6, jeden z opisów niefortunnie umieściłem na ścieżce pokrytej cyną i jest nieczytelny. W innym miejscu jeden z kondensatorów mogłem umieścić odrobinę inaczej niż w oryginale oraz z innym rozstawem nóżek. W następnej wersji poprawki wymaga rozstaw otworów pod potencjometr montażowy i trymer foliowy. Spraw kosmetycznych jest więcej, ale one nie wpływają na pracę układu. Sam proces rysowania i rekonstrukcji jest ciekawym doświadczeniem, ale należy mieć na uwadze, że nie jest on zabawą z cyklu „narysuj – zamów - pospawaj – włącz i działa''.

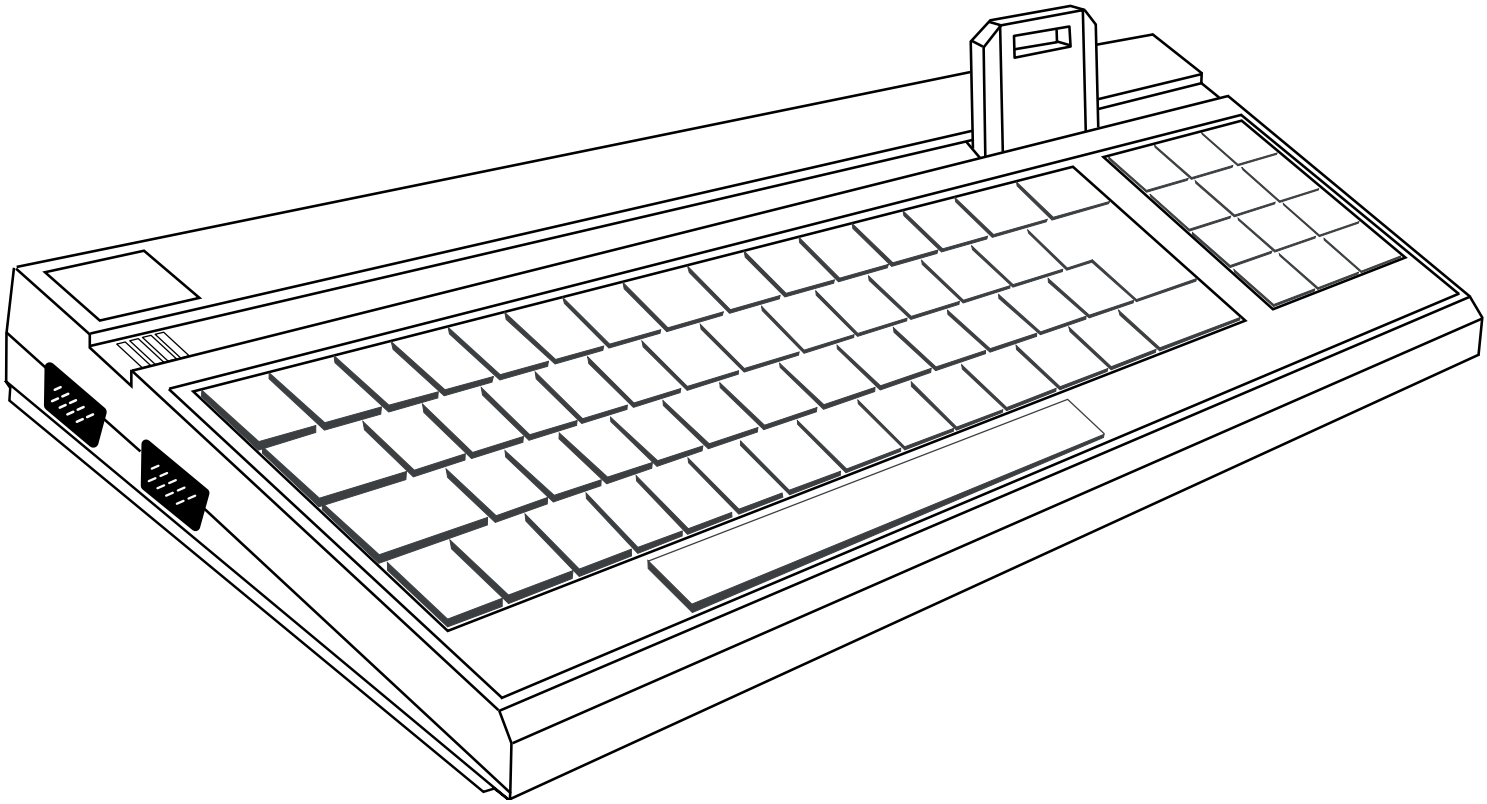
Zbudowanie tego układu od początku wymaga sporej dozy samozaparcia i samokontroli, ponieważ można się zniechęcić i stracić cierpliwość już na samym początku, przy próbie trasowania skanów. Ponadto w schemacie z ETI znalazlem kilka nieścisłości i pomyłek, np. dwa razy występuje kondensator C7 o tej samej pojemności a kondensatora C8 nie ma. Natomiast lista materiałowa specyfikuje kondensator C8 do nastawy impulsu generowanego przez multiwibrator typu 74121. C7 na schemacie ma wartość 1 nF, a C8 powinien być o pojemności 47 nF. Bez odnalezienia odpowiedniej karty katalogowej układu scalonego i zgrubnego przeliczenia wymaganej pojemnosci ani rusz. Nie wszystkie wartości elementów na schemacie były czytelne. Pracy i przeszkód jest wiele, zanim dotrzemy do celu. ■

Uwaga dla czytelnika: mój artykuł nie miał na celu nauki rysowania ścieżek ani nauki konkretnego programu, ponieważ* Copper Connections *nie jest jedynym pakietem, który można znaleźć w Internecie. Pracując z leciwymi dokumentami i ich skanami czasem trzeba szukać źródeł w różnych miejscach, weryfikować nieczytelne partie dokumentacji i przy okazji dobrze się bawić. Ze swojej strony życzę wszystkim tyle samo owocnych uruchomień, co zaprojektowanych lub odtworzonych układów.



TIMEX COMPUTER 3256

KOMPUTER-WIDMO. NIEGDYŚ MAŁO KTO O NIM SŁYSZAŁ, JESZCZE MNIEJ GO WIDZIAŁO NA OCZY, A OFICJALNIE WRĘCZ NIGDY NIE POWSTAŁ. DOPIERO W EPOCE INTERNETU I MEDIÓW SPOŁECZNOŚCIOWYCH, SKĄPE INFORMACJE NA JEGO TEMAT ZACZĘŁY OBIEGAĆ ŚWIAT, WRAZ ZE SKANAMI ZE STARYCH, PORTUGALSKICH GAZET.



JT

Choć dziś trudno w to uwierzyć, pod koniec lat 80. portugalski Timex planował wypuścić na rynek kolejny model 8-bitowego komputera. Choć już wtedy na rynku gościły Atari ST oraz Amiga, coraz powszechniejsze stawały się też komputery PC – jednym słowem, świat parł ku 16 (i więcej) bitom. Tyle, że nie każdego było wtedy stać na sprzęt szesnastobitowy – tak w Polsce, jak i w niezbyt bogatej wówczas Portugalii. I nie chodzi wyłącznie o zasobność portfela przeciętnego obywatela, ale również o budżety, jakimi dysponowały instytucje edukacyjne – a to głównie o nie TC3256 miał stoczyć bój. W sumie znamy to dobrze z naszego podwórka, gdy w latach osiemdziesiątych polska edukacja walczyła o własny model komputera dla szkół, zakończony umiarkowanym, lecz jednak sukcesem Elwro 800 Junior.

Co wiadomo?

Niewiele. I w sumie żadnej z informacji nie można do końca ufać, bo cała sytuacja bardziej wygląda na studium wykonalności, przerwane w momencie definitywnego zamknięcia produkcji komputerów w zakładach w Caparice pod Lizboną. Pierwsze wzmianki o TC3256 zaczęły pojawiać się w 1987 roku, ale termin premiery nie był ustalony i mógł to być nawet – cytując wikipedię – rok 1989. Umieszczony w jednej z komputerowych gazet szkic wyraźnie wskazuje, że bazą projektową miał być Timex Terminal 3000, wnoszący w wianie obudowę z solidną, 69-przyciskową klawiaturą. Największą rzucającą się w oczy zmianą (w stosunku do TT3000) jest gniado standardowych, timexowych kartridży, znanych z modeli TS/TC2068, które zamiast pod klapką, jak to miało miejsce w starszych Timexach, znalazło się na górze, powyżej klawiatury – podobnie, jak ma to miejsce np. w komputerach MSX.

Lewy bok komputera gości znane z TC2048 gniazdo joysticka w standardzie Kempston oraz drugie złącze 9-pinowe – najprawdopodobniej port szeregowy RS232, o którego obecności wspominają rozmaite publikacje. Na przeciwległym boku umieszczono – dokładnie jak w terminalu TT3000 – 15-bolcowe złącze dla stacji FDD3000. Tylna ścianka komputera zawierać miała wszystkie złącza audio/wideo (w tym wyjście RGB), złącze krawędziowe oraz gniazdo zasilania. Tam też prawdopodobnie znajdowałoby się gniazdo sieci edukacyjnej TENET (Timex Educational Network), o której mało wiadomo, bo… sieć ta najwyraźniej skończyła swój żywot na etapie prototypowania. Wygląda jednak na typową pętlę prądową, czyli rozwiązanie podobne do oferowanego przez np. ZX Interface 1.

SOFTFILE

NOTÍCIAS

TIMEX - NOVOS PRODUTOS

Quem pensar que a TIMEX (TMX PORTUGAL, LTD) parou ou está com dificuldades no campo da criação enganou-se e redondamente.

Segundo nos foi dado observar e comunicado os produtos em questão o TC 3256, 256K Personal Computer, e o gravador estarão à público proximamente.

Embora já operacionais a sua saída está atrasada pela dificuldade na aquisição de determinados componentes no mercado internacional (prazos de entrega extremamente longos) o que provoca, obviamente, atrasos no arranque de produção.

O novo computador TIMEX TC 3256 é um personal computer de excelente concepção e características inovadoras.

A nova máquina da TIMEX possui à partida 256K RAM podendo ser expansível até 1 Mbyte. A CPU é um Z80A.

Ao ligá-lo o utilizador tem à sua frente, no ecrã, cinco opções que definem a sua estrutura base: BASIC, TIMEWORD+, CP/M, TERMINAL, DISK START e CARTRIDGE.

O «Basic» é o Basic da linha TIMEX (SPECTRUM compatível) mas agora com novos comandos. O «TIMEWORD+» é um processador de texto que, sinceramente, quem conhece o WORDSTAR ficará confuso ao verificar a sua similaridade e ainda com a possibilidade de inclusão de screens no próprio texto. A opção «CP/M Terminal» para utilização com o sistema FDD 3000 e consequentemente em CP/M. «DISK START» é a opção com maior interesse e inovadora porquanto não é oferecida aos utilizadores em nenhum micro do mercado

ormática

JORNAL DE NOTICIAS 14.MAIO

COM O NOVO TC 3256

TIMEX APOSTA NAS LIGAÇÕES EM REDE

Disponível no mercado português dentro de alguns dias, o TC 3256, novo computador com a «etiqueta» TIMEX, embora por enquanto esteja sob a forma de protótipo, começa já a tumultuar a imprensa da especialidade, sempre interessada nas novidades.

A máquina em causa, possuidora de um conjunto de características, em muitos aspectos, inusitadas, é dotada essencialmente pelos seus 256 Kb de RAM, dos quais 208 Kb utilizáveis como RAMDISK, pela ROM de 64 Kb, e pelos «slots», «bus» de expansão e várias fichas para conexão de periféricos, que permitem a sua utilização em pequenas e médias empresas, e lazem dela uma máquina com grande extensa como «coração» de ciclos de trabalho ligados às mais diversas áreas (aqui pensamos, por exemplo, nas suas fortes possibilidades de utilização ao nível do projecto MINERVA).

Mas esperos, do porto, o que é o TC 3256.

Surpresa imediata quando apetrechos de diversos elementos da embalagem com que a adquirimos é, sem dúvida, a fonte de alimentação do computador que, quase duas vezes maior que o arrojo transformador do «Spectrum», possui um cabo com vários condutores e uma ficha de ligação tipo DIN, idêntica à muito diferentes, portanto, daquelas a que, para as mesmas funções, estávamos habituados.

Razão para tal facto reside no desejo de eliminar o conversor de função e o regulador de 5 Volts presentes na placa do ZX, e na desmultiplicação de tudo o efectuado dentro do módulo do próprio transformador, o que, para além de outras vantagens (a maior parte das avanços do «Spectrum» envolve o conversor), evita o excessivo calor após algumas horas de utilização, como resultado da dissipação da voltagem excedente, proporcionada pelo anterior transformador no processo de alimentação dos circuitos.

• Máquina muito versátil

As novidades, contudo, não se limitam à fonte de alimentação.

Mesmo desligado, o TC apresenta-se dotado de muito mais do que no mercado dos competidores «Spectrum» não é idêntico.

Com efeito, se deslocarmos o olhar pensamento e superfície da caixa que rodeia o teclado (talvez o melhor de todos os adoptados pelos computadores do idêntico porte), facilmente nos apercebemos de que esta mais parente uma máquina que é, sem dúvida, extremamente versátil, como consequência das diversas interligações previstas.

Numa breve referência a algumas destas ligações podemos citar como exemplo: uma entrada para «joysticks», uma entrada/saída para FDD (desta vez sem necessidade de equipar à pequena caixa prata que se ligava directamente ao «bus» de expansão, tirotando-esta para outras interligações simultâneas), o «bus» de expansão (que embora não sendo 100% compatível com o do «Spectrum» contém muitos dos sinais dados nas mesmas posições, o que permite a utilização de uma vasta gama de periféricos), uma «slot» para «cartridge» que possibilita a utilização de software em EPROM, uma entrada/saída TTL para comando e controlo de circuitos externos, uma entrada/saída RS 232 para interligação de «modems», impressoras e outros periféricos, uma saída para monitor RGB, uma saída UHF para TV (canal 36), e os usuais EAR e MIC para utilização da cassete como memória de massa.

Nas ligações possíveis, o TC 3256 prevê ainda a in loco - ligação em sistema de rede de 25 unidades (para uma distância máxima de 100 metros, através do «Timex Educational NETwork» (TENET).

Este sistema de trabalho que actualmente se julga como o sistema que irá ser adoptado em grande escala nos próximos anos, não só ao nível empresarial mas também em utilizações domésticas, é, assim, proposto pela TIMEX portuguesa que encontrará aqui o ponto mais forte da máquina que agora apresentou, quando a vir submetida ao julgamento dos seus utilizadores.

• Múltiplas características

A descrição do novo produto TIMEX, desta vez totalmente idealizado e concebido por técnicos portugueses, é, como seria oportunidade de constatar-se, de longe que ele essas nas lojas, uma tarefa difícil devido aos muitos aspectos difundidos que se mostram bastante interessantes.

Quase em estadiça, características referentes - para além das que já referimos - são as possibilidades de funcionamento como terminal CP/M, como computador BASIC compatível «Spectrum», e como processador de texto (utilizando o TIMEWORD +, uma versão melhorada do TIMEWORD actualmente comercializado para o TC 2068), e a opção de inicializar o sistema a partir do disco de memória ou do «cartridge». Tudo isto selectivamente com a presença de uma tecla no menu principal inicial.

Quando se encontra em modo «Spectrum», o novo TC mantém activas todas as instruções do BASIC Sinclair, adicionando-lhe ainda um sub-set de instruções, seleccionadas através de uma sintaxe compatível pelas instruções antigas e alguns símbolos de um único caracter, antes não-utilizados ou utilizados em contexto diferente por exemplo, [!], [↑], [&].

Através deste conjunto de instruções complementares, que no manual do utilizador nos surge com a designação de TEC (=Timex Extended Command-), o TC 3256 permite o controlo de dispositivos que Sir Clive Sinclair nunca pensou utilizar como base no dialecto de BASIC por ele criado.

Nestes «novos» comandos encontramos, por exemplo, a figura destinada a trabalhar no modo de resolução dupla (512 x 192 «picture elements» em gráficos, 64 colunas de texto), a comandar o PSG AY-3-8912 (gerador de sons programável de 3 canais), ou a utilizar a RS 232 como canal de comunicações.

Computador que pretende atingir um público que se almeja, agora, a definir como mais exigente e mais apreciador das potencialidades deste tipo de máquinas, fugindo à sua mera utilização como fonte de jogos vídeo, isto documente do «Spectrum» quase não se mostrasse como tal numa comparação com o seu antepassado remoto.

Por isso podemos dizer que o TC 3256 surge numa época diferente, procurando satisfazer necessidades diversas.

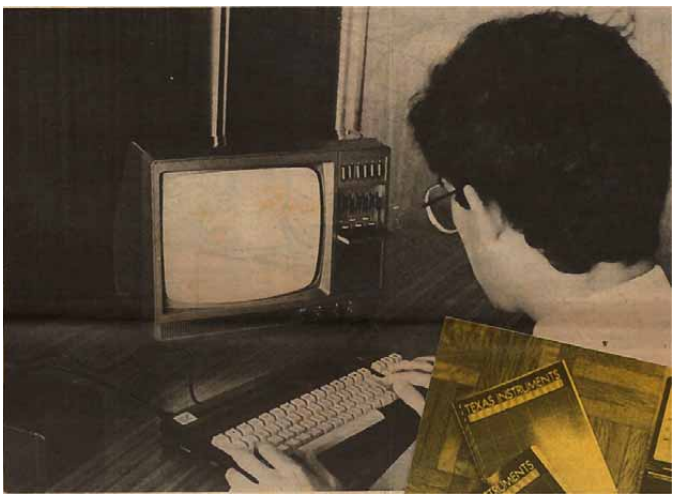
CONFERÊNCIA IBÉRICA DE COMUNICAÇÃO DE DADOS

ARTYKUŁY W PRASIE PORTUGALSKIEJ Z ZAPOWIEDZIĄ TC3256

Coś nowego, coś starego

Co najbardziej rzuca się w oczy, to 256KB pamięci RAM, którą można rozszerzyć do 1 megabajta. ROM też urósł – jest go aż 64KB – zorganizowany w 4 bloki po 16KB o zupełnie różnych funkcjonalnościach: oprócz standardowego Sinclair BASICa, mamy do wyboru 80-kolumnowy edytor tekstu Timeword+ (notabene przeniesiony żywcem z kartridża dla TC2068) albo też terminal CP/M (wymagający oczywiście podłączenia stacji FDD3000). Ostatni blok zajmuje rozszerzenie BASICa obsługujące RAM-dysk, sieć TENET, RS232 oraz układ AY-3-8912. Sieć TENET miała być prawdopodobnie obsługiwana przez dedykowany mikrokontroler (prototyp kontrolera sieciowego zbudowano na układzie Intel 8031) i oferować konfigurację do 25 połączonych komputerów na dystansie do 100 metrów. Oprócz współdzielenia zasobów (np. stacji dysków), możliwa miała być wymiana komunikatów między użytkownikami, a wszystko przy użyciu nowych komend BASICa. Dodatkowe urządzenia np. drukarki można by było podłączać przez port szeregowy o prędkości 300-9600 bodów, który miał zostać zrealizowany poprzez bit banging, czyli programowo – prawdopodobnie tak samo, jak w Spectrum 128K – poprzez manipulację liniami portu wejścia/wyjścia układu AY. Nie wiadomo niestety, czy sam układ AY miał działać pod adresami zgodnymi ze Spectrum 128K, czy też pod tymi znanymi z TS/TC2068. Jednak implementacja gniazda joysticka w standardzie Kempston pozwala przypuszczać, że projektanci nowego komputera wysnuli wnioski z popełnionych niegdyś błędów i baczniej przyglądali się, co (i w jaki sposób) w nowszych modelach Spectrum piszczy.

W celu zachowania zgodności z ZX Spectrum, TC3256 miał być z założenia kompatybilny z najbardziej udanym z modeli – czyli z TC2048. Nic zatem dziwnego, że oprócz taktowanego identycznym zegarem (3,5 MHz) procesora Z80, odziedziczyłby on po starszym bracie również układ SCLD, czyli ekwiwalent znanego z ZX Spectrum układu ULA. I to najwyraźniej bez żadnych zmian – nie pojawiły się bowiem żadne nowe możliwości graficzne. Z jednej strony to trochę rozczarowuje, zwłaszcza patrząc na to, co raptem kilka miesięcy później zaprezentował Sam Coupé. Z drugiej jednak strony uspokaja – ten układ SCLD jest naprawdę dobry, a do tego niezawodny. Trzeba się jednak pogodzić z tym, że początek ramki obrazu przypada na inny takt, niż w oryginalnych modelach Spectrum 48K, co niesie ze sobą określone konsekwencje w przypadku oglądania dem lub grania w niektóre gry, np. z efektami na borderze.



„MICROSETE", MAJ 1987

Trochę gdybania

Kontynuując temat SCLD, warto przyjrzeć się bliżej jego możliwościom zarządzania pamięcią, które tylko częściowo zostały wykorzystane w modelu 2068, a praktycznie wcale w 2048. Prawie na pewno mechanizm zawarty w SCLD zostałby zastosowany w TC3256 – choć to nie wszystko, bo 256 kilobajtów to znacznie powyżej możliwości przewidzianych przez konstruktorów układu w 1983 roku.

Do zarządzania pamięcią Timex używa dwóch portów, 244 ($F4) oraz 255 ($FF), choć z tego drugiego wykorzystywany jest tylko jeden, najstarszy bit. Służy on do przełączania pamięci ROM w najniższej ćwiartce przestrzeni adresowej ($0000 - $3FFF, czyli 0 - 16383), ponieważ modele 2068 miały po dwa bloki pamięci ROM – 16KB z Timex BASICem oraz dodatkowo 8KB z obsługą nowych komend (modele te miały też Sinclair BASIC dołączony w formie kartridża – bez niego wiele gier dla Spectrum nie chciało działać). Z kolei port $F4 pozwala „wyłączać" kolejne bloki pamięci o rozmiarze 8KB każdy, przy czym bit zero odpowiada za obszar $0000 - $1FFF (0 - 8191), bit 1 – $2000 - $3FFF (8192 - 16383) i tak dalej, aż do najstarszego bitu, sterującego blokiem

DANE TECHNICZNE

PROCESOR: Z80A
RAM: 256K (208K RAMDRIVE/48K BASE MEMORY)
ROM: 64K
16K SINCLAIR BASIC
16K TIMEWORD WORD PROCESSOR
16K TIMEX EXTENDED BASIC (TENET, DISK, RAMDRIVE)
16K CP/M TERMINAL EMULATOR
TRYB TEKSTOWY : 32X24 / 64X24
TRYB GRAFICZNY: 256X192 / 512X192
DŹWIĘK: BEEP AND AY-3-8192 (CAN OUTPUT TO TV)
DŻOJSTIK: 1 KEMPSON
CARTRIDGE PORT: 1
DISK DRIVE: TOS / CP/M
TV OUTPUT MONITORS:
VIDEO COMPOSITE AND RGB (COLOUR/ MONOCHROME)
COLOURS: 8+BRIGHT=15
KEYBOARD: 69 KEYS, PROFESSIONAL KEYBOARD WITH NUMERIC KEYBOARD AND FUNCTION KEYS (CAPS LOCK, EXTENDED, CURSORS, EDIT DELETE, BREAK)
RS232: 1 (300 TO 9600 BAUD)
MIC AND EAR: MOŻNA UŻYWAĆ MAGNETOFONU

$E000 - $FFFF (57344-65535). W miejsce wyłączonego bloku załączany byłby blok pamięci „zewnętrznej", znajdującej się np. w kartridżu. Jednak TC3256 miał być wyposażony w dużą pamięć wewnętrzną, tak więc odłączanie bloków z pamięci podstawowej (tej, która jest zgodna ze Spectrum 48K) przyłączałoby rozszerzoną pamięć RAM, zorganizowaną w formie 4 banków po 64KB każdy. Wybór jednego z tych 4 banków, jak również wybór jednej z 4 pamięci ROM w obszarze dolnych 16 kilobajtów wymaga minimum 2 dodatkowych linii sterujących (lub czterech dla pojemności 1 MB) – zgaduję, że miały do tego służyć niewykorzystywane linie w porcie układu AY, choć jest to czysta spekulacja. Po prostu takie rozwiązanie byłoby najprostsze. A ponieważ timexowa ULA zapewnia 8-bitowe odświeżanie pamięci DRAM (procesor Z80 dostarcza tylko 7 bitów adresu odświeżania, a dodatkowy, ósmy bit jest zaimplementowany właśnie w SCLD), można było śmiało zastosować dość tanie i łatwo dostępne układy DRAM 44256/414256, czyli o organizacji 256KBit x 4. Pozwalałoby to na zminimalizowanie dodatkowej logiki do sterowania i bankowania tak dużej pamięci. Rozszerzenie pamięci do 1 MB miałoby zapewne formę wewnętrznego modułu, ponieważ szyna pamięci DRAM

PROTOTYP TENET. FOT. JOHNNY RED (TIMEX.JOHNNYRED.ORG)

ma multipleksowane linie adresowe oraz dodatkowe sygnały RAS i CAS, które nie są wyprowadzone na standardowe złącze krawędziowe. Kto wie, możliwe, że na płycie komputera czekałyby podstawki pod dodatkowych 6 kostek DRAM.

Tak duża pamięć RAM służyć miała przede wszystkim jako RAM-dysk o pojemności 208KB, dostępny z BASICa dzięki nowym komendom: LOAD!, SAVE!, itp. To nie wszystko. Przecież używając portu $F4 można odłączyć od procesora całą główną pamięć, wliczając w to pamięć obrazu. Jeśli w to miejsce zamelduje się zewnętrzny (w sensie – z puli pamięci dodatkowej) bank 64KB – dostajemy do rąk komputer, będący niczym puste płótno dla malarza – bez żadnych sprzętowych ograniczeń w całym obszarze adresowym pamięci. To doskonałe pole doświadczalne dla programistów, chcących stworzyć np. nowy system operacyjny lub język programowania. Specjalna procedura w jednym z dodatkowych ROMów miała umożliwiać załadowanie do tej pamięci pliku z dyskietki.

Jeszcze jeden szczegół techniczny, wynikający z odziedziczonej po poprzednikach architektury komputera. Ze względu na podział RAM na blok pamięci contended (obszar 16KB z pamięcią obrazu) i uncontended (pozostała pamięć), zachowany musiał zostać odrębny blok 16KB, obsługiwany bezpośrednio przez SCLD. Można więc zakładać, że łączna pamięć RAM wynosiłaby 272KB (256 + 16). Zastosowanie starego układu zablokowało też możliwość zastosowania szybszego mikroprocesora, a szkoda, bo operacje na pamięci obrazu w wyższej rozdzielczości (512x192) – a zwłaszcza obsługa trybu 64 lub 80 kolumn – są bardzo pracochłonne, co wyraźnie widać po czasie „rysowania" tekstu na ekranie.

Ktokolwiek widział, ktokolwiek wie

Przy projektowaniu TC3256 portugalski Timex dość inteligentnie korzystał z doświadczeń nabranych na obu rynkach – rodzimym oraz polskim. Wsłuchiwał się również w potrzeby placówek oświatowych. Stąd nowy model miał być wzbogacony o to, czego początkowo brakowało ubogim krewnym z Wysp Brytyjskich – mamy więc solidną klawiaturę, wyjście monitorowe RGB, możliwość łączenia w sieć, współdzielenia stacji dysków itp. Jednak największym atutem miała być uniwersalność w doborze konfiguracji ROM i RAM, uzupełniona wysokiej jakości oprogramowaniem (które już częściowo istniało dla TC 2068). Pomysł dobry, choć podpatrzony u innych producentów – stronicowanie RAM, przełączanie ROM-ów lub ich całkowita dezaktywacja istniały wówczas w Amstradach, rozmaitych modelach MSX, a nawet w ostatnich modelach ZX Spectrum 128K +2A/B i +3. Niestety pomysł ten nie doczekał się realizacji w wykonaniu Timexa, ponieważ produkcja została zamknięta, firma zaś powróciła do korzeni – czyli do produkcji budzików. Tak więc TC3256 zaistniał wyłącznie jako prototyp, lub prototypy - będące aktualnie w niewiadomo, czyim posiadaniu. Z drugiej strony – nie ma gwarancji, że nowy model odniósłby sukces, bo oferował umiarkowane możliwości graficzne i dźwiękowe, znane w dodatku z wcześniejszych modeli. Nie szykował się zatem jakiś wielki przełom – zwłaszcza, że sercem komputera miał być ten sam procesor (i taktowany taką samą częstotliwością), jaki był wkładany do komputerów Sinclaira od niemal dekady.

Kiedy w 1988 roku kupowałem swój pierwszy własny komputer – Timexa 2048 w Centralnej Składnicy Harcerskiej – miałem świadomość, że wydaję pieniądze na sprzęt przestarzały, jednak na moją decyzję wpłynął cały szereg mniej lub bardziej emocjonalnych czynników. I jeśli rok później trafiłaby się możliwość kupienia TC3256 (i miałbym akurat wolne środki), to też bym go kupił – gdyby tylko nie był komputerem-widmem.

■

MICRODRIVE I JEGO NOWE WCIELENIE - VDRIVE DLA ZX SPECTRUM

MICRODRIVE BYŁ SZTANDAROWYM PRODUKTEM SINCLAIRA PRZEZNACZONYM DLA ZX SPECTRUM – JEDYNĄ, SZYBKĄ PAMIĘCIĄ MASOWĄ PRZEWIDZIANĄ DO ZAPISU I ODCZYTU, AŻ DO MODELU (SYGNOWANEGO JUŻ PRZEZ AMSTRADA PO PRZEJĘCIU FIRMY SINCLAIR W 1986 ROKU) ZX SPECTRUM 128K +3, KTÓRY OTRZYMAŁ NAPĘD DYSKOWY 3".

TOOLOUD

Trochę historii

W momencie wprowadzenia na rynek, Microdrive zdeklasował wydajnością stacje dysków, nie wspominając już o dużo niższym koszcie zakupu samego urządzenia. Przynajmniej tak to wyglądało jeśli spojrzymy na urządzenie przez pryzmat opłacalności.

Napęd Microdrive oraz oraz wymagane rozszerzenie sprzętowe Interface 1 pojawiły się dopiero w połowie 1983 roku, ponad pół roku po oficjalnej premierze ZX Spectrum i pozwalały za niecałe 100 GBP cieszyć się prędkością 120KBitów, czyli ponad 6.25 razy większą niż wprowadzona w podobnym okresie stacja dysków Atari 1050. Kasetki umożliwiały zapisywanie danych 85 - 100KB, czyli mniej więcej tyle, ile pojemność formatowania dyskietki SD (single density – 90k). Ich wadą była oczywiście fizyczna forma – kasetki były jednostronne i jak czas pokazał, nośnik taśmowy rozciągał się, tracąc dane.

Wydaje się, że rynek Microdrive przejął częściowo zgodny programowo system dyskowy Opus, który od 1985 roku był jednym z głównych systemów dyskowych dla ZX Spectrum, obok MGT D+ oraz Beta.

vDrive ZX

Autor vDrive Charles to australijczyk pasjonujący się sprzętem Sinclaira – stworzył i oprogramował kilka współczesnych „zamienników" specjalizowanego układu ULA, pomagających odratować

uszkodzone komputery i peryferia Sinclaira: vLA81, vLA82, vLA128 – odpowiednio dla ZX81, ZX Spectrum 16/48K i ZX Spectrum 128K/128K +2, oraz vLA1 (zamiennik ULA dla Interface 1) oraz vDrive ZX i vDrive QL – współczesne wersje napędów Microdrive korzystające z karty SD jako nośnika obrazów .mdr zawierających dane kasetek Microdrive w postaci cyfrowej.

Komplet zawiera uruchomione urządzenie, które potrzebuje… obudowy. Tutaj mamy dwie drogi – albo szukamy „dawcy" w postaci niedziałającego napędu Microdrive albo… drukujemy dostępny pro-

jekt. Trzecia opcja to zakup wydrukowanej obudowy – i tutaj dobra wiadomość, być może Djordje (RetroRadionics) uruchomi produkcję form do napędów Microdrive.

Demontaż oryginalnego napędu z obudowy niestety, jak w przypadku ZX Spectrum 16/48k w klasycznej obudowie, wymaga podgrzania aluminiowej nakładki z nadrukiem (faceplate) w celu rozpuszczenia kleju. Wtedy ostrożnie możemy zdemontować nakładkę – oczywiście istnieje ryzyko, że ją zegniemy. Potem pozostaje nam odkręcić górną część obudowy i po uzyskaniu dostępu do środka obudowy, zdemontować mocowanie samego napędu. vDrive został zaprojektowany jako część do wymiany na zasadzie 1:1 – jest praktycznie identyczny rozmiarowo, wszystko łącznie z diodą sygnalizującą pracę napędu pasuje idealnie.

Sam napęd po zamontowaniu jest gotowy do pracy poza jednym szczegółem – musimy przygotować sobie kartę SD.

vDrive obsługuje format plików MDR – i pozwala uzyskać 127 albo 126KB pojemności obrazu Microdrive (w zależności od wersji ROM Interface 1). Co istotne – firmware obsługuje obecnie tylko jeden poziom katalogów, więc dla napędu będą widoczne pliki tylko z katalogów znajdujących się w katalogu głównym karty.

Jak obsługiwać vDrive?

Najpierw ładujemy komendą RUN (przy wyjętej karcie SD) zestaw Toolkit, który jest wbudowanym w system obrazem Microdrive.

Po uruchomieniu go wpisujemy komendę:

.SDINIT co pozwoli nam zainicjować kartę SD.

Pozostałe komendy (.MKDIR, .RMDIR, .CD, .LI) odpowiadają typowym polecenio DOSa – utwórz katalog, usuń katalog, przejdź do katalogu, czy wreszcie wyświetl zawartość katalogu.

Jak utworzyć nowy obraz nośnika Microdrive?

Załóżmy, że stworzyliśmy katalog TESTY komendą
.MKDIR ˝TESTY˝
i przeszliśmy do tego katalogu komendą
.CD ˝TESTY˝
Sam obraz tworzymy komendą:
.MKIMG˝TEST1˝
Jeżeli chodzi o nazwy plików, obowiązuje nas limit 8 znaków.
Takie pliki możemy oczywiście usuwać komendą
.RM ˝TEST1˝

Jak przypisać obraz nośnika do napędu?

Finalnym ruchem, gdy mamy już plik – obraz nośnika Microdrive – jest przypisanie go do jednego z ośmiu dostępnych wirtualnych napędów co robimy komendą:
.LD 1 ˝TESTIMG1˝
Jeżeli ta operacja się powiedzie, to komenda
.LV
(czyli list vDrives) pokaże, że napęd nr 1 ma status BLANK, ponieważ obraz jaki stworzyliśmy jest pusty i nie został on sformatowany. W przypadku postępowania z fizycznym oryginalnym napędem i kasetką Microdrive powinniśmy wpisać komendę
FORMAT ˝m˝;1;˝Cart 1˝
ale w przypadku korzystania z vDrive wpisujemy komendę
.F1 ˝Cart 1˝
Jeżeli wpiszemy teraz polecenie CAT albo komendę .LV to napęd pierwszy będzie miał już oznaczenie „Cart 1" zamiast BLANK, co oznacza, że jesteśmy gotowi do używania go.

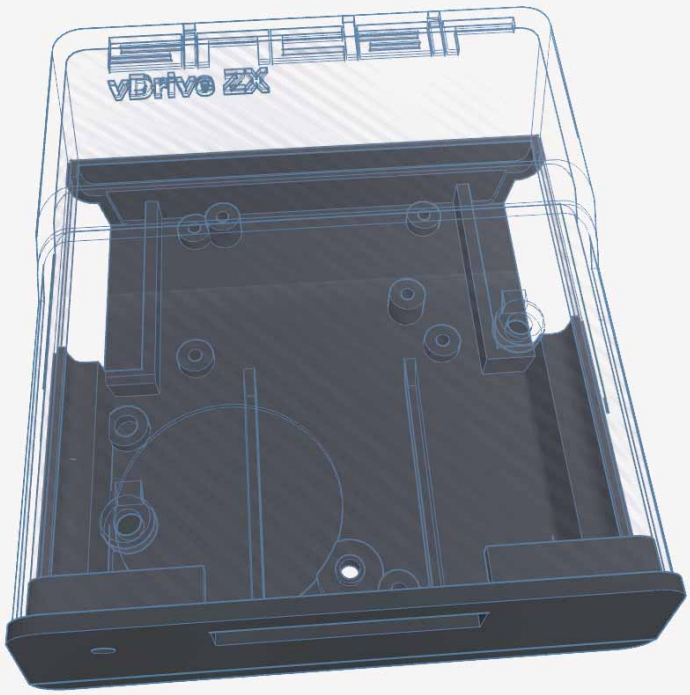
Co więcej za pomocą komend Toolkit vDrive?

Po pierwsze możemy dodać więcej wirtualnych napędów
.MKDRV 3
doda kolejne trzy napędy do już istniejących, wszystkie na początku mają status EMPTY, ponieważ nie przypisaliśmy do nich żadnych obrazów nośników Microdrive.
Jeżeli chcemy usunąć któryś z napędów, to używamy komendy
.RMDRV 2
Po dokonywaniu takich zmian polecam sprawdzić aktualne przypisanie istniejących wirtualnych napędów komendą
.LV
Co jeżeli chcemy zapisać konfigurację napędów i obrazów Microdrive do późniejszego użytku? Autor vDrive pomyślał o tym i stworzył banki z konfiguracjami. Istotna kwestia to stworzenie najpierw banku, a potem zapisywanie do niego konfiguracji. Bank tworzymy komendą
.MKBANK ˝BANK1˝
a przełączamy go komendą
.SB ˝BANK1˝
Oczywiście dany bank możemy usunąć komendą
.RMBANK ˝BANK1˝
z jednym tylko zastrzeżeniem – nie można skasować aktualnie wybranego banku. Aby to zrobić, musimy przełączyć się najpierw na inny bank. W przypadku nazw banków mamy do dyspozycji 10 znaków, a cała konfiguracja ląduje w pliku VDRIVEZX.CFG w głównym katalogu karty SD.

Sam Toolkit umożliwia szereg operacji na plikach – kopiowanie, przenoszenie ich, kontrolę dostępu (zapis/tylko odczyt), aktualizację firmware i wiele innych czynności, łącznie z wyłączeniem Toolkita, ale omówienie tego wykracza poza potrzeby jak i objętość tego artykułu.

Po co mi vDrive?

Po pierwsze, żeby pobawić się bezstresowo takim trochę wirtualnym Microdrivem, ale w połączeniu z realną, działającą maszyną. Dla części osób zmienianie padów w kasetce, szukanie działającego napędu etc. może zniechęcić do tematu.



Po drugie, vDrive to pierwsze rozwiązanie cyfrowe pozwalające na bardzo prosty i intuicyjny backup zawartości kasetek (pod warunkiem, że mamy sprawny napęd i kabel do podłączenia jeszcze jednego napędu Microdrive). Niestety Microdrive i Interface 1 wykluczają podłączenie innego systemu pamięci masowej nadpisującego swoim ROMem system ZXa, czyli ani OPUS ani MGT D+, Beta, czy divMMC/IDE nie zadziałają. Pozostaje transmisja szeregowa na inny komputer albo zgrywanie na kasetę magnetofonową. Oczywiście vDrive bezproblemowo działa z Multiface 1 czy Multiface 128 – i tutaj pojawiają się nowe możliwości „zabawy".

Osobom posiadającym dostęp do drukarki 3D polecam wydrukowanie sobie obudowy do vDrive, co pozwoli nam uniknąć kanibalizacji napędu. Niestety projekt jaki pojawił się na Thingiverse nie jest idealny do druku na każdej drukarce z racji zbyt cienkich ścian.

Reasumując – zarówno Microdrive w swojej oryginalnej postaci jak i sam vDrive jest w dzisiejszych czasach adresowany do wąskiej grupy spectrumowców – którym dostarczy bardzo komfortowego i bezstresowego użytkowania. ■

DZIESIĘĆ RZECZY JAKIE POWINNIŚMY WIEDZIEĆ O NAPĘDACH MICRODRIVE I ICH NOŚNIKACH

1. Główny problem, jaki spotykają użytkownicy Microdrive i Interface 1 to błędy w ROMie w wersji 1, powodujące brak komunikacji z napędem Microdrive.
2. Wersję ROM Interface 1 można sprawdzić komendą: CLOSE #0: PRINT PEEK 23729 W wyniku otrzymamy 0 dla ROM w wersji 1 albo 80 dla ROM w wersji 2.
3. Po włączeniu komputera z włożoną do napędu kasetką możemy po prostu napisać RUN (R) i uruchomić w ten sposób pierwszy program na taśmie Microdrive.
4. Nowy nośnik przygotowujemy do użycia formatując komendą: FORMAT "m";1;"blank"
5. Zawartość kasetki Microdrive wyświetlamy komendą: CAT 1
6. Stare kasetki Microdrive wymagają dzisiaj obligatoryjnie zmiany „padu", czyli filcowego docisku taśmy (podobnie jak w kasetach magnetofonowych). Z powodzeniem można użyć 2 mm filcu i dwustronnej taśmy samoprzylepnej. Stare filce kruszą się i odpadają, co w efekcie może unieruchomić albo nawet uszkodzić napęd. Nie ma znaczenia czy będzie to fabrycznie nowa kasetka, czy używana – klej, którym pad/gąbka jest przyklejony przestaje trzymać, a sama gąbka rozsypuje się wraz z upływem czasu. Tutaj pady do kaset magnetofonowych wychodzą na dużo trwalsze i bezproblemowe.
7. Możemy podłączyć na raz do 8 napędów. W zamierzeniach, wraz z opcją sieci, za pośrednictwem Interface 1 można było w ten sposób udostępniać ich zawartość innym komputerom.
8. Firma Aztec opracowała konkurencyjne rozwiązanie – Wafadrive – miało być wytrzymalsze niż Microdrive, dziś okazuje się być… dokładnie odwrotnie.
9. Pomimo innowacyjności i atrakcyjnej ceny napędy Microdrive nie zdobyły dużej popularności wśród wydawców – oprócz programów użytkowych takich jak Tasword czy Masterfile na nośnik trafiły tylko pojedyncze gry jak Ant Attack dołączony do zestawu Games przez firmę Sinclair.
10. W dzisiejszych czasach napęd Microdrive jest dość problematyczny w użytkowaniu, wymaga sprawnego Interface 1, sprawnego napędu Microdrive wraz z nietypową taśmą połączeniową oraz kasetek z wymienionymi padami dociskającymi taśmę magnetyczną. Z tych czterech potencjalnych problemów trzy niweluje powstałe urządzenie - vDrive ZX.
11. Aby nie "kanibalizować" oryginalnych obudów Microdrive w sieci jest dostępny projekt 3d gotowy do druku. Djordje z Retroradionics również zapowiedział produkcję replik obudów.

OLDSCHOOL DEMOMAKING

mat/ESI **CZĘŚĆ 3**

Artykuł ten jest bezpośrednią kontynuacją „cyklu", którego dwie pierwsze części napisałem w roku 2012 dla speccy.pl i z którymi można (a nawet należy) zapoznać się przed przeczytaniem części trzeciej i ewentualnych kolejnych.

Tak jak pisałem na początku pierwszej części, tekst ten wymaga przynajmniej podstawowej znajomości asemblera Z80 i nie jest kursem tegoż asemblera. Dobrym wprowadzeniem do asemblera może być publikowany w ***Zin80*** cykl *Asembler Z80 od podstaw dla prawie każdego* autorstwa Tygrysa. Poniższy zaś tekst pozwoli nauczyć się, jak tworzy się podstawowe efekty wykorzystywane w demach na ZX Spectrum.

Kod prezentowany w tekście kompilowany jest przy użyciu kompilatora *Pasmo* i używa pewnych, specyficznych dla niego, konstrukcji. Nie powinno to jednak stanowić przeszkody w użyciu dowolnego innego kompilatora – może co najwyżej wymagać trochę więcej pisania.

W dwóch pierwszych częściach opisałem tworzenie podstawowych scrollerów pracujących bezpośrednio w pamięci ekranu, w części trzeciej zajmiemy się scrollerami z buforem. Scroller taki różni się od wersji pracującej „w ekranie" tym, że dane bitmapowe są najpierw przygotowywane w buforze w innym obszarze pamięci, a dopiero stamtąd kopiowane są na ekran. Dzięki temu możliwe jest uzyskanie stosunkowo prostymi metodami efektów, które robione bezpośrednio na samym ekranie byłyby albo bardzo trudne/kosztowne czasowo albo wręcz niewykonalne. Wadą takiego rozwiązania jest oczywiście to, że musimy obrobić znacznie więcej danych – najpierw raz sam scroller, potem przerzucenie go na ekran, a następnie na początku kolejnej ramki kasowanie poprzedniego stanu, jeśli w danej ramce scroller zmienia pozycję. Mimo tego zwykle wynik wart jest wysiłku, a odpowiednia optymalizacja pozwala zaoszczędzić czas w niektórych miejscach.

Pierwszym przykładem będzie scroller, który, poza przesuwaniem tekstu w poziomie, będzie się dodatkowo przesuwał w pionie w górę i w dół. Na początek zakładamy, że scroller będzie rysowany na pustym ekranie, potem rozszerzymy go również o tło.

Początek kodu będzie tylko odrobinę bardziej skomplikowany niż w przypadku „zwykłego" scrollera.

```
    org $8000
main_loop:
    halt
    call  clear_scroll
    call  one_scroll_buffer
    call  copy_buffer
    ld    bc,$7ffe
    in    a,(c)
    and   1
    jr    nz,main_loop
    ret
```

Program zaczynamy pod adresem $8000 i nie bawiąc się w przygotowania (tym zajmiemy się później) zaczynamy główną pętlę. HALT synchronizuje nasz kod z rastrem, a następnie wywołujemy po kolei procedury kasowania poprzedniej pozycji scrollera (`clear_scroll`), generowania następnej pozycji (`one_scroll_buffer`), i kopiowanie wygenerowanych danych w ekran (`copy_buffer`). Następny fragment odczytuje stan klawiatury sprawdzając, czy została wciśnięta spacja i jeśli nie, to skacze do początku głównej pętli (`main_loop`). RET kończy wykonanie w przeciwnym wypadku.

Najpierw zajmiemy się procedurą właściwego scrollowania – jest ona bardzo zbliżona do „zwykłego" scrollera w ekranie.

Zmienna `bpos` przechowuje licznik kolejnych przesunięć licząc od 7 do 0 – sprawdzamy czy jest w niej zero, jeśli tak to znaczy, że musimy przesunąć pozycję w naszym tekście i przygotować kolejny znak do wsunięcia do bufora (`move_text`). Jeśli licznik nie doszedł do zera, skaczemy do właściwego przesuwania całego bufora (`do_scroll`).

```
one_scroll_buffer:
    ld    a,(bpos)
    or    a
    jr    z,move_text
    dec   a
    ld    (bpos),a
    jr    do_scroll
```

Jeśli licznik osiągnął zero (albo został na zero ustawiony – zmienna `bpos` ma na początku właśnie wartość zero) ustawiamy go na 7 – na zakończenie kodu z `move_text` dojdziemy do części `do_scroll`, która wykona pierwsze przesunięcie, stąd wartość 7, a nie 8.

```
move_text:
    ld    a,7
    ld    (bpos),a
```

Zmienna `text_addr` przechowuje aktualny adres w tekście scrollera, spod którego mamy pobrać kolejny znak – pobieramy spod tego adresu jeden bajt, zwiększamy adres o jeden, zapisujemy w to samo miejsce, a potem sprawdzamy czy pobrany bajt ma wartość 13 (ENTER) – taki przyjąłem znacznik końca tekstu, mógłby być oczywiście dowolny inny, nie występujący, w tekście. Jeśli nie doszliśmy do końca – przeskakujemy do właściwego pobierania znaku, jeśli tak – ładujemy do HL ponownie początek tekstu (etykieta text), pobieramy bajt, zwiększamy adres o jeden i zapisujemy do `text_adr`.

```
text_adr: equ $+1
    ld    hl,text
    ld    a,(hl)
    inc   hl
    ld    (text_adr),hl
```

```
    cp      13
    jr      nz,get_char
    ld      hl,text
    ld      a,(hl)
    inc     hl
    ld      (text_adr),hl
```

Kod znaku z A przepisujemy do HL a następnie mnożymy przez 8 i dodajemy $3C00. Pod adresem $3D00 w ROMie ZX Spectrum znajdują się bitmapy standardowego zestawu czcionek, dodajemy jednak $3C00, ponieważ drukowalne kody znaków zaczynają się od 32 (spacja), które przemnożone przez 8 da nam 256 (czyli $100) – moglibyśmy odejmować od A 32 albo przechowywać tekst z wartościami od razu pomniejszonymi o 32, ale pierwsza opcja to dodatkowa operacja, druga zaś wymaga wcześniejszego przygotowania tekstu. Użycie $3C00 zamiast $3D00 jest może odrobinę mniej czytelne, ale upraszcza kod nie wpływając na jego wydłużenie.

```
get_char:
    ld    h,0
    ld    l,a
    add   hl,hl
    add   hl,hl
    add   hl,hl
    ld    de,$3c00
    add   hl,de
```

Po wyliczeniu adresu naszego znaku (jest w HL), do DE ładujemy adres naszego bufora dla tego znaku (8 bajtów pod adresem `buf`) i wykonujemy 8 razy LDI kopiując kolejne 8 bajtów spod HL pod DE.

```
    ld    de,buf
    rept 8
      ldi
    endm
```

Po przygotowaniu znaku w buforze możemy przystąpić do właściwego przesuwania scrollera. Bufor scrollera ma 256 bajtów, czyli 8 linii po 32 bajty. Zaczynamy od załadowania do HL adresu ostatniego bajtu pierwszej linii bufora a do DE adresu bufora znaku. 8 w B to oczywiście licznik linii scrollera.

```
do_scroll:
    ld    hl,scroll_buf+$1f
    ld    de,buf
    ld    b,8
```

Na początku zewnętrznej pętli odkładamy adres końca linii z HL na stos, a do C ładujemy licznik bajtów w linii – 32. Do A pobieramy bajt z bufora znaków, przesuwamy go w lewo, tak, że najstarszy bit wysuwa się do znacznika *Carry,* a przesuniętą wartość zapisujemy z powrotem do bufora znaku.

```
loop1:
    push hl
    ld c,32
    ld a,(de)
    rla
    ld (de),a
```

Wewnętrzna pętla pobiera bajt z bufora, przesuwa go w lewo równocześnie wsuwając bit z *Cy* na najniższą pozycję, a najstarszy bit wysuwając do *Cy*. Zmniejszamy następnie HL, czyli przechodzimy w lewo do kolejnego znaku w linii, zmniejszamy licznik wewnętrznej pętli i jeśli nie doszliśmy do zera, zamykamy pętlę. Żadna z instrukcji w pętli poza RLA nie modyfikuje *Cy,* dzięki czemu kolejne bity „przeskakują" z bajtu do bajtu w lewo.

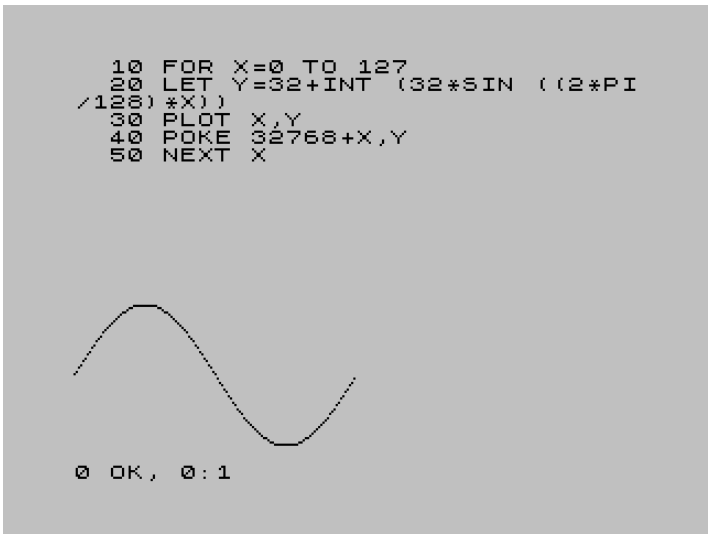
```
loop2:
    ld    a,(hl)
    rla
    ld    (hl),a
    dec   hl
    dec   c
    jr    nz,loop2
```

Po zamknięciu wewnętrznej pętli przeskakujemy do kolejnego bajtu w buforze znaku, zwiększając DE a następnie przeskakujemy do kolejnej linii głównego bufora – podnosimy ze stosu HL, odkładamy chwilowo BC (B zawiera licznik zewnętrznej pętli), dodajemy BC do HL i podnosimy BC ze stosu. Na koniec DJNZ domyka zewnętrzną pętlę przesuwając po kolei 8 linii po 32 bajty.

```
    inc   de
    pop   hl
    push  bc
    ld    bc,32
    add   hl,bc
    pop   bc
    djnz  loop1
    ret
```

Zanim przejdziemy do procedury kopiowania bufora w ekran, musimy przygotować „tablicę sinusów", czyli zestaw kolejnych pozycji na jakie mamy kopiować bufor w kolejnych ramkach, żeby ruch był „gładki". Możemy te dane wygenerować na wiele sposobów – jeden z nich to prosty programik w BASICu ZX Spectrum:

```
10 FOR X=0 TO 127
20 LET Y=32+INT (32*SIN ((2*PI/128) * X ))
30 PLOT X,Y
40 POKE 32768+X,Y
50 NEXT X
```



Program ten wyliczy kolejne 128 wartości pełnego okresu sinusoidy o amplitudzie 64 pikseli i wartości te zapisze do pamięci od adresu 32768 (oraz „wyplotuje" na ekran). Program można uruchomić na emulatorze i po wykonaniu zapisać do pliku 128 bajtów spod adresu 32768 – potem ten plik (w naszym wypadku *sin.bin*) wczytamy do generowanego kodu wynikowego naszego programu pod adres `sin_tab`.

Procedura kopiowania zaczyna się od deklaracji zmiennej `sin_pos` – będzie to pozycja w naszej „tablicy sinusów", na której ma się w danej klatce pojawić nasz scroller. Do tej pozycji (w HL) dodajemy adres tablicy a następnie pobieramy z niej bajt i dodajemy do nie-

go 50 – w ten sposób najwyższy punkt, w którym może się znaleźć nasz scroller to 50 linia ekranu, najniższy zaś to 114 linia (plus 8 linii samego scrollera). Wybrana pozycja to nie tylko kwestia estetyki, ale również tego, że cała procedura scrollera od początku ramki zajmuje czas właśnie do mniej więcej 50 linii ekranu – gdyby scroller znalazł się wyżej, pojawiłby się brzydki efekt „zrywania". Przesunięcie pozwala uniknąć tego problemu, a jak ustalić konkretny punkt tego przesunięcia zademonstruję później.

```
copy_buffer:
sin_pos:  equ $+1
    ld    hl,0
    ld    de,sin_tab
    add   hl,de
    ld    a,(hl)
    add   a,50
```

Kolejny fragment wylicza w DE adres danej linii ekranu zgodnie z organizacją pamięci ZX Spectrum. Sam algorytm nie ma w tej chwili strategicznego znaczenia – używając B jako rejestru pomocniczego, program wykonuje ciąg operacji na bitach uzyskując, na koniec w DE właściwy adres w pamięci ekranu.

```
    ld    b,a
    and   a
    rra
    scf
    rra
    and   a
    rra
    xor   b
    and   $f8
    xor   b
    ld    d,a
    xor   a
    xor   b
    and   $c7
    xor   b
    rlca
    rlca
    ld    e,a
```

Po wyliczeniu pozycji zaczynamy kopiowanie – do HL ładujemy adres bufora, do B licznik linii do skopiowania, po czym odkładamy BC i DE na stos i kopiujemy 32 bajty spod HL pod DE. Na koniec HL wskazuje na początek kolejnej linii w buforze.

```
    ld     hl,scroll_buf
    ld     b,8
copy_buf1:
    push   bc
    push   de
    rept   32
      ldi
    endm
```

Podnosimy DE ze stosu i wyliczamy adres następnej linii – tym razem bez skomplikowanych obliczeń. Najpierw zwiększamy D o 1 – w ten sposób przeskakujemy do następnej linii w ramach kolejnych „znaków", czyli sekwencji ośmiu linii. Jeśli w efekcie pierwsze 3 bity nie są zerowe (wynik `and 7`), to wszystko jest OK – przeskakujemy dalej. Jeśli nie, to znaczy, że przeskoczyliśmy do kolejnej linii znakowej i musimy dodatkowo zwiększyć E o 32 żeby to skompensować. Jeśli po tej operacji ustawione zostało *Cy,* to znaczy, że mamy już ustawiony właściwy adres, bo przeskoczyliśmy do kolejnej tercji ekranu. Jeśli nie, to dodatkowo cofamy D o 8, wracając do początku właściwej linii znakowej.

Na koniec podnosimy BC ze stosu i zamykamy pętlę kopiującą kolejne linie.

```
    pop de
    inc d
    ld a,d
    and 7
    jp nz,copy_skip
    ld a,e
    add a,32
    ld e,a
    jp c,copy_skip
    ld a,d
    sub 8
    ld d,a
copy_skip:
    pop bc
    djnz copy_buf1
    ret
```

Wszystkie te operacje są dość kosztowne obliczeniowo i później pokażę, jak można ich uniknąć tablicując sobie odpowiednie adresy.

Na koniec pozostała jeszcze tylko procedura kasowania poprzedniej wersji scrollera przed wyświetleniem aktualnej. Zaczynamy od pobrania poprzedniej pozycji w „tablicy sinusów" – wartość tę zapisujemy do HL, a następnie zwiększamy o 1 „przycinając" do długości naszej tablicy, czyli 128 bajtów i nową wartość zapisujemy z powrotem do `sin_pos`.

```
clear_scroll:
    ld    a,(sin_pos)
    ld    h,0
    ld    l,a
    inc   a
    and   $7f
    ld    (sin_pos),a
```

Spod ustalonej pozycji pobieramy bajt, i tak jak przy kopiowaniu, zwiększamy go o 50 i wyliczamy pozycję na ekranie – tym razem w HL.

```
    ld     de,sin_tab
    add    hl,de
    ld     a,(hl)
    add    a,50
    ld     b,a
    and    a
    rra
    scf
    rra
    and    a
    rra
    xor    b
    and    $f8
    xor    b
    ld     h,a
    xor    a
    xor    b
    and    $c7
    xor    b
    rlca
    rlca
    ld     l,a
```

Po wyliczeniu adresu zaczynamy zewnętrzną pętlę – 8 linii z licznikiem w B. Odkładamy BC i HL na stos, ładujemy do B kolejny licznik – 32 bajty w linii, zerujemy A i w wewnętrznej pętli ładujemy pod HL

zero z A, zwiększając HL. Następnie podnosimy HL, i znowu, tak jak przy kopiowaniu, przeliczamy adres kolejnej linii. Na koniec podnosimy BC z licznikiem zewnętrznej pętli i domykamy tą pętlę, kończąc kasowanie. Znowu widać, że cały proces z przeliczaniem adresów nie jest zbyt optymalny i w kolejnej części tekstu pokażę, jak można go przyspieszyć.

```
      ld    b,8
  clear_loop1:
      push  bc
      push  hl
      xor   a
      ld    b,32
  clear_loop2:
      ld    (hl),a
      inc   hl
      djnz  clear_loop2
      pop   hl
      inc   h
      ld    a,h
      and   7
      jp    nz,clear_skip
      ld    a,l
      add   a,32
      ld    l,a
      jp    c,clear_skip
      ld    a,h
      sub   8
      ld    h,a
  clear_skip:
      pop   bc
      djnz  clear_loop1
      ret
```

Na koniec pozostają jeszcze tylko definicje zmiennych – 8 bajtów bufora znaków (`buf`), 256 bajtów głównego bufora (`scroll_buf`), pozycja scrollera w ramach bajtu (`bpos`), sam tekst scrollera (`text`) i „tablica sinusów" (`sin_tab`), wczytywana z pliku binarnego dyrektywą *incbin*. Ostatnia linia to oczywiście informacja dla kompilatora *Pasmo*, niezbędna do poprawnego działania opcji automatycznego generowania pliku TAP z loaderem.

```
  buf:      ds 8
  scroll_buf: ds 256
  bpos:     db 0
  text:     db "oldschool demomaking, czesc3
(c) 2020 mat "
            db "for Zin80...", 13
  sin_tab:  incbin "sin.bin"
      end $8000
```

Zakładając, że nasz kod znajduje się w pliku `scroller.asm`, kompilację wykonujemy przez:

```
pasmo --tapbas scroller.asm scroller.tap
```

W wyniku dostaniemy plik TAP, gotowy do uruchomienia w wybranym emulatorze albo na fizycznym sprzęcie.

W kolejnej części cyklu dodamy do naszego scrollera tło czyli wyświetlanie obrazka zamiast pustego ekranu. Spróbujemy go też przyspieszyć tak, aby zajmował mniej czasu procesora, kosztem rozmiaru samego kodu. ■

→ OLSCHOOOL DEMOMAKING, CZĘŚĆ 1
https://www.speccy.pl/articles.php?article_id=11
→ OLDSCHOOL DEMOMAKING, CZĘŚĆ 2
https://www.speccy.pl/articles.php?article_id=12
→ PACZKA PLIKÓW DO ARTYKUŁU
https://www.speccy.pl/images/articles/demomaking_part3.zip

PIERWSZE KROKI SPECTRUMOWEGO KODERA

SACHY / WANTED TEAM ^ RESISTANCE ^ LAMERS **CZĘŚĆ 2**

W poprzednim artykule opisałem niezbędne podstawy, by napisać i zasemblować bardzo podstawowy, ale w pełni działający kawałek kodu, dający się uruchomić na emulatorze lub prawdziwym sprzęcie i wyświetlający pewne „ćwiczeniowe śmieci" na ekranie. Opisałem też ogólną „mapę pamięci" ZX Spectrum, czyli jak wyglądają i gdzie leżą (między jakimi adresami) obszary pamięci ROM i RAM. Wiemy też gdzie znajduje się pamięć ekranu ($4000) i bufor atrybutów kolorów ($5800) oraz gdzie najlepiej pokierować asembler *Pasmo*, by układał w pamięci nasz program (dla przypomnienia, wybraliśmy adres $8000). Sprawę kolorów poznaliśmy dość dokładnie, natomiast co do samej pamięci ekranu wspomniałem jedynie, że każdy bajt czy nawet bit tam wpisany powoduje wyświetlania pikseli – w takim kształcie, jak ustawione na 1 bity we wpisywanym bajcie. Zasygnalizowałem też wstępnie, że pamięć ekranu jest zorganizowana w trochę bardziej skomplikowany sposób niż bufor atrybutów i że do tego tematu wrócimy w kolejnym odcinku. Tak naprawdę jest to kolejny logiczny krok w poznawaniu Spectrum, więc zaczynamy.

Jak wcześniej wspominałem, lubię ZX za jego prostotę. Otóż w zasadzie komputer ten posiada tylko jedną, podstawową rozdzielczość ekranu (pomijam tricki oparte na cyklowaniu i atrybutach) i ogólnie jest to 256 pikseli w poziomie oraz 192 w pionie. Mamy więc 192 poziome linie, a każda z nich jest w stanie wyświetlić 256 punktów. Wszystkie te piksele poukładane są w wymienionych wyżej liniach w pamięci ekranu, zaczynającej się w RAM od adresu $4000.

Przyjrzyjmy się na razie pojedynczej linii i niech będzie to pierwsza od góry linia ekranu. Jak wiadomo obraz wyświetlany jest od lewej do prawej krawędzi widocznego ekranu. Wspomniane 256 punktów w danej linii jest umieszczone w 32 kolejnych bajtach. Dlaczego w 32? Po prostu, każdy bajt przechowuje 8 pikseli, jeśli więc podzielimy 256 pikseli na grupy po 8 bitów wyjdzie właśnie 32 bajty. Co więcej, pierwsza linia ekranu to kawałek pamięci (a konkretnie: wspomniane kolejne 32 bajty), które znajdują się dokładnie na początku naszego bufora ekranu (nazwy bufor/obszar/pamięć ekranu można stosować zamiennie). Jeżeli więc do któregokolwiek bajtu znajdującego się za adresem $4000 (a przed $5800, czyli przed początkiem bufora atrybutów kolorów) wpiszemy jakąkolwiek wartość różną od zera, „coś" pojawi się na ekranie. To „coś" zależy od zawartości wpisanego w ten fragment pamięci bajtu. Jeżeli wpiszemy tam bajt taki jak $01, to zapali się jeden pixel. Wpisując np. $20 też zapalimy jeden pixel, ale na innej pozycji. Dlaczego? Bo tu nie chodzi o bajt jako liczbę, ale raczej jako „pole bitowe" o możliwych do ustawienia ośmiu pozycjach bita/pixela. Zamieńmy liczby $01 i $20 z zapisu szesnastkowego (hex) na binarny i otrzymamy %00000001 i %00100000. Jak widać, jeśli wiemy, że zero w naszym bajcie to brak pixela, a jedynka to piksel narysowany, możemy w ten sposób (czyli wrzucając bajty z bitami w ten określony obszar pamięci) zacząć rysować na ekranie. Od tej chwili tylko od nas zależy, czy na ekran wrzucimy statyczny obrazek (który jest właśnie zbiorem punktów umieszczonych w bajtach), wyliczymy i narysujemy tzw. sinus-plotter, scroll czy wektorka. Wszystko, co będziemy chcieli pokazać na ekranie, ostatecznie sprowadza się do wpisywania bitów i bajtów w ten obszar, a co tam chcemy pokazać, to już zupełnie inna sprawa.

Na razie skupimy się więc na rzeczach podstawowych, czyli prostym rysowaniu i czyszczeniu ekranu. Zanim to jednak zrobimy, trzeba poznać mechanizm pętli, czyli powtarzania tej samej operacji.

Na początek zapiszemy pierwszą linię ekranu przykładowym „szlaczkiem". Tak jak w poprzednim artykule, będziemy w jego treści umieszczać tylko kod zmieniony pomiędzy etykietami `Demo_Start` i `Demo_Stop`. Reszta kodu (dyrektywy ORG, END, czy definicje adresów) pozostaje taka jak poprzednio. Na początek napiszemy najprostszą pętlę, czyli taką, w której wiemy, ile razy ma się wykonać.

```
Demo_Start:
  ei                    ; enable interrupts
  ld   hl,GFX_SCR_START ; pixel screen buffor
                        ; start address to HL
  ld   b,%00001111      ; pixel pattern to draw
                        ; in every byte on the
                        ; screen (1st line)
  ld   a,32             ; let A is our loop
                        ; counter
My_loop:
  halt                  ; HALT - let's
                        ; wait to start
                        ; a new frame

  ld   (hl),b           ; put a byte from reg. B
                        ; to the screen and show
                        ; some pixels
  inc  hl               ; add 1 to the address
                        ; so it points to another
                        ; byte
  dec  a                ; substract 1 from our
                        ; loop counter
  cp   0                ; all loops done?
  jr   nz,My_loop       ; if counter is bigger
                        ; than 0, do another run
Demo_Stop:
```

Większość elementów tego kodu poznaliśmy w artykule w poprzednim numerze Zin80. Dodatkowo Tygrys opisał w nim wszystkie instrukcje naszego procesora, jeśli więc nie znamy którejś z nich – zachęcam do odszukania jej we wspomnianym tekście, który aktualnie służy mi właśnie za szybką podręczna pomoc, gdy zapomnę detali dotyczących którejś instrukcji.

O pisaniu pętli warto wiedzieć to, że zazwyczaj na początku przygotowujemy jej działanie (w zależności od tego co chcemy osiągnąć), np. planując ile razy ma się wykonać, na jakich obszarach działać i kiedy się zakończyć, a dopiero później – co ma zrobić. Dodatkowo warto wspomnieć, że każdą pętlę, a w zasadzie każde zadanie jakie ma ona wykonać, można zazwyczaj zaprogramować w więcej niż tylko jeden sposób. To od programisty zależy jaki wariant wybierze do wykonania tego, co akurat potrzebuje. Niektóre z nich wykonują się szybciej niż pozostałe, inne potrzebują mniej miejsca w pamięci, kolejne są prostsze do napisania itd. Warto więc znać ich jak najwięcej i wykorzystywać te najlepiej nam pasujące.

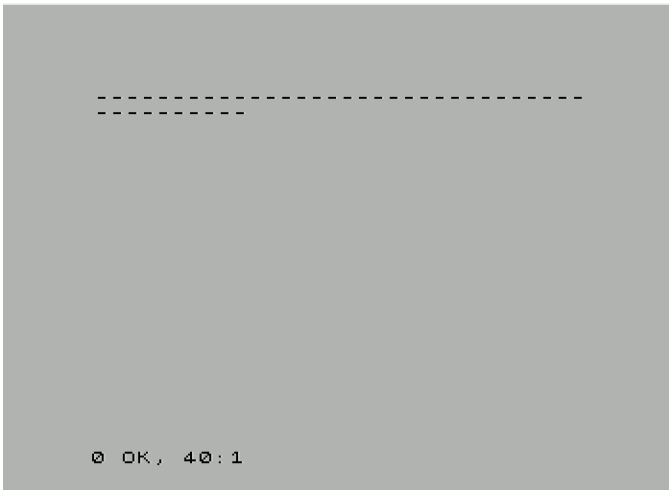
Wracając do przykładu – w rejestrach HL umieściliśmy adres początku bufora ekranu, w rejestrze B zaś bajt, który chcemy wyrysować na ekranie (pół bajta pikseli „wygaszonych", czyli skasowanych, a drugie pół pixeli „zapalonych", czyli ustawionych/widocznych). Ostatni rejestr przygotowany przed rozpoczęciem pętli rysowania to licznik, który będzie służył do sprawdzania, czy narysowaliśmy już odpowiednią liczbę bajtów (tu: zaplanowane 32). Ustawiając odpowiednie wartości w wymienionych wyżej rejestrach, przygotowaliśmy pętlę do pracy i czas ją puścić w ruch. Jako, że pętla ma się „zapętlać", rozpoczniemy ją etykietą (w tym przykładzie będzie to `My_loop`) i od niej zacznie się właściwe „ciało pętli", czyli instrukcje robocze, które wykonają właściwą pracę. Opisaną przed chwilą etykietę wykorzystamy do „zapętlania" naszej pętli, czyli do pokierowania procesorem Z80 w taki sposób, by wykonał zaplanowane czynności ponownie lub też zakończył działanie pętli, jeśli stwierdzi, że wszystko co planowaliśmy zostało wykonane. To sprawdzenie wykonania wszystkich zadań pętli możemy nazwać warunkiem zakończenia pętli. W tym przykładzie w rejestrze A mamy zwykły licznik liczby wykonań pętli, gdy więc się „skończy", będzie to oznaczało, że wszystko zostało wykonane. Zacznijmy jednak działać. Po etykiecie mamy instrukcję HALT, która spowoduje, że wykonanie pętli będzie ograniczone do jednego przebiegu (wykonania) podczas pojedynczego wyświetlania ekranu. Ma to na celu spowolnienie wyświetlania pixeli tak, żebyśmy mogli zobaczyć jak to się odbywa, czyli bajt po bajcie. Bez instrukcji HALT, nasz Z80 narysowałby wszystko tak szybko, że nasze oko zobaczyłoby tylko efekt końcowy. Tutaj chcemy widzieć jak to działa, stąd użycie przydatnego rozkazu HALT (który opisałem nieco dokładniej w poprzednim artykule). Mamy więc instrukcję HALT i czekamy dzięki niej na rozpoczęcie wyświetlania ekranu (który jak wiemy, w systemie PAL odświeżany jest 50 razy na sekundę). Następna instrukcja to LD (HL),B, która bajt z rejestru B zapisze pod adres wskazywany przez rejestry HL, czyli aktualnie w pierwszy bajt bufora ekranu i tym samym narysuje nasze pixele na ekranie. I już! Pojawiają się nasze pixele, narysowane tak, jak bity w bajcie umieszczonym w rejestrze B. Gdyby nie było pętli i chodziło nam o narysowanie tylko tych pixeli, tu moglibyśmy zakończyć nasz mały program. Jako, że chcemy narysować cały szlaczek w górnej linii ekranu, trzeba działać dalej. Dlatego więc, by nie nadpisywać kolejny raz tego samego bajtu, tą samą zawartością, musimy w rejestrach HL wpisać adres kolejnego bajtu. Tu sprawa jest prosta, bo wystarczy do pary rejestrów HL dodać jeden i w ten sposób rejestry HL będą zawierały adres kolejnej komórki pamięci widocznej na ekranie. Z80 oprócz zwykłego dodawania ma też instrukcję INC, która właśnie zwiększa podany rejestr (lub parę rejestrów, która traktowana jest tutaj jako „wspólny" rejestr HL) dokładnie o jeden i tym samym przesuwa adres w zakładanym przez nas kierunku. Tym samym przygotowujemy już warunki do ponownego uruchomienia pętli. Dalej instrukcja DEC odejmuje jeden od rejestru A, czyli zmniejsza licznik zaplanowanych wykonań pętli i… jesteśmy gotowi do kolejnego „obrotu". Instrukcja DEC jest podobna do instrukcji INC, tylko że zamiast dodawać 1 – odejmuje. Kolejna instrukcja to CP, czyli compare – porównaj – czy rejestr A jest równy zero, czyli czy nasz licznik obrotów pętli jest już „pusty", więc wszystkie obroty pętli zostały już wykonane. Jeśli nie (czyli nasz licznik w rejestrze A nie jest pusty), to kolejna instrukcja `JR NZ,My_loop` wykona skok (JR – Jump Relative) do etykiety `My_loop` (czyli początku naszej pętli) i wyrysuje kolejne pixele w kolejnym bajcie ekranu. Jeśli jednak licznik w rejestrze A będzie równy zero, to instrukcja JR nie wykona się i przejdziemy do dalszej części programu i etykiety `Demo_Stop`, za którą zakończy się działanie naszego przykładu. W jaki sposób rozkaz JR rozpozna, czy trzeba wykonać pętlę, czy też „iść dalej"? Tutaj warto wczytać się w Tygrysowy opis instrukcji: JR NZ,etykieta, CP oraz opisu flag procesora. Chodzi jednak o to, że instrukcja CP 0 sprawdzi, że w rejestrze A jest właśnie zero (licznik pętli skończył się) i ustawi w procesorze Z80 pewien bit zwany bitem lub flagą Zero (ang. Zero Flag). Ten bit/flaga jest o tyle ważny, że przydaje się właśnie w takich sytuacjach. Kiedy licznik jakiejś czynności skończy się (wyzeruje), to instrukcją CP będzie można to sprawdzić, a następująca po niej instrukcja skoku JR będzie poinformowana, że skok ma wykonać w zależności od tego, czy bit/flaga Zero jest ustawiona na 1 czy na 0. W naszym przykładzie widzimy: `jr nz,My_loop`, gdzie „nz" przed etykietą `My_loop` (do której mamy skoczyć by wykonać kolejną pętlę) oznacza właśnie „NOT ZERO", czyli skocz i wykonaj kolejną pętlę, jeśli bit/flaga Zero nie została ustawiona (zapalona) na jeden – a flaga ta będzie zapalona wtedy, gdy licznik pętli będzie równy zero. W opisie wygląda to na skomplikowane, ale sam kod jak widać jest prosty i używając go szybko łapie się o co chodzi (tylko trzeba go używać, czyli pisać własne testowe programy). Należy też tutaj napisać, że podany wyżej przykład miał za zadanie przedstawić najprostszą pętlę w zrozumiałych krokach oraz pokazać sposób użycia kilku przydatnych instrukcji. Natomiast trzeba też po-

wiedzieć, że to nie jest porządnie napisana, typowa pętla dla Z80. Otóż procesor ten ma kilka innych instrukcji i sposobów konstruowania pętli, które są zazwyczaj lepsze niż ta prosta powyżej, ponieważ albo używają mniej rejestrów, albo wykorzystują stworzone do zapętlania programu instrukcje (które same potrafią zmniejszać liczniki pętli, czy też zmieniać adresy do wpisywanie bajtów, itp.). Tutaj chodziło mi wyłącznie o pokazanie elementów, z jakich ogólnie składa się typowa pętla, oraz o jak najjaśniejsze zapisanie tego przykładu, gdyż na razie trenujemy nasze podstawowe umiejętności kodowania. Podobną pętlę możemy zapisać przy pomocy instrukcji DJNZ, tyle że inaczej trzeba zaplanować użycie rejestrów A i B oraz dokładniej wczytać się w sposób działania samej instrukcji:

```
Demo_Start:
  ei                  ; enable interrupts
  ld  hl,GFX_SCR_START ; pixel scree
                      ; buffor start
                      ; address
  ld  a,%00001111
  ld  b,32            ; B is the loop counter
My_loop:
  halt                ; HALT - let's wait
                      ; to start a new frame

  ld  (hl),a          ; put a byte from reg. A
                      ; to the screen
  inc hl
  djnz My_loop        ; do the loop or - not

Demo_Stop:
```

Poznając sposoby pisania pętli mamy już kontrolę nad tym, co dokładnie chcemy zrobić (np. jaki fragment ekranu wypełnić czy wyczyścić, jak duży obszar pamięci gdzieś skopiować, itp.). Pętle pomogą nam zapoznać się z tym, jak skonstruowana jest pamięć ekranu w ZX Spectrum, bo jest tu pewna niespodzianka. O ile zapisanie pojedynczej linii ekranu jest proste (to pokazały przykłady powyżej), to zapisanie kilku następujących po sobie linii już takie nie jest. Otóż, skoro kolejnymi bajtami wypełnialiśmy kolejne, następujące po sobie fragmenty górnej linii ekranu, to można by przypuszczać, że po zapisaniu wszystkich 32 bajtów i dołożeniu kolejnego, jego zawartość powinna znaleźć się na początku drugiej linii, tuż pod właśnie narysowaną poprzednią, prawda? Cóż, wykonajmy eksperyment i w naszych pętlach z poprzednich przykładów zmieńmy liczniki z 32 np. na 42. Również, dla czytelności naszego eksperymentu dodajmy przed pętlą wywołanie procedury z pamięci ROM, która wyczyści nam ekran (zrobimy to tylko tymczasowo, bo później napiszemy sobie własną i szybszą). Dodajemy więc rozkaz CALL 3435, zmieniamy licznik pętli na 42, uruchamiamy program i co widzimy?

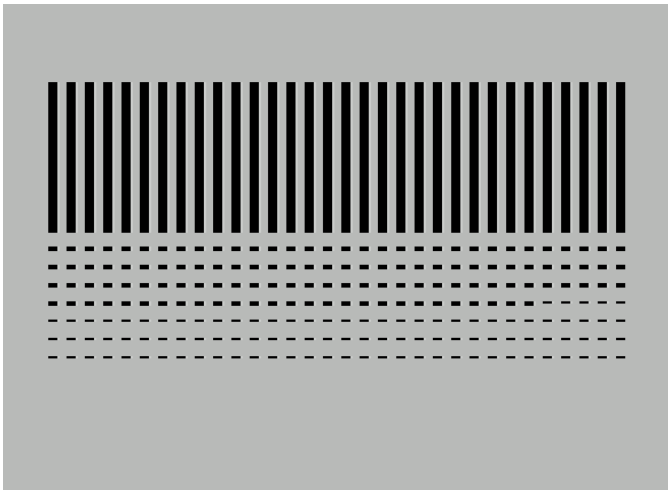
```
Demo_Start:
  ei                        ; enable interrupts
  call 3435                 ; clear screen

  ld  hl,GFX_SCR_START      ; pixel screen
                            ; buffor start
                            ; address
  ld  a,%00001111
  ld  b,32+10               ; B is our
                            ; loop counter

My_loop:
  halt                ; HALT - let's wait to
                      ; start a new frame

  ld  (hl),a          ; put a byte from reg. A
                      ; to the screen
  inc hl              ;
  djnz My_loop

Demo_Stop:
```



Zamiast spodziewanych kolejnych bajtów w linii nr 2, bajty pojawiają się, owszem, ale 8 linii niżej. Powtórzmy eksperyment dla wartości licznika (rejestr B) równej 255 (a potem zero – warto spróbować). Jak widzimy, po zapełnieniu całej linii, kolejna pojawia się 8 linii niżej. To powoli pokazuje nam, jak zorganizowany jest ekran w ZX Spectrum i z czym będziemy musieli mierzyć się w przyszłości. Żeby jednak zobaczyć, jak zapełnić kolejne linie, musimy zmienić typ pętli na inny, gdyż właśnie wykorzystaliśmy cały 1-bajtowy licznik pętli dostępny dla instrukcji DJNZ i musimy użyć licznika większego, czyli np. pary rejestrów DE, co da nam licznik 16-bitowy (2 bajtowy), a więc o wiele bardziej pojemny (o pojemności od 0 do 65535). Powróćmy do pierwszej wersji pętli (bez instrukcji DJNZ), ale z pewnymi modyfikacjami: wielkości licznika i sposobu sprawdzania, czy już się wyczerpał:

```
Demo_Start:
  ei                  ; enable interrupts
  call 3435           ; clear screen

  ld  hl,GFX_SCR_START ; pixel screen buffor
                       ; start address
  ld  b,%00001111     ; pixel pattern to draw
                      ; in every byte on the
                      ; screen (1st line)
  ld  de,32*192       ; let DE is our loop counter
My_loop:
  halt                ; HALT - let's wait to
                      ; start a new frame

  ld  (hl),b          ; put a byte from reg. B
                      ; to the screen and show
                      ; some pixels
  inc hl              ; add 1 to the address so
                      ; it points to another byte
  dec de              ; substract 1 from our
                      ; loop counter

  ld  a,d             ; load higher 16bit counter
                      ; part to reg. A...
  or  e               ; ...and OR it with lower
                      ; counter byte, just to get
                      ; 0 or non-0

  jr  nz,My_loop      ; if counter is different
                      ; than 0, do another run

Demo_Stop:
```

Odpalmy powyższy przykład i popatrzmy na rezultat. Jak widać, po zapełnieniu ośmiu linii „pierwszych", następuje zapełnienie ośmiu linii „drugich" a potem „trzecich"… i tak do zarysowania 1/3 ekranu. Tu kolejna niespodzianka. Jak wiemy, ekran składa się ze 192 linii,



po 32 bajty każda. W sumie oznacza to, że bufor ekranu zajmuje 6144 bajty (192*32=6144). Jeżeli pozwolimy naszemu programowi działać dostatecznie długo, to zobaczymy, że ekran (mimo że wypełniamy kolejne bajty w pamięci – jeden po drugim) zapełni się jakby w trzech obszarach – najpierw zarysuje się cały obszar górny, potem zacznie być wypełniany środkowy, a na samym końcu dolny. Jeżeli z kolei 6144 bajty podzielimy na 3 obszary, to wyjdzie nam, że każdy z nich zajmuje 2048 bajtów. Tak więc, jeśli chcemy coś narysować w dolnej części ekranu, musimy „przeskoczyć" 4KB, by znaleźć się na początku dolnego obszaru. Każda z tych trzech części, zbudowana jest w taki sposób, jaki pokazała nam pętla wypełniająca pierwsze 256 bajtów bufora ekranu. Mamy więc w każdej z tych 3 części jakby 8 linii wielkości znaku w BASICu. Znaki wypełniają obszar o wysokości 8 linii. Stąd też, gdy wypełniamy obszar górnej części (zapisując kolejny bajt po poprzednim bajcie, w sposób ciągły i bez przeskakiwania) to najpierw wypełnimy pierwszą „linię grafiki" (o wielkości 32 bajtów) w pierwszej linii znaków, następnie pierwszą linię w drugiej linii znaków i będziemy tak wypełniać wszystkie pierwsze linie (każda po 32 bajty) wszystkich 8 linii znaków. Dopiero, gdy to się dokona, wrzucając dalej kolejne bajty w bufor ekranu zaczniemy wypełniać drugie „linie grafiki" w każdej z 8 linii znaków, potem trzecie „linie grafiki" w każdej z 8 linii znaków i… tak aż do zapełnienia ostatniej, ósmej linii „grafiki" we wszystkich liniach znaków. W taki sposób wypełnimy górny obszar ekranu. Gdy obszar będzie już zarysowany i dodamy kolejne bajty, zaczniemy wypełniać obszar środkowy w taki sam sposób, jak obszar górny, czyli najpierw pierwsze „linie grafiki" (znów po 32 bajty) we wszystkich 8 liniach znaków, potem drugie „linie grafiki" we wszystkich 8 liniach znaków i tak do zapisania obszaru środkowego. Jak łatwo się domyślić, obszar dolny zaczniemy zapisywać zaraz po zakończeniu wypełniania obszaru środkowego, dokładnie w taki sam sposób jak oba obszary poprzednie. Opis ten zapewne nie jest zbyt czytelny, ale ciężko opisać to słowami języka naturalnego. Najlepszy sposób na poznanie, w jaki sposób zorganizowany jest ekran w ZX Spectrum, to własnoręczne uruchomienie podanych przykładów i zabawa w modyfikowanie liczników, czy też wartości bajtu wpisywanego na ekran (nic nie stoi na przeszkodzie, by do tego bajtu wpisywać np. kopię wartości dolnego bajtu licznika, czyli np. rejestr E). Dodatkowo, polecam zakomentowanie sobie roboczo rozkazu HALT czy CALL 3435 (oraz inne podobne eksperymenty, jak np. dodanie kilku rozkazów HALT jeden po drugim) i sprawdzenie, jaki przyniesie to efekt. Taka zabawa kodem zdecydowanie lepiej pozwoli zrozumieć działanie ekranu, niż jakikolwiek opis słowny. Warto też poeksperymentować z różnymi pętlami, np. wypełniając każdy z 3 obszarów ekranu innym bajtem. Hm, czy nasze eksperymenty z buforem ekranu czegoś nie przypominają? Mnie od razu w oczy rzuciło się podobieństwo do… „loading screenów" z gier (◀), tak charakterystycznych właśnie dla ZX Spectrum!

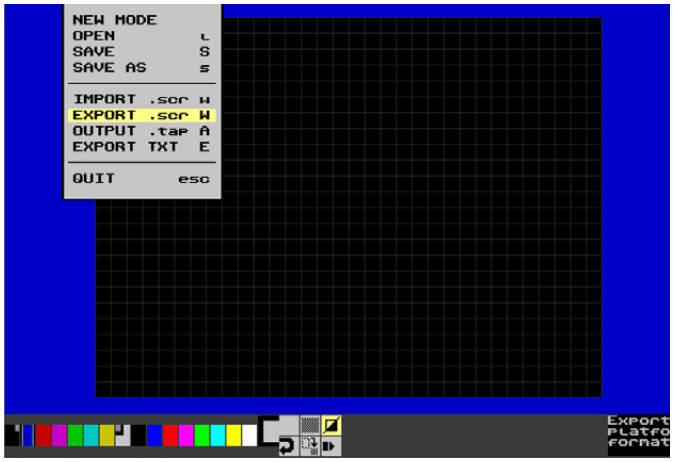
Powyższy opis, wraz z przedstawionym w poprzednim artykule buforem atrybutów kolorów kończy sprawę zapoznawania się z budową spectrumowego ekranu. Może warto jeszcze wspomnieć o opisanych poprzednio adresach początków buforów ekranu i atrybutów kolorów. Jak pamiętamy, zdefiniowaliśmy sobie je w kodzie jako:

```
GFX_SCR_START:
 equ $4000            ;16384: 2048*3, 6144 bytes
GFX_SCR_ATTR:
 equ $5800            ;22528: 32x24, 768 bytes
```

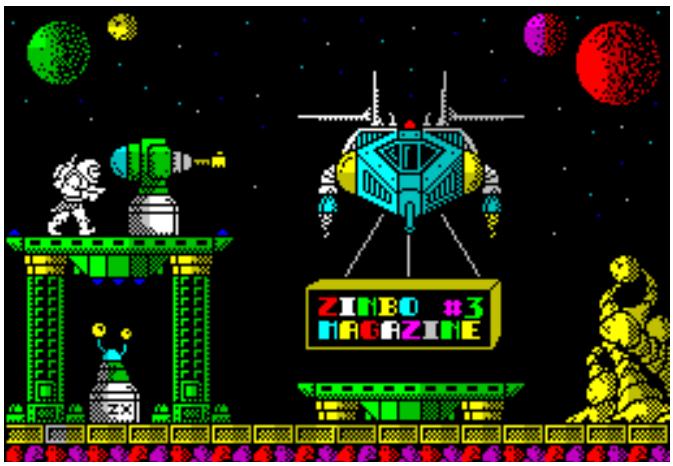
Przy definicjach (w komentarzach) zapisałem sobie drobne „przypominajki", które pomagają mi odświeżyć sobie budowę ekranu, gdy po dłuższej przerwie wracam do jakiegoś kodu na ZX. Jak widać, adres startowy bufora grafiki to $4000. Gdy dodamy do niego 3*2048 bajtów (czyli nasze 3 obszary ekranu) to otrzymamy $5800, czyli dokładnie adres początku bufora atrybutów kolorów. Oznacza to, że bufor atrybutów znajduje się dokładnie za buforem graficznym ekranu. Jeśli więc w przypadku jakiegoś niechcianego błędu nasz kod zacznie „rysować" poza pamięcią przeznaczoną na obszar graficzny, to trafimy zapewne właśnie w atrybuty kolorów i na ekranie pokażą się dość przypadkowe i niechciane kolorowe kleksy. Z drugiej jednak strony, taką organizację pamięci w ZX Spectrum da się wykorzystać. Tak się składa, że dokładnie taki właśnie format (6144 bajty bitmapy ekranu + 768 bajtów atrybutów kolorów) ma spectrumowy plik graficzny o rozszerzeniu .SCR. Format ten nie ma żadnej kompresji, jest więc dokładnym zrzutem pamięci, czyli kopią pamięci zapisaną w pliku. Nie jest to może zbyt oszczędny format, bo prawie 7kB na jeden obrazek, przy 48kB całej pamięci RAM to jednak sporo (zazwyczaj trzyma się je spakowane aż do momentu użycia), ale jego plusem jest prostota, bo da się obrazek w nim zawarty wyświetlić jedną prostą pętlą, przepisującą dany obrazek bezpośrednio w pamięć ekranu i w atrybuty. Z taką wiedzą i kodem z przykładów je-

steśmy w stanie napisać program wyświetlający nasz własny rysunek na ekranie, jak również „zasymulować loading screen" nieistniejącej gry (może Quake albo Mortal Kombat na ZX48k?). Zanim jednak przejdziemy do programowania wyświetlania, poznamy kolejny ważny aspekt tworzenia dem – przygotowywanie danych.

Główne elementy, z których składa się typowe intro/demo czy gra, to zazwyczaj kod, muzyka i grafika. Jeżeli chodzi o te dwa ostatnie składniki, to raczej używa się jakichś narzędzi czy edytorów, by je stworzyć, niż pisze kod, który je generuje (choć oczywiście jest to możliwe i stosowane). Dlatego ważną sprawą dla każdego kodera jest poznanie pecetowych lub spectrumowych narzędzi, przy pomocy których uda się łatwo i przyjemnie stworzyć fajne „multimedia" do wykorzystania w produkcji. Zapewne jest wiele narzędzi, w których można przygotować grafikę dla „Gumiaka", ale kiedy ja poszukiwałem czegoś użytecznego do takiego celu, w internecie znalazłem polecany program *MultiPaint*. Spodobało mi się w nim to, że jest w stanie wspierać przygotowywanie grafiki na wiele platform (2 tryby graficzne ZX: Spectrum/ULA+, 2 tryby C64, 2 tryby C=Plus4, MSX1, Amstrad CPC a nawet Sinclair QL) łącznie z ich paletami, i pilnować ich ograniczeń sprzętowych (pomocne gridy), to jeszcze wygląda jak mój ulubiony *Deluxe Paint II* lub *III* z Amigi, czyli w mojej opinii najlepiej zrealizowany edytor do amatorskiego pixelowania. *MultiPaint* w trybie ZX Spectrum ma dostępny panel narzędzi do rysowania, łatwo dostępną paletę, pomocny grid 8x8 (pilnujący ograniczeń kolorów) oraz tryb importu i exportu wspomnianych wcześniej plików .SCR, czyli właśnie to, czego nam potrzeba. (▼)



Przygotujmy więc, przy jego pomocy, jakiś całoekranowy obrazek, po czym przy użyciu menu: „FILE/Export .scr" zapiszmy plik .SCR, który wyświetlimy naszym kodem wykonywalnym z pliku .TAP. Na potrzeby tego artykułu pozwoliłem sobie ściągnąć z internetu (w formacie .png) obrazek z legendarnej gry *Exolon* i przerobić go lekko, by bardziej pasował do treści.



Gotowy obrazek zapisujemy w tym samym folderze, w którym trzymamy nasz kod, by uniknąć podawania ścieżek do pliku. Oczywiście przy większych produkcjach warto stosować np. osobne podfoldery dla grafik, dźwięków czy nawet osobnych plików z kodem, ale tym możemy zająć się wtedy, gdy będzie taka potrzeba. Na razie – im prościej, tym lepiej. Obrazek zapisujemy jako .png (do celów ewentualnych późniejszych poprawek czy szybkiego podglądu w Windows) oraz koniecznie eksportujemy, czyli zapisujemy do formatu .scr (screen). Dla ułatwienia obrazek nazwałem sobie ExolonZB.scr. Teraz możemy „wciągnąć" go do kodu programu i wyświetlić na ekranie. Do wczytywania plików binarnych do programów pisanych w asemblerze używamy kolejnej niezwykle przydatnej dyrektywy (inaczej „instrukcji") dla programu tłumaczącego (asemblera) *Pasmo*, czyli INCBIN. Dyrektywa ta po prostu wczytuje jakikolwiek wskazany plik (bo plikiem binarnym może być muzyka, grafika, tablica wartości sinusa, tekst scrolla czy nawet inny kawałek kodu programu) i umieszcza go w miejscu swojego (czyli dyrektywy INCBIN) wystąpienia. Aby później wygodnie odszukać nasz obrazek, zazwyczaj przed dyrektywą INCBIN dodaje się jakąś etykietę po to, by czytelnie opisywała jakie dane za nią leżą. Dodatkowo, warto też skorzystać ze znanej nam już dyrektywy `ORG Adres`, gdyż wtedy dyrektywa INCBIN wczyta nam nasz obrazek dokładnie w to miejsce w pamięci, w które chcemy. Może w tym przykładzie nie ma to wielkiego znaczenia, ale pisząc intro/demo/grę na dość ograniczoną ilość pamięci RAM jaką jest 48kB warto mieć dokładną kontrolę nad tym, gdzie które dane rozłożymy. Czasem też ułożenie danych pod konkretnym i „okrągłym" adresem może przyspieszyć dostęp do nich, ale to już temat na bardziej zaawansowany odcinek tego mini-kursu. Zmodyfikujmy więc poprzedni przykład do postaci takiej jak poniżej i przyjrzyjmy mu się, gdyż doszło kilka ciekawych elementów.

```
Demo_Start:
  ei                            ; enable interrupts
  ld   hl,GFX_SCR_START         ; pixel screen
                                ; buffor start
                                ; address
  ld   bc,My_picture            ; memory address of
                                ; my picture
  ld   de,32*192+768            ; let DE is our
                                ; loop counter
My_loop:
  halt                          ; HALT - let's wait
                                ; to start a new
                                ; frame
  ld   a,(bc)
  inc bc
  ld   (hl),a                   ; put a byte from
                                ; reg. B to the
                                ; screen and show
                                ; some pixels
  inc hl                        ; add 1 to the
                                ; address so it
                                ; points to another
                                ; byte
  dec de                        ; substract 1 from
                                ; our loop counter

  ld   a,d
  or   e
  jr   nz,My_loop               ; if counter is
                                ; bigger than 0, do
                                ; another run

;--- infinite loop to avoid BASIC prompt
Wait:
  jr   Wait
;-----------------------------

  ORG $a000
My_picture:
```

```
  INCBIN            ExolonZB.scr

Demo_Stop:
```

Zacznijmy od końca. Jak wspominałem, pojawiła się dyrektywa ORG, która sprawi, że występująca po niej etykieta `My_picture` będzie znajdowała się pod adresem $a000. Tym samym wymusiliśmy na asemblerze *Pasmo*, by stworzony przez niego wykonywalny plik .tap ładując dane obrazka do pamięci umieścił je od adresu $a000. Etykieta `My_picture` w zasadzie nie jest teraz konieczna (bo wyświetlając obrazek moglibyśmy posłużyć się po prostu adresem $a000), ale takie opisowe oznaczenie położenia danych w pamięci to dobra i przydatna rzecz, kiedy danych będzie o wiele więcej (np. kolejnych obrazków, fontów i innych elementów graficznych). Łatwiej będzie nam zapamiętać gdzie i co zostało rozłożone w pamięci a czytelność kodu zyska na tym zdecydowanie. Mamy więc dyrektywę ORG $a000, etykietę `My_picture` oraz dyrektywę INCBIN, która to właśnie „wciągnie" nasz obrazek pod wskazany przez nas adres pamięci. Tym sposobem mamy dane obrazka przygotowane do użycia przez nasz kod.

Bezpośrednio przed operacjami wczytania obrazka, mamy pętlę `Wait`. Jest to „pętla nieskończona", ponieważ jedynym jej rozkazem i powodem działania jest skok do samej siebie. Wykorzystujemy ją, by zakończyć działanie programu i nie pozwolić na wyjście do BASICa, gdyż jego prompt zamazałby nam dolną część obrazka. Najciekawsze rzeczy znajdują się jednak na początku programu. Jak widać licznikiem wykonań pętli, czyli licznikiem bajtów do przepisania z obszaru pamięci zajmowanej przez wczytany obrazek, do pamięci ekranu i bufora atrybutów kolorów jest nadal para rejestrów DE, z tym, że teraz jego wartością początkową jest suma liczby wszystkich „graficznych" (bitmapowych) bajtów ekranu (192 linie po 32 bajty w każdej) oraz całego bufora atrybutów kolorów, czyli 24 linie „pixeli kolorów 8x8", po 32 atrybuty każda. Łącznie daje to 6912 bajtów i zgadza się dokładnie w wielością pliku ExolonZB.scr (jak widać, wszystko układa się w logiczną całość). Do pary rejestrów BC ładujemy natomiast adres naszego obrazka, czyli wykorzystujemy etykietę `My_picture` (tu widać wspomnianą lepszą czytelność kodu w porównaniu do bezpośredniego zastosowania adresu $a000). Przed wejściem w pętlę mamy więc ustawiony licznik na odpowiednią liczbę bajtów do skopiowania (rejestry DE), adres początku bufora ekranu w rejestrach HL oraz adres początku danych obrazka w rejestrach BC – możemy więc wystartować nasze zadanie.

Po wejściu w pętlę wykona się najpierw rozkaz HALT, a potem pierwszy bajt obrazka zostanie załadowany do rejestru A (Akumulatora). Następnie zwiększymy sobie adres zawarty w BC, by przy kolejnym wykonaniu „obrotu" pętli, wskazywał już kolejny bajt obrazka do pobrania. Dalej zawartość rejestru A przepisujemy w pierwszy bajt bufora graficznego ekranu (i pierwsze pixele pojawiają się w tym momencie na ekranie) po czym zwiększamy adres w HL o jeden, by kolejny obrót pętli wypełniał pixelami kolejny bajt ekranu. Po tej operacji zmniejszamy liczbę bajtów do skopiowania na ekran o jeden i sprawdzamy, czy wszystkie bajty zostały już przepisane. Jeśli licznik jest różny od zera (a więc jeszcze jakieś bajty zostały do przerzucenia na ekran) to pętla wykona się kolejny raz. Jeżeli jednak w liczniku mamy zero, to znaczy, że wszystkie bajty zostały już skopiowane i na ekranie powinien być widoczny cały obrazek. W tym momencie nasza główna pętla programu kończy się i procesor przechodzi do kolejnej instrukcji, którą jest rozkaz `JR Wait`, realizujący naszą pętlę nieskończoną. Tym samym osiągnęliśmy zakładany rezultat. Oczywiście polecam modyfikacje kodu, takie jak choćby zakomentowanie instrukcji HALT, zmiana licznika, początku przerzucania danych czy zmiana adresu docelowego (wczytanie części obrazka w inny fragment ekranu). Ciekawym ćwiczeniem może być wpisanie obrazka w pamięć ekranu „od końca". Tym sposobem, najpierw zostaną ustawione atrybuty kolorów, a dopiero po nich zaczną pojawiać się bitmapowe dane graficzne. By tego dokonać, należy oczywiście adresy zawarte w rejestrach BC i HL przenieść na koniec obrazka oraz buforów ekranu (łącznie z buforem atrybutów) oraz zmniejszać je (używając instrukcji DEC zamiast INC) po każdym przepisanym bajcie. Zmiana adresów w taki sposób, by wskazywały koniec buforów ekranu czy obrazka jest prosta – należy po prostu dodać wielkość obrazka (w bajtach) zarówno do adresu początku obrazka, jak i do początku adresu buforów graficznych. Tak więc dodajemy do nich 32*192+768 i wygląda to tak:

```
  ld   hl,GFX_SCR_START + 32*192+768
  ld   bc,My_picture + 32*192+768
```

Wracając jednak do pierwszej wersji wyświetlania obrazka, warto wspomnieć, że tą samą pętlę można zapisać krócej, należy jednak zapoznać się z kolejnymi instrukcjami procesora Z80, czyli LDI oraz LDIR. Dokładny opis ich działania zamieścił Tygrys w swoim artykule i po raz kolejny tam właśnie odsyłam zainteresowanych. W skrócie, LDI przepisuje bajt bezpośrednio spod adresu wskazywanego przez parę HL pod adres wskazywany przez parę DE, bez pośredniego użycia rejestru Akumulatora oraz automatycznie inkrementując (powiększając o jeden) oba adresy (czyli odpada potrzeba użycie instrukcji INC DE i INC HL). Dodatkowo, instrukcja LDI sama zmniejsza licznik bajtów w BC, więc dzięki niej zrealizujemy niemal całą pętlę i do dodania zostanie tylko sprawdzenie czy licznik w BC jest równy zero, by pętla zakończyła się. Poniżej krócej zapisany przykład – zwracam uwagę na zmianę rejestrów adresów i licznika! Instrukcja LDI wymaga konkretnych adresów w konkretnych rejestrach, a więc w rejestrach HL adres początku obrazka (czyli skąd mają być pobierane dane do przepisania) a w rejestrach DE adres buforów ekranu (czyli dokąd dane mają trafić). BC to wspomniany wcześniej licznik. Należy tego przestrzegać, bo Z80 ma wiele instrukcji, które wymagają konkretnych rzeczy w ściśle określonych rejestrach a pomyłki z tym związane są niezwykle frustrujące i niełatwe do wykrycia podczas poprawiania źle działającego programu. Polecam zawsze trzymanie dobrej „rozpiski" rozkazów Z80 w pobliżu i częstą weryfikację tego, co właśnie zapisujemy, bo nie zawsze asembler zgłosi nam błąd (np. jeśli pomylimy, co jest źródłem danych a co adresem docelowym, wszystko ładnie się skompiluje i tylko program zadziała niezgodnie z jego oczekiwanymi założeniami). Nowa wersja pętli wygląda właśnie tak:

```
Demo_Start:
  ei                ; enable interrupts
  ld   de,GFX_SCR_START ; pixel screen
                    ; buffor start address
  ld   hl,My_picture  ; memory address
                    ; of my picture
  ld   bc,32*192+768  ; let BC is our loop
                    ; counter
My_loop:
  halt              ; HALT - let's wait to
                    ; start a new frame
  ldi               ; copy byte from (HL)
                    ; to (DE), decrement BC
  ld   a,b
  or   c            ; is counter in BC
                    ; empty already?
  jr   nz,My_loop   ; if counter is bigger
                    ; than 0, do another run

  ; infinite loop to avoid BASIC prompt
Wait:
  jr   Wait
;---------------------------
  ORG $a000
My_picture:
  INCBIN            ExolonZB.scr

Demo_Stop:
```

Dodam jeszcze, że gdyby chodziło nam wyłącznie o wrzucenie obrazka na ekran, bez opóźniania instrukcją HALT i przepisywania po jednym bajcie – można by użyć instrukcji LDIR i jeszcze skrócić naszą pętlę (praktycznie do jednej instrukcji), ale to już zostawię Czytelnikom, których zachęcam do przeprowadzenia własnych eksperymentów. ■

W tym artykule to wszystko, co planowałem zaprezentować. Mam nadzieję, że poruszane tematy są zrozumiałe i przystępnie opisane. Tradycyjnie proszę o opinie na mail: sachozol@op.pl. W kolejnym artykule zajmiemy się zapewne scrollami, polecam więc zaznajomienie się z dwoma świetnymi artykułami opublikowanymi na portalu speccy.pl w dziale „Artykuły/Programowanie" (chodzi o „Oldschool demomaking, cz.1 i 2"), których autorem jest legenda sceny ZX Spectrum w Polsce, czyli Mat/ESI.

Assembler Z80
od postaw dla prawie każdego

TYGRYS **CZĘŚĆ 2**

W poprzedniej części kursu zajmowaliśmy się podstawami. Podstawy wystarczyły, aby napisać banalny efekt polegający na zmianie kolorów na ekranie. Dziś poznamy kolejne instrukcje asemblera Z80 oraz wykorzystamy je, aby napisać kolejny program. Tym razem będzie to tester joysticków. Po raz kolejny program będzie dedykowany ZX Spectrum, ale nic nie stoi na przeszkodzie, aby napisać podobny program dla innych systemów. Tak jak poprzednim razem, tak i tym, przedstawię jedynie niezbędną wiedzę do napisania programu. Kolejne przydatne informacje znajdą się w następnej części kursu, gdzie wykorzystamy je w kolejnym programie.

OPERACJE NA BITACH

`BIT, SET, RES`

Do operacji na poszczególnych bitach służą instrukcje `BIT`, `SET` i `RES`. Przyjmują dwa argumenty, pierwszy to numer bitu (numerowany od 0 do 7, przy czym numer 0 jest pierwszy od prawej strony), na którym należy wykonać operację, drugim zaś jest podmiot działania – rejestr ośmiobitowy lub wskaźnik pamięci (HL) albo (IX+index)

Kolejność bitów w bajcie:

7	6	5	4	3	2	1	0

`BIT` służy do testowania stanu konkretnego bitu we wskazanym rejestrze. Wartość bitu jest kopiowana do flagi **Z** w rejestrze statusu **F**. Oto kilka przykładów zapisu:

```
BIT 0,a          ; testuje bit 0 w rejestrze A
BIT 2,c          ; testuje bit 2 w rejestrze C
BIT 4,(IX+10)    ; testuje bit 4
                 ; w zapisany w pamięci
                 ; o adresie wynikającym
                 ; z sumy IX+10
```

Tak zbadany bit można wykorzystać w instrukcji warunkowej, jak w poniższym przykładzie:

```
LD a,%0000001    ; bit zerowy ustawiony na 1
BIT 0,a          ; sprawdzenie stanu bitu 0
                 ; skok, jeżeli ma wartość 1
JR NZ,flaga_Z_UstawionaNaJeden
```

Do ustawiania, to znaczy przypisania wartości 1 konkretnemu bitowi służy instrukcja `SET`.

```
SET 7,c          ; ustawia bit 7 na 1
                 ; w rejestrze C
SET 0,(hl)       ; ustawia bit 0 na 1
                 ; w komórce pamięci
                 ; wskazywanej przez HL
LD a,0
SET 0,a          ; po tej operacji
                 ; A będzie mieć wartość %00000001
```

`RES` to instrukcja przeciwna to `SET`. Resetuje (ustawiając na 0) zadany bit we wskazanym rejestrze. Przykład:

```
LD a,%00000001
RES 7,a     ; A zmieni wartość
            ; z 1 na 129 (bin: %10000001)
```

W ZX BASIC nie ma instrukcji operujących na poszczególnych bitach. Pewną analogią mogą być instrukcje do stawiania pixeli na ekranie, gdzie odpowiednikiem `BIT` mógłby być `POINT (x,y)`, `SET` zaś byłby `PLOT x,y`, zaś bliski `RES` byłby `PLOT INVERSE 1; x, y`.

OPERACJE I/O

Podstawowym sposobem komunikacji z urządzeniami zewnętrznymi są operacje wejścia/wyjścia. Komunikacja odbywa się poprzez porty, które można traktować podobnie jak adresację pamięci – tyle że odnoszącą się do urządzeń zewnętrznych. Możliwe jest „adresowanie" portu 8 i 16 bitowego. Służą do tego instrukcje `IN` oraz `OUT`. Choć ten 8 bitowy port jest tak naprawdę 16 bitowy, zaś adresowanie 8 bitowe odnosi się do młodszego bajtu adresu, starszy zaś dostarczany jest przez Akumulator, o czym będzie jeszcze wzmianka poniżej. Obsługa odwołań oraz pełnego/niepełnego adresowania leży w gestii sprzętu, niemniej jednak nie jest to temat niniejszego artykułu.

`IN` służy do odczytania wartości z zadanego portu i umieszczenie jej w rejestrze. Numer portu można podać jako argument dla instrukcji – odczytana z portu wartość zapisywana jest w Akumulatorze:

```
IN a,(port 8bit)
```

Zatem

```
IN a,($1F)       ; odczytanie wartości
                 ; z portu Kempstona
```

Odpowiednikiem w BASIC jest na przykład:

```
LET a = IN 31 : REM odczytanie wartości z portu Kempstona (hex: $1F)
```

Za pomocą `IN a,(nn)` da się również odczytywać wartości z 16-bitowych portów. Wtedy w rejestrze A zapisuje się starszy bajt numeru portu, zaś w argumencie dla `IN` - młodszy. Poniższy przykład prezentuje jak odczytać rząd klawiszy, w którym znajduje się klawisz „1". Adresem portu jest $F7FE (dziesiętnie: 63486):

```
LD a,$F7
IN a,($FE)
```

Odpowiednikiem tego w BASIC byłoby:

```
LET a=IN 63486
```

Istnieje bardziej elastyczna forma `IN`, która 16 bitowy adres portu czerpie z rejestru **BC**, zaś wynik potrafi przekazać do dowolnego rejestru ośmiobitowego (A,B,C,D,E,H,L):

```
IN r,(c)
```

Zapis `(c)` może być mylący – należy zapamiętać go jako wyjątek. Osoby zainteresowane, dlaczego tak jest odsyłam do oficjalnej dokumentacji procesora Z80 (link na końcu artykułu).

Powyższy przykład można zapisać jako:

```
LD bc,$F7FE
IN h,(c)
```

Co w wersji dla BASIC wyglądałoby:

```
LET bc=63486
LET h=IN bc
```

`OUT` z kolei służy do wysłania wartości na zadany port. Tu, podobnie jak dla `IN`, możemy bezpośrednio zaadresować port 8-bitowy lub, poprzez rejestr BC, 16 bitowy. Analogicznie zatem do `IN`

```
OUT (port 8bit),a
```

wyśle na port o podanym numerze zawartość rejestru A. Tu też zachodzi taka sama prawidłowość z adresowaniem pośrednim przez wartość starszego bajtu w Akumulatorze, jak w instrukcji `IN`. Zwykle, w związku ze sposobem obsługi sprzętowej portów, ten fakt można pominąć (dla dociekliwych – ten sam port może inaczej się zachowywać kiedy wysyłamy do niego dane, a inaczej, jeżeli je odczytujemy, patrz port $FD).

Dla przykładu, instrukcja która w ZX Spectrum zmieni kolor ramki na czerwony:

```
LD a,2
OUT ($FE),a      ; zmiana koloru ramki
                 ; na czerwoną
```

W BASICu będzie miała postać:

```
OUT 254,2
```

Analogicznie, podając w rejestrze BC, 16 bitowy zapis portu, osiągniemy to samo poprzez:

```
LD bc,$00FE
LD h,2
OUT (c),h
```

Rejestr H został wybrany jako przykład – można zastosować dowolny inny: A,B,C,D,E,H,L. Korzystając z faktu, że adres portu mieści się w ośmiu bitach, możemy zmodyfikować powyższy przykład tak, aby użyć rejestru B, zamiast H. Jest to z pewnością element optymalizacji miejsca (2 bajty mniej kodu) Oto jak to może wyglądać:

```
LD bc,$02FE
OUT (c),b
```

PODSTAWOWE OPERACJE NA BLOKACH PAMIĘCI

Jednym z „magicznych" elementów Z80 są instrukcje blokowe, dzięki którym możliwe jest kopiowanie bloków pamięci, przesyłanie danych z pamięci do portu 8bit, z portu do pamięci oraz przeszukiwanie pamięci w poszukiwaniu wzorcowego bajtu. My jednak zajmiemy się najbardziej powszechnym przypadkiem – przesyłaniem bloków danych. Dziś przyjrzymy się instrukcjom `LDI` oraz `LDIR`.

Zacznijmy jednak od programu w BASIC, który skopiuje nam kilkaset bajtów z jednego obszaru pamięci do drugiego:

```
10 LET hl=1000 :
 LET de=16384 :
 LET bc=300
20 POKE de,PEEK hl :
 LET hl=hl+1 :
 LET de=de+1 :
 LET bc=bc-1
30 IF bc<>0 THEN GO TO 20
```

W zmiennej hl przechowujemy adres pamięci, z której będziemy kopiować dane, w de – adres docelowy zaś bc zawiera licznik powtórzeń do wykonania. Instrukcja PEEK odczytuje bajt ze wskazanego adresu pamięci, zaś POKE wpisuje podaną wartość do wskazanego adresu pamięci. Po skopiowaniu bajtu, następuje zwiększenie wartości zmiennych hl i de o jeden, zaś wartość bc jest pomniejszona o 1. Operacja będzie powtarzana dopóki wartość bc jest różna od 0.

Odpowiednikiem linii 20 i 30 BASICa jest operacja `LDIR` (**L**oa**D** **I**ncrement **R**epeat).

```
LD hl,1000
LD de,16384
LD bc,300
LDIR
```

Podobna do LDIR instrukcja `LDI` różni się tym, że nie powtarza operacji. Dzięki temu przerzucanie danych jest szybsze (wewnętrzna „pętla" w LDIR „zjada" dodatkowy czas procesora) choć kod zajmuje znacznie więcej miejsca w pamięci.

Przenoszenie pamięci od początku bloku jest wygodne, niemniej czasami trzeba przenieść dane zaczynając od jego końca. Służy do tego instrukcja `LDDR` (**L**oa**D** **D**ecrement **R**epeat). Różnica w stosunku do LDIR jest taka, że wartości rejestrów HL i DE są zmniejszane o 1.

(PIERWSZY) PROGRAM NARZĘDZIOWY

Po tym, jak poznaliśmy kilka dodatkowych instrukcji, możemy napisać program narzędziowy. Pomoże nam w diagnozowaniu działania interfejsów joysticków. Czy to możliwe z tak małą liczbą instrukcji? Przekonajmy się…

Tester joysticków to program, który pokazuje aktualnie wybrany stan urządzenia typu joystick. Do ZX Spectrum podłącza się co najmniej trzy typy – Sinclair I, Sinclair II i Kempston. Oba joye typu Sinclair są podpięte pod jeden z rzędów klawiatury i odczytuje się je

tak samo jak klawisze. Dla każdego typu manipulatora napisana jest oddzielna procedura odczytująca i analizująca otrzymane dane. Każda z nich jest napisana inaczej – tak aby pokazać możliwe sposoby analizy otrzymanych danych.

Założeniem programu jest odczyt wartości dla poszczególnych joysticków a następnie prezentacja ich w czytelny sposób. Komunikaty tekstowe byłyby zapewne wystarczające, jednak zdecydowałem się proces uatrakcyjnić i wyświetlane będą duże ikonki przy wykorzystaniu mapy atrybutów.

```
; pozycja konkretnych
; obrazków-kierunków na ekranie
ATTR_GORA equ 22528 + 0*32 + 12
ATTR_LEWO equ 22528 + 7*32 + 5
ATTR_PRAWO equ 22528 + 7*32 + 19
ATTR_FIRE equ 22528 + 7*32 + 12
ATTR_DOL equ 22528 + 14*32 + 12

LICZBA_TYPOW_JOY: equ 3

; ZX Spectrum ROM - procedury
CLEAR equ $0DAF      ; THE ,CLEARING THE
                     ; WHOLE DISPLAY'
                     ; SUBROUTINE
CHAN_OPEN equ $1601  ; THE ,CHAN-OPEN'
                     ; SUBROUTINE
PRINT_CHAR equ $10   ; THE ,PRINT A
                     ;CHARACTER' RESTART
; ZX Spectrum - zmienne systemowej
LAST_K equ $5C08     ; LAST-K - ostatni
                     ; wciśnięty klawisz
ATTR_P equ $5C8D     ; ATTR-P - kolor dla
                     ; obszaru ekranu
```

Powyżej znajduje się blok z definicją stałych. Zaletą użycia stałych jest to, że gdy zmieniamy wartość w jednym miejscu, zostanie ona użyta w całym programie. Oszczędzamy czas nie tracąc go na zmienianie wartości w kilku miejscach w kodzie. Tak więc zgromadziliśmy w jednym miejscu adresy pod jakimi mają się wyświetlać informacje dotyczące poszczególnych kierunków (zmienne ATTR). Zdefiniowaliśmy stałą `LICZBA_TYPOW_JOY`, która zawiera liczbę obecnie wspieranych rodzajów manipulatorów oraz adresy procedur w ROM, które zostaną wywołane aby poprawnie zainicjować działanie programu.

```
 org 25000
start
                ; inicjowanie ekranu
ld a,7          ; białe pixele, tło czarne
ld (ATTR_P),a   ; ustalenie koloru
call CLEAR      ; wyczyszczenie ekranu
ld a,0          ; 0 = czarny kolor ramki
out ($FE),a     ; wyślij daną na port $FE
ld a,2          ; 2 = górny ekran
call CHAN_OPEN  ; konfiguracja kanału dla PRINT
                ; wyświetlanie napisów
                ; informujących o trybie pracy
ld de,_napis_pomoc   ; adres tekstu do
                     ; rejestru DE
call wyswietl_napis  ; wyświetlenie
                     ; napisu z linią
                     ; pomocy
call wyswietl_typ    ; napis z typem joya
```

Inicjalizacja programu to blok, który powinien znajdować się w każdym programie. Tutaj następuje ustawienie zmiennych systemowych, wywołanie procedur czyszczenia ekranu oraz ustawiania odpowiednich kanałów systemowych. Posłużą nam one do wyświetlania napisów przy użyciu procedur systemowych. W drugiej części bloku następuje wywołanie procedur, których opis będzie można znaleźć poniżej.

```
; glowna petla programu
 _glowa_petla
 halt                        ; oczekiwanie
                             ; na początek
                             ; ramki ekranu
 call resetuj_stany  ; zaczynamy
                             ; inicjalnym
                             ; stanem kierunków
 call zmiana_konfiguracji    ; obsługa zmiany
                             ; konfiguracji
 call odczyt_portow  ; odczyt portów
                             ; w zależności
                             ; od konfigu
 call wyswielenie_stanow     ; wyświetla
                             ; aktualnego
                             ; stanu kierunków
 jr _glowa_petla     ; skok do
                             ; początku pętli
```

Pętla główna jest odwołaniem do procedur, które powinny być wywołane po to, aby program mógł realizować swoje zadanie. Instrukcja `HALT` czeka na przerwanie, po którym, co 1/50 sekundy, uruchamiane są procedury systemowe, chociażby czytające klawiaturę, z czego będziemy korzystać chwilę później. Następnie wykonujemy resetowanie stanów kierunków, procedury reagujące na klawisz zmiany konfiguracji, odczyt wartości z konkretnych portów oraz ich wyświetlanie. Zamknięcie pętli następuje poprzez skok bezwarunkowy do jej początku, gdzie ponownie następuje czekanie na przerwanie. Bez tego oczekiwania pętla wykonywałby się, zupełnie niepotrzebnie, kilkaset razy na sekundę.

```
 ; procedura zerująca stan użycia poszczególnych kie-
runków
 resetuj_stany
  xor a              ; zerowanie akumulatora,
                     ; odpowiednik LD a,0
  ld (_stan_gora),a  ; po kolei
                     ; ustawienie
                     ; inicjalnej
                     ; wartości
  ld (_stan_dol),a   ; dla każdego z kierunków
  ld (_stan_lewo),a
  ld (_stan_prawo),a
  ld (_stan_fire),a
  ret                ; oraz powrót
                     ; do miejsca wywołania
```

Powyższa procedura wpisuje zera w komórki pamięci przechowujące stany kierunków. Zero oznaczać będzie nieaktywny kierunek, zaś wartości różne od zera – aktywny.

```
; obsługa zmiany konfiguracji
; korzystamy ze zmiennej systemowej LAST K
zmiana_konfiguracji
 ld a,(LAST_K)       ; odczytaj ostatnio
                     ; wciśnięty klawisz
 cp ,z'              ; czy to jest 'z' ?
 ret nz              ; nie był to 'z'
                     ; - wyjście z procedury
 xor a               ; szybkie zerowanie
                     ; rejestru A
 ld (LAST_K),a       ; zerowanie zmiennej
```

```
                    ; systemowej
 ld a,(_typ_joya)    ; odczytanie aktualnej
                     ; wartości typu joya
 inc a               ; zwiększenie licznika
 cp LICZBA_TYPOW_JOY ; wartość
                     ; maksymalna?
 jr nz,_zmiana_konfiguracji_omin
                     ; nie, przejdź dalej
 ld a,0              ; tak - zeruj licznik
_zmiana_konfiguracji_omin:
 ld (_typ_joya),a ; zapisanie
                     ; aktualnej wartości A
 call wyswietl_typ   ; wyświetl aktualny
                     ; tym joya
 ret
```

Podprogram zmiany konfiguracji najpierw sprawdza czy został naciśnięty klawisz „Z". Jeżeli „Z" nie został naciśnięty, procedura wraca do miejsca z którego została wywołana (instrukcja `RET`). Gdy jednak „Z" zostanie wciśnięty następuje zwiększenie licznika nazwanego „`_typ_joya`". To nic innego jak indeks wskazujący na wybrane urządzenie. Gdy indeks przekroczy wartość graniczną, jest zerowany. Wynikowa wartość jest zapisywana ponownie do zmiennej. Ostatnim elementem jest wywołanie procedury wyświetlającej zaktualizowany typ urządzenia.

```
; procedura wyświetlająca napis z typem urządzenia
wyswietl_typ
 ld de,_napis_typ       ; adres napisu 'Typ: '
 call wyswietl_napis    ; wyświetlenie
                        ; na ekranie
; wyświetlenie napisu z wybranym typem joya
 ld a,(_typ_joya)       ; pobranie wartości
                        ; ze zmiennej
                        ; obliczanie adresu
                        ; wskaźnika do napisu
 add a,a                ; mnożenie *2 jako
                        ; dodawanie A+A
 ld l,a                 ; przekazanie
                        ; wyniku mnożenia
 ld h,0                 ; zerowanie
                        ; starszego bajtu
 ld bc,_napisy  ; do BC adres tablicy
                        ; wskaźników napisów
 add hl,bc              ; dodanie wartości,
                        ; w HL wyliczany adres

 ld e,(hl)       ; pobranie
                        ; młodszego bajtu
 inc hl                 ; zwiększenie adresu
                        ; o jeden...
 ld d,(hl)       ; do starszego bajtu
; finalnie - w DE - adres napisu
 jp wyswietl_napis      ; skok do procedury
                        ; drukującej
```

Ten fragment programu odpowiada za wyświetlenie informacji o aktualnym typie joysticka. Jako pierwsze następuje wyświetlenie tekstu „Typ: ". Kolejnym krokiem jest pobranie numeru obsługiwanego urządzenia, a następnie obliczenie adresu w pamięci, pod którym znajduje się wskaźnik do interesującego nas napisu. Wiedząc że mamy tablicę adresów 16 bitowych, a każdy taki adres zajmuje 2 bajty, możemy łatwo obliczyć adres przesunięcia względem początku tablicy. Wystarczy, że mamy jego indeks, który należy pomnożyć przez 2. Mnożenie przez 2 można zrealizować poprzez dodanie tej samej wartości do siebie (X+X=2X) lub przesunięcie bitowe. Na ten moment wiemy jak dodawać, więc z tego skorzystamy. Do tak obliczonej wartości dodajemy adres początku tablicy adresów, a następnie odczytujemy dane – najpierw młodszy bajt a następnie starszy (w takiej kolejności zapisywane są liczby 16 bitowe dla Z80). Mając adres, wykonujemy skok do procedury wyświetlającej napis.

```
 ; procedura odczytująca wartości portów
 ; konfiguracja wskazuje jaką
 ; procedurę obsługującą konkretny
 ; typ urządzenia należy wywołać
 odczyt_portow
  ld a,(_typ_joya)     ; odczytanie
                              ; typu joya
  cp 0                        ; 0 = kempston
  jp z,odczytaj_kempston      ; kempston - przeskocz
do procedury
  cp 1
  jp z,odczytaj_sinclair1     ; sinclair1 - wywołanie
podprogramu
  cp 2
  jp z,odczytaj_sinclair2     ; sinclair2 - przejście
do procedury
  ret                         ; jeżeli wartość
                              ; jest inna
                              ; niż znane
                              ; to powróć do
                              ; miejsca wywołania
```

Zakładamy że mamy wybrane urządzenie. Kolejnym krokiem jest jego obsługa w programie. Procedura pobiera numer urządzenia, a następnie porównuje numery i jeżeli się zgadzają przekazuje działania do właściwego podprogramu.

```
 ; bit 0: prawo
 ; bit 1: lewo
 ; bit 2: dół
 ; bit 3: góra
 ; bit 4: fire 1
 odczytaj_kempston
  in a,($1F)      ; odczyt wartości
                  ; z portu $1F
  ld b,a          ; skopiowanie jej
                  ; do rejestru B
  cp 255          ; jeżeli zwróconą
                  ; wartością jest 255
  ret z           ; to znaczy że
                  ; nie wykryto interface
  ld a,1          ; do rej. A wartość
                  ; różna od 0

  bit 0,b         ; sprawdzamy bit 0 w rej. B
  jr z,_odczytaj_kempston_1   ; jeżeli ma
                      ; wartość '0' to przeskocz
  ld (_stan_prawo),a  ; zapisanie
                      ; stanu kierunku
 _odczytaj_kempston_1
  bit 1,b             ; sprawdzamy bit 1 w rej. B
  jr z,_odczytaj_kempston_2   ; jeżeli bit
                              ; zgaszony
                              ; przeskocz
  ld (_stan_lewo),a
 _odczytaj_kempston_2
  bit 2,b ; sprawdzamy bit 2 w rej. B
  jr z,_odczytaj_kempston_3   ; jeżeli bit
                              ; zgaszony
                              ; to przeskocz
```

```
 ld (_stan_dol),a
_odczytaj_kempston_3
 bit 3,b                    ; sprawdzamy bit 3
                            ; w rej. B
 jr z,_odczytaj_kempston_4  ; jeżeli
                            ; bit zgaszony
                            ; to przeskocz
 ld (_stan_gora),a
_odczytaj_kempston_4
 bit 4,b            ; sprawdzamy bit 4 w rej. B
 ret z              ; jeżeli bit zgaszony
                    ; to wyjdź z procedury
 ld (_stan_fire),a  ; zapis informacji
                    ; o aktywnym stanie FIRE
 ret
```

Od tego miejsca zaczynają się procedury odczytu wartości z dedykowanych portów oraz ich analizy. Każdy typ joysticka obsługiwany jest przez oddzielny podprogram, ponieważ inne bity odpowiadają ich stanom/kierunkom. Specjalnie przedstawiłem kilka sposobów na realizację tego samego zadania. Do czytelnika należy ocena, która z nich jest dla niego czytelna i najbardziej mu odpowiada. Oczywiście nie są to wszystkie metody, a jedynie te najprostsze. Poniżej znajduje się odpowiednik takiej procedury w BASICu, choć niestety nie zadziała on na ZX BASIC, gdyż nie posiada on operacji bitowych:

```
10 LET v=IN port
20 IF NOT (v & bit_prawo) THEN GO TO 30
25 LET _stan_prawo = 1
30 IF NOT (v & bit_lewo) THEN GO TO 40
35 LET _stan_lewo = 1
40 IF NOT (v & bit_gora) THEN GO TO 50
45 LET _stan_gora = 1
50 IF NOT (v & bit_dol) THEN GO TO 60
55 LET _stan_gol = 1
60 IF NOT (v & bit_fire) THEN GO TO 70
65 LET _stan_fire = 1
70 RETURN
```

W programie użyłem operatora `AND` (&), który jest dostępny w innych językach programowania. Niestety w ZX BASIC on nie występuje. Dla przypomnienia jeżeli operacja `v AND 2^x` w wyniku zwróci wartość niezerową, oznacza to że bit **x** w liczbie **v** jest ustawiony na 1.

```
       ; Sinclair_1 - 6 (lewo), 7 (prawo),
       ; 8 (dół), 9 (góra) and 0 (fire)
 odczytaj_sinclair1
  ld bc,$EFFE     ; port dla rzędu
                  ; klawiszy od 6 do 0
  in a,(c)        ; odczytanie wartości z portu
  ld b,a          ; tymczasowe zapisanie w rej. B

  and 1           ; a AND 1 w wyniku
                  ; da wartość niezerową
                  ; jeżeli bit 0 będzie zapalony
                  ; dla nas wartość rej
                  ; A = 0 oznacza
                  ; że dany kierunek
                  ; jest wciśnięty
  jr nz,_analizuj_sinclair1_1 ; jeżeli NieZero
                              ; to przeskocz
  ld a,c              ; skopiuj wartość
                      ; z rej C. (niezerową)
  ld (_stan_fire),a   ; prześlij
                      ; do komórki pamięci
 _analizuj_sinclair1_1
  ld a,b              ; odzyskaj wartość
                      ; odczytaną z portu
  and 2               ; sprawdź czy bit 1
                      ; jest zapalony
  jr nz,_analizuj_sinclair1_2 ; jeżeli NieZero
                              ; to przeskocz
  ld a,c
  ld (_stan_gora),a
 _analizuj_sinclair1_2
  ld a,b
  and 4               ; sprawdź czy bit 2
                      ; jest zapalony
  jr nz,_analizuj_sinclair1_3
  ld a,c
  ld (_stan_dol),a
 _analizuj_sinclair1_3
  ld a,b
  and 8               ; sprawdź czy bit 3
                      ; jest zapalony
  jr nz,_analizuj_sinclair1_4
  ld a,c
  ld (_stan_prawo),a
 _analizuj_sinclair1_4
  ld a,b
  and 16              ; sprawdź czy bit 4
                      ; jest zapalony
  ret nz              ; wyjdź jeżeli NieZero
  ld a,c
  ld (_stan_lewo),a
  ret
```

Klawiatura jest matrycą 8 półrzędów po 5 kolumn. Półrzędy odpowiadają liniom adresowym od A8 do A15 a odczyt klawiatury następuje co 1/50 sekundy – wtedy skanowane są kolejne porty w ten sposób, że na liniach adresowych (od A8 do A15) pojawia się sekwencyjnie 0. Każdy rząd ma przypisany numer portu – jeśli jakiś klawisz jest wciśnięty na odpowiedniej linii danych klawiatury (KBD0 doKBD4) pojawia się – odwrotnie niż w przypadku aktywności portu Kempston – stan niski. Logika negatywna nie jest niczym niezwykłym a wynika z historii procesorów oraz konstrukcji matrycy klawiatury z prostym adresowaniem. Dlatego też operacja testowania polega na sprawdzeniu, czy operacja `AND 2^bit` wyzeruje rejestr A czy też nie. Jeżeli wyzeruje, oznacza to, że badany klawisz lub kierunek joysticka jest aktywny. Operacje powtarza się dla wszystkich interesujących nas bitów/kierunków.

```
   ; Sinclair_2 - 1 (left), 2 (right),
   ; 3 (down), 4 (up) and 5 (fire)
 odczytaj_sinclair2:
  ld bc,$F7FE     ; załadowanie numeru
                  ; portu do rej. BC
  in b,(c)        ; odczyt wartości
                  ; z portu do rejestru B
    ; pełny adres portu w BC nie będzie już
    ; do niczego potrzebny
  ld a,1          ; do rej. A wartość 1 = aktywny
  bit 0,b         ; testowanie bitu 0 w B
  jr nz,_analizuj_sinclair2_1 ; omiń
                              ; jeżeli NieZero
  ld (_stan_lewo),a
 _analizuj_sinclair2_1
  bit 1,b ; testowanie bitu 1 w B
  jr nz,_analizuj_sinclair2_2      ; omiń
                                   ; jeżeli NieZero
  ld (_stan_prawo),a
 _analizuj_sinclair2_2
  bit 2,b ; testowanie bitu 2 w B
  jr nz,_analizuj_sinclair2_3      ; omiń
```

```
                                ; jeżeli NieZero
 ld (_stan_dol),a
_analizuj_sinclair2_3
 bit 3,b ; testowanie bitu 3 w B
 jr nz,_analizuj_sinclair2_4    ; omiń
                                ; jeżeli NieZero
 ld (_stan_gora),a
_analizuj_sinclair2_4
 bit 4,b
; testowanie
; bitu 4 w B
 ret nz                         ; wyjdź
                                ; jeżeli NieZero
 ld (_stan_fire),a
 ret
```

Kolejna procedura, tym razem do obsługi portów dla Sinclair 2. Tym razem przedstawiam rozwiązanie używające instrukcji BIT testującej poszczególne bity. Cała reszta procedury działa dokładnie tak samo jak dla Sinclair 1.

```
wyswielenie_stanow
 ld hl,_gora              ; do HL adres dla
                          ; wzorca znaku 'góra'
 ld de,ATTR_GORA          ; do DE
                          ; -pozycja na ekranie
 ld a,(_stan_gora)        ; odczytanie
                          ; stanu kierunku
 call _wyswietl_proc      ; wywołanie procedury
                          ; obsługi wyświetlania
 ld hl,_dol
 ld de,ATTR_DOL
 ld a,(_stan_dol)         ; załadownie
                          ; odpowiednich danych
 call _wyswietl_proc      ; i skok do
                          ; procedury obsługi
 ld hl,_lewo
 ld de,ATTR_LEWO
 ld a,(_stan_lewo)
 call _wyswietl_proc
 ld hl,_prawo
 ld de,ATTR_PRAWO
 ld a,(_stan_prawo)
 call _wyswietl_proc
 ld hl,_fire
 ld de,ATTR_FIRE
 ld a,(_stan_fire)
 call _wyswietl_proc
 ret
```

Kolejna procedura pobiera informacje o konkretnych kierunkach oraz wzorcu obrazka dla danego kierunku, a następnie wywołuje uniwersalną procedurę do obsługi przekazanych danych. Dzięki uniwersalności procedury nie musimy powtarzać jej wielokrotnie. Jej działanie jest sterowane poprzez wartości przekazanych parametrów. Odpowiednik w ZX BASIC powtórzony tyle razy, ile jest badanych kierunków/bitów wyglądałby następująco:

```
LET hl=_obrazek_kierunek
LET de=adres_na_ekranie
LET a=_stan_kierunek
GO SUB _wyswietl_proc
```

Poniżej znajduje się procedura, która wykorzystuje przekazane dane. Dobrym nawykiem jest opisywanie procedur oraz parametrów wejściowych i wyjściowych. Można to zrobić chociażby w poniżej zaproponowany sposób.

```
; Procedura wyświetlająca informacje o kierunku
; na podstawie parametrów wejściowych:
; HL - adres obrazka
; DE - adres na ekranie
; A - stan danego kierunku
_wyswietl_proc
 or a             ; sprawdzenie czy w rej. A
                  ; jest wartość 0
                  ; or A ustawia flagę Z
                  ; jeżeli wartość A = 0
 jp nz,wyswietl_kierunek     ; jeżeli NieZero
                             ; do rysuj zgodnie
 ; z przekazanymi parametrami
 ld hl,_puste     ; kierunek jest nieaktywny,
                  ; więc użyjmy
; pustego obrazka
; HL - wzór do wyświetlenia
; DE - adres pamięci ekranu
wyswietl_kierunek
 ld a,7           ; obrazek ma 7 linii
                  ; wysokości

_wyswietl_kierunek
 ld bc,7          ; i 7 bajtów szerokości
 ldir             ; przesyłamy 7 bajtów
                  ; z adresu HL do DE
                  ; czyli przenosimy
                  ; na ekran 1 linię obrazka
 ld bc,32-7       ; musimy wyliczyć
                  ; adres kolejnej linii
 ex de,hl         ; najłatwiej
                  ; poprzez dodanie
                  ; brakującej liczby
 add hl,bc        ; bajtów
 ex de,hl         ; konstrukcję dodawania
                  ; liczb 16bit poznaliśmy
                  ; w poprzednim odcinku kursu
 dec a            ; zmniejszamy licznik linii do wyświetlenia
 jr nz,_wyswietl_kierunek     ; i powtarzamy
                              ; operację
                              ; jeżeli nie
                              ; osiągnęliśmy zera
 ret
```

Procedura do wyświetlania aktualnych stanów joysticka na podstawie przekazanych parametrów. Przekazany w rejestrze HL adres wskazuje na obrazek prezentujący wybrany kierunek, zaś w DE – adres na ekranie, gdzie ma być umieszczony. W akumulatorze przechowywany jest stan. Wiemy, że jeżeli otrzymamy wartość 0, to dany kierunek (nie musimy wiedzieć jaki) jest nieaktywny. Sprawdzamy zatem wartość A i jeżeli jest zerem, zmieniamy HL na adres pustego obrazka. Następnym krokiem jest samo wyświetlenie obrazka pod konkretnym adresem. Na wstępie założyliśmy, że obrazek będzie mieć wielkość 7 „pixeli" w pionie i poziomie. Posługujemy się pamięcią atrybutów w ZX Spectrum. Jej rozmiar to 24 linie po 32 znaki zapisywane jako ciągły blok danych. LDIR kopiuje zadaną w BC liczbę bajtów. Aktualna pozycja w HL wskazuje na nową linię obrazka, adres ekranowy musimy jednak obliczyć. Wiedząc, że cały wiersz to 32 znaki, zaś szerokość obrazka to 7 bajtów, łatwo obliczyć, że powinniśmy dodać do aktualnego adresu 25 bajtów (32-7). Wykorzystujemy do tego poznaną w pierwszej części kursu konstrukcję dodawania do DE wartości z BC. Operację powtarzamy tyle razy, ile jest linii obrazka, zaś aktualny jej licznik jest przechowywany w akumulatorze. Jeśli w HL mieliśmy adres pustego obrazka to w efekcie poprzednia zawartość ekranu zostanie skasowana.

```
   ; Wyświetlenie napisu z adresem w DE.
   ; Liczba 255 kończy napis
   ; DE - napis
wyswietl_napis
   ld a,(de)        ; odczytanie litery
   cp 255           ; czy to 255
   ret z            ; tak - koniec napisu, wyjście
   call PRINT_CHAR ; wywołanie systemowej
                    ; procedury, która
                    ; sama odkłada a na końcu przywraca
                    ; używane przez nią
                    ; rejestry na stosie
   inc de           ; zwiększenie licznika pamięci
                    ; = przejście do kolejnej litery
   jr wyswietl_napis    ; powtórzenie operacji
                        ; dla kolejnego znaku
```

Podprogram do wyświetlania napisów. Ich adres przekazywany jest w DE. Wykorzystujemy do tego systemową procedurę PRINT_CHAR (o adresie $10). Zwykle, aby nie stracić danych w rejestrach, odkłada się je na stosie, a po powrocie z podprogramu przywraca, lecz wspomniana procedura robi to za nas i tym razem nie musimy się o to martwić. Procedura jest, tak jak poprzednie, przykładem pętli z kontrolowanym wyjściem warunkowym.

To są wszystkie niezbędne do działania procedury. Aż tyle albo tylko tyle. Potrzebujemy jeszcze danych oraz zmiennych, niezbędnych do działania programu. Ich deklaracja wygląda następująco:

```
_napis_typ
 db 22, 20, 0 ; AT y,x
 defm „Typ: „,255    ; pasmo pozwala
                     ; również na używanie db
_napis_pomoc
 db 22, 21, 0 ; AT Y X
 defm „Z - zmiana typu joya“,255
```

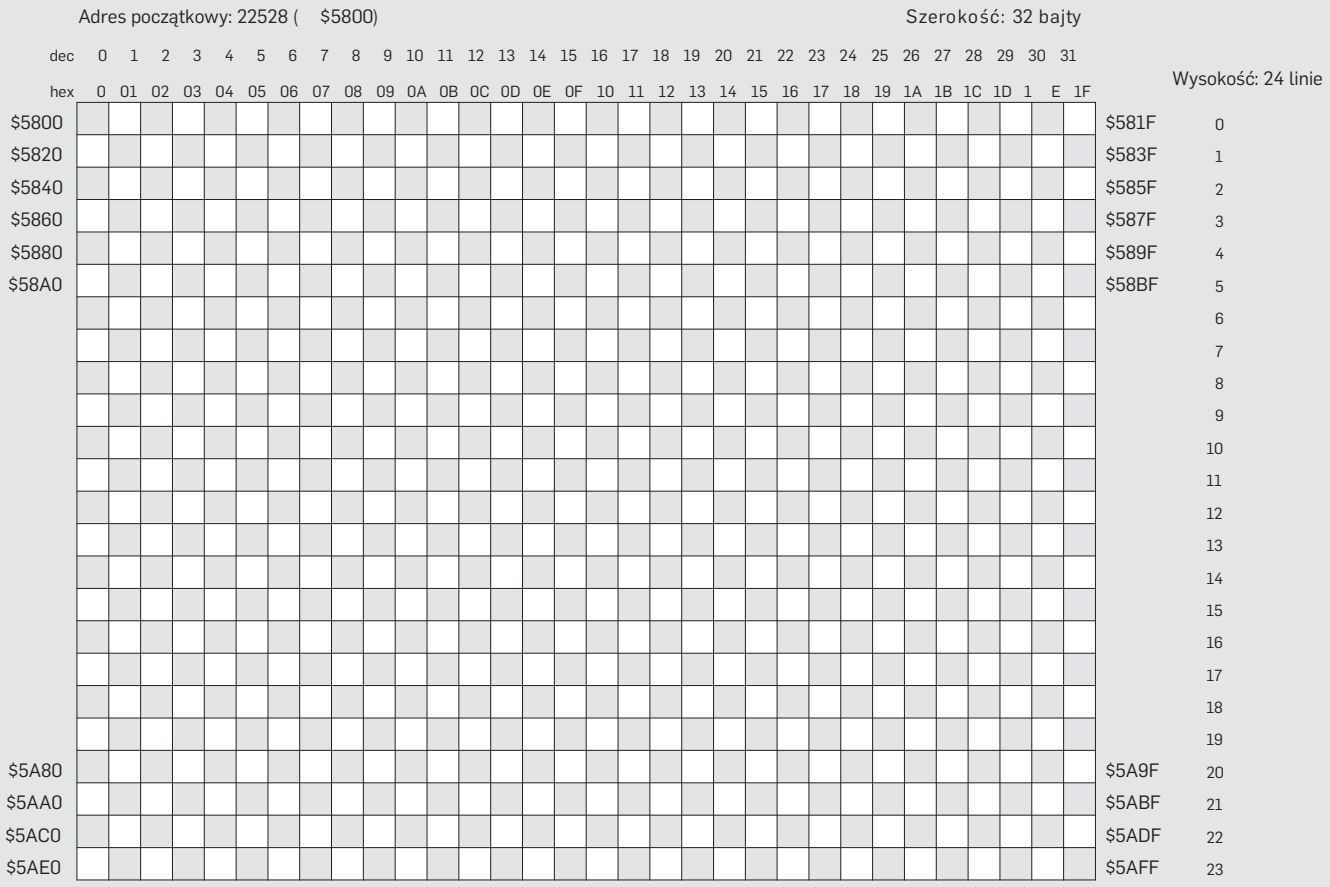
Pod tymi stałymi kryją się napisy oraz ich pozycje na ekranie. Wykorzystywany jest znany z BASICa AT (y,x) który umieszcza napis w linii Y na pozycji X. Kod znaku sterującego dla AT to 22, pozostałe zaś dwie liczby to pozycje Y i X na ekranie. Definicja ciągu znaków zakończona jest liczbą 255 – to nasz znacznik końca napisu, wykorzystywany w procedurze `wyswietl_napis`.

```
_napis_kempston
 defm „Kempston „,255
_napis_sinclair1
 defm „Sinclair 1“,255
_napis_sinclair2
 defm „Sinclair 2“,255

_napisy
 dw _napis_kempston
 dw _napis_sinclair1
 dw _napis_sinclair2

  ; 0 - kempston ; 1 - sinclair 1 ; 2 - sinclair 2
_typ_joya: db 0

  ; 0 - nieaktywne, 1 - aktywe
_stan_gora: db 0
_stan_dol: db 0
_stan_lewo: db 0
_stan_prawo: db 0
_stan_fire: db 0
```

`_napisy` to tablica adresów napisów dla typów joysticków. Każdy adres to 16 bitowe słowo, zaś dw to pseudo-instrukcja asemblera zapisująca w pamięci 16 bitową daną. Dla pierwszych trzech wartości to będą adresy napisów zdefiniowanych kilka linijek wyżej.

Na końcu znajdują się zmienne przechowujące informacje o typie joysticka oraz stanach dla poszczególnych kierunków.

Organizacja pamięci atrybutów w ZX Spectrum

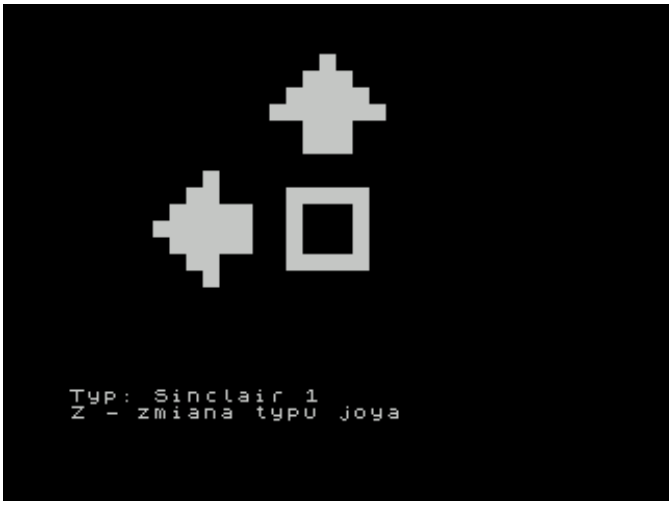


```
_0: equ 7
_1: equ 56

_lewo:
 db _0,_0,_0,_1,_0,_0,_0
 db _0,_0,_1,_1,_0,_0,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _1,_1,_1,_1,_1,_1,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_0,_1,_1,_0,_0,_0
 db _0,_0,_0,_1,_0,_0,_0
_prawo:
 db _0,_0,_0,_1,_0,_0,_0
 db _0,_0,_0,_1,_1,_0,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_1,_1,_1,_1,_1,_1
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_0,_0,_1,_1,_0,_0
 db _0,_0,_0,_1,_0,_0,_0
_gora:
 db _0,_0,_0,_1,_0,_0,_0
 db _0,_0,_1,_1,_1,_0,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _1,_1,_1,_1,_1,_1,_1
 db _0,_0,_1,_1,_1,_0,_0
 db _0,_0,_1,_1,_1,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
_dol:
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_1,_1,_1,_0,_0
 db _0,_0,_1,_1,_1,_0,_0
 db _1,_1,_1,_1,_1,_1,_1
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_0,_1,_1,_1,_0,_0
 db _0,_0,_0,_1,_0,_0,_0
_fire:
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_1,_0,_0,_0,_1,_0
 db _0,_1,_0,_0,_0,_1,_0
 db _0,_1,_0,_0,_0,_1,_0
 db _0,_1,_1,_1,_1,_1,_0
 db _0,_0,_0,_0,_0,_0,_0

_puste:
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
 db _0,_0,_0,_0,_0,_0,_0
```

Pozostaje nam jedynie zdefiniować obrazki ilustrujące każdy z kierunków. Z racji tego, że to są atrybuty i każdy bajt odzwierciedla kolor, a nie pixel w rozumieniu grafiki, użyłem pewnego sposobu. Jako stałą _0 zdefiniowałem kolor tła, zaś _1 – jako kolor pixela. Dzięki temu łatwo było mi już definiować „grafikę". Dla _1 jest to liczba 56 – czarne pixele, białe tło. Jeżeli ktoś chciałby zmienić kolory na inne - wystarczy zmodyfikować tę stałą, a kolor zmieni się prawie automatycznie (po ponownym skompilowaniu)

```
end start
```

Na samym końcu „end", który informuje kompilator pasmo (opisany w pierwszej części kursu), od jakiego adresu program powinien wystartować. Dzięki temu będzie mógł wygenerować poprawnie BASICowy loader.



Kompilacja programu:

```
pasmo --tapbas joytester.asm joytester.tap
```

Od teraz można się cieszyć nowym programem narzędziowym! ■

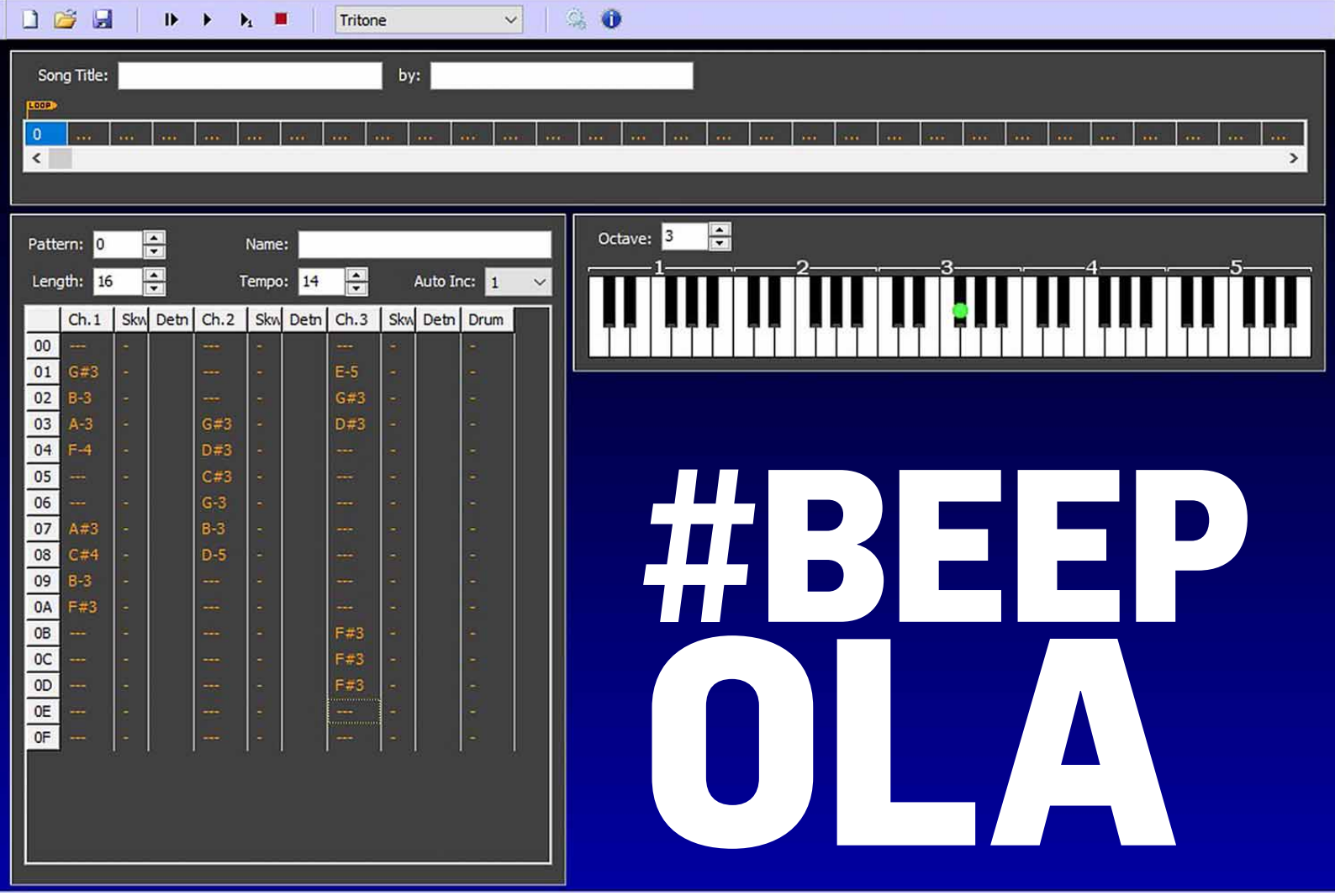
PRACA DOMOWA

Program został napisany tak, aby był zrozumiały. Można go jednak zoptymalizować, i to pod względem wydajności (szybsze działanie), jak i oszczędność miejsca. Proponuję przeanalizować program na następnie spróbować go ulepszyć, a efekty prac przysłać do mnie na adres tygrys@speccy.pl. Podpowiedzi jak to zrobić zostały przemycone w pierwszej części kursu. Jako dodatkową pomoc w linkach zamieszczam odnośnik do strony, gdzie przedstawiono sposoby na optymalizację programów w asemblerze Z80. Jednocześnie chciałbym przypomnieć, że optymalizujemy programy na samym końcu. Przede wszystkim muszą działać tak jak chcemy. Źle przeprowadzona optymalizacja może sprawić, że program przestanie działać zgodnie z założeniami.

OPERACJE NA BITACH	NR = 0, 1, 2, 3, 4, 5, 6, 7 REJ = A, B, C, D, E, H, L, (HL) BIT NR,REJ - TESTOWANIE BITU SET NR, REJ - USTAWIANIE BITU RES NR, REJ - ZEROWANIE BITU
OPERACJE BLOKOWE NA PAMIĘCI	HL = ŹRÓDŁO, DE = PRZEZNACZENIE, BC = ILOŚĆ LDI - (DE) = (HL), INC HL, INC DE, DEC BC LDIR - LDI DOPÓKI BC <> 0 LDD - (DE) = (HL); DEC HL, DEC DE, DEC BC LDDR - LDD DOPÓKI BC <> 0
OPERACJE WEJŚCIA / WYJŚCIA	REJ = A, B, C, D, E, H, L IN A,(PORT 8BIT) IN REJ, (C) - BC = PORT 16BIT OUT (PORT 8BIT),A OUT (C), A

LINKI

Pasmo - http://pasmo.speccy.org/
ZX Spin - https://sites.google.com/site/ulaplus/home/zx-spin-and-basin
Fuse - http://fuse-emulator.sourceforge.net/
Specyfikacja ZX Spectrum
https://www.worldofspectrum.org/faq/reference/reference.htm
Optymalizacja programów dla Z80:
https://wikiti.brandonw.net/index.php?title=Z80_Optimization
http://z80-heaven.wikidot.com/optimization
Link do archiwum z programem:
https://www.speccy.pl/images/articles/kurs_asm_cz2.zip



CHICADII

Jakkolwiek by nie porównywać ze sobą mainstreamowego ośmiobitowego sprzętu lat osiemdziesiątych pod względem możliwości dźwiękowych, to nie da się nie dojść do wniosku, że sir Clive Sinclair był fonicznym minimalistą. W porównaniu do wypasionych układów muzycznych konkurencji – SID w C64 czy POKEY w Atari (oddajmy sprawiedliwość – pojawiły się one później) rzężący beeper Spectrusia był piskliwym śmiechem na sali, co zresztą zostało bardzo szybko zauważone i skorygowane pojawieniem się zewnętrznego układu AY-3-8910 (i tak zapóźnionego), który został zaimplementowany wewnątrz konstrukcji późniejszych, 128 kilobajtowych iteracji rodziny ZX Spectrum. Jednak zanim to nastało, ośmiobitowi deweloperzy musieli pracować na tym, co fabryka dała, a sukcesywna praca organiczna doprowadzała nierzadko do spektakularnych efektów. To, co udawało się wycisnąć z zaimplementowanego jednobitowego brzęczka na podstawie dostępnych ówcześnie narzędzi, budzi najwyższy szacunek.

Cóż, o ile w środku lat osiemdziesiątych Spectrum był game-breakerem, o tyle trudniej było być spectrumowcem na początku lat dziewięćdziesiątych. Odnośnie muzycznych ekwilibrystyk gumiaka i spółki, w spojrzeniach i gestach commodo – i ataro – fanów nie było litości, a bycie spectrumowcem w tym czasie można porównać do bycia grubym w podstawówce. Kto to przeżył i przetrwał, ten da sobie radę w życiu. Takoż żyjemy.

Dzisiaj żyjemy w takich czasach, w których tworzenie jest nieporównywalnie łatwiejsze. Dostaliśmy w łapki szereg wygodnych narzędzi, które w zasadzie wytrącają wykręty o barierach technologicznych. W przypadku muzyki na ZX Spectrum zdecydowana większość tego typu oprogramowania koncentruje się na wykorzystywaniu potencjału wspomnianego wcześniej układu AY – wraz z historycznym pojawieniem się nowej koncepcji opisu i tworzenia muzyki, jaka zaimplementowana została w przełomowym *Soundtrackerze*. Ten też kierunek ewolucja ochoczo podchwyciła i pognała do przodu, zostawiając chlipiącego beepera niemal w samotności.

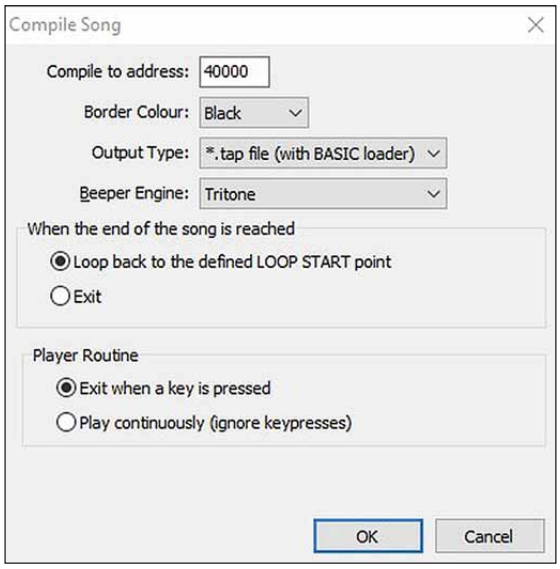
Osobiście nie miałem dużego doświadczenia w obcowaniu z programami do tworzenia muzyki na ZX spectrum – w zasadzie zetknąłem się bliżej z dwoma – bardzo dobrym *Wham! The Music Box* – opierającym się na klasycznym tworzeniu muzyki za pomocą pięciolinii i nut, oraz później – ze wspomnianym *Soundtrackerem*, który systemem typu tracker postawił na głowie sposób tworzenia muzyki na komputerach, przeszedł szturmem przez szesnastobitowce i do momentu pojawienia się oprogramowania klasy DAW niepodzielnie królował (i w swojej dziedzinie nadal jest stosowany i uznawany) na rynku komputerowego tworzenia muzyki.

Beeper jednak nie umarł, garstka fanów jaka się ostała nie pozostała bierna i również zaczęła tworzyć narzędzia wyciskające siódme poty z tej, jak to powiadają „użytecznej maszyny". Takim narzędziem właśnie jest bohater niniejszego tekstu, czyli program *Beepola*. Jest to prosta aplikacja działająca natywnie w środowisku Windows, w której za pomocą myszki możemy wyciskać wspomniane siódme poty ze spectrumowego beepera. Narzędzie bazuje generalnie na koncepcji trackerów i posiada analogiczną filozofię tworzenia muzyki. Podchodząc naukowo do sztuki: pieśń dzielimy na segmenty, w obrębie każdego z nich można zdefiniować maksymalnie 126, następujących po sobie dźwięków, tworzących układ melodyczny. Do dyspozycji, w zależności od wybranego engine'u mamy kilka równoległych kanałów (hmm… przy 1-bitowym urządzeniu, to intrygujące) oraz narzędzie perkusyjne. Każdemu segmentowi nadajemy tempo oraz, w finale, układamy sekwencję odtwarzania wykreowanych segmentów. Do edycji otrzymujemy również narzędzia transpozycji melodii na poziomie segmentów oraz całego utworu, łączenia segmentów, przycinania, wydłużania

itd. Po obróbce play i voilá! Muzyka płynie z chipa. Proste? Proste!

Najciekawsze jednak jest to, co z tego chipa da się wycisnąć i w jakim stylu. Dla istot o mniej nerdowsko-trackerowym stylu tworzenia muzyki zaprojektowano typową muzyczną pięciooktawową klawiaturę, naciśnięcie klawisza której skutkuje pojawieniem się dźwięku w układzie melodycznym w obrębie edytowanego segmentu. Prosta rzecz, a robi wielką różnicę w trackerowej edycji.

Beepola udostępnia 12 engine'ów muzycznych, każdy z nich ma swój własny charakter i własne możliwości. Od wyboru engine'u uzależniony jest właśnie styl brzmienia utworu, engine nakazuje beeperowi sposób odtwarzania zaprogramowanej linii melodycznej. To właśnie jest ten punkt, w którym dźwięk różni się od siebie w taki sposób, jak różnią się BASICowe piski od

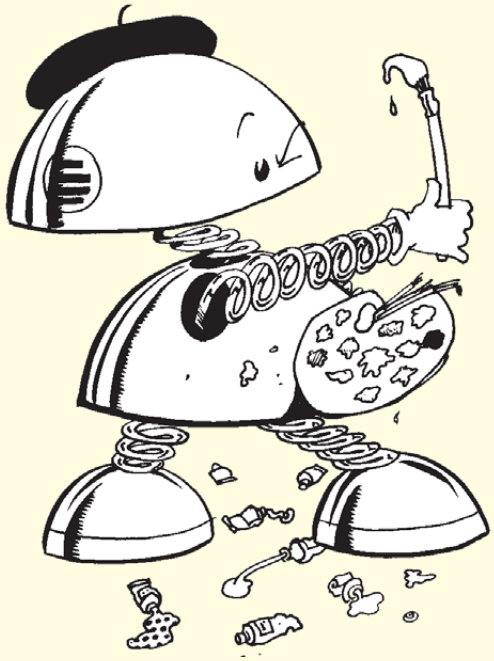


beeperowego soundtracku np. w *Renegade*. Silniki są zróżnicowane i dają wiele opcji pracy – różnią się od siebie liczbą kanałów, typami instrumentów perkusyjnych, czy modulacją poszczególnych dźwięków, skutkujących odmiennym finalnym brzmieniem całości.

Co bardzo przydatne, również z deweloperskiego punktu widzenia, program oferuje bezpośrednią kompilację stworzonego utworu do plików .tap (z loaderem lub bez), .bin lub .asm z możliwością zdefiniowania docelowego adresu całości oraz określenia przerwania do wywołania w tablicy wektorów przerwań. Prosty BASICowy loader zawiera również podstawowy odtwarzacz, umożliwiający wysłuchanie utworu na gołym Spectrumie. Nasze dzieło możemy również wyeksportować do pliku wav.

Programik jest genialny w swojej prostocie. W sumie jedyne czego mi brakuje, to możliwości przypisania danego engine'u do segmentu i stosowania różnych engine'ów w obrębie jednego utworu. Cóż, cieszmy się z tego co jest, *Beepola* skłania do pracy, zabawy, ale i zadumy: „ileż można w prosty sposób wycisnąć ze spectrumowego brzęczyka. Nie ma wymówek dla nietworzenia". ■

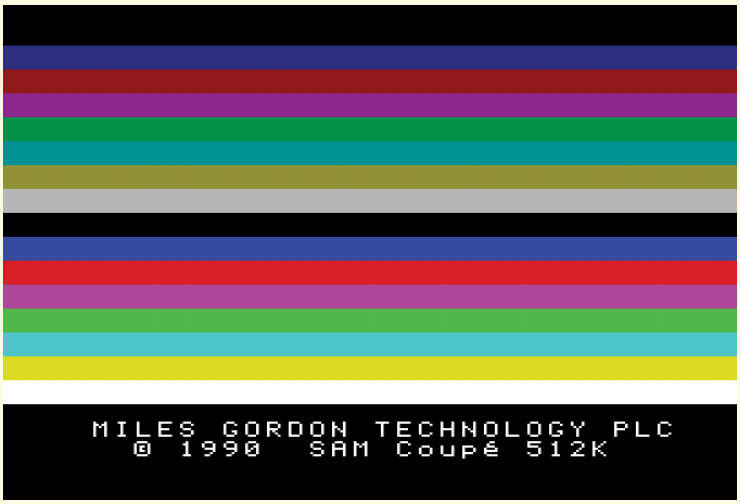
BEEPOLA
WERSJA: 1.08.01
AUTOR: CHRIS COWLEY
PLATFORMA: WINDOWS



SAM BASIC
czyli BASIC doskonały (GRAFIKA)

SIR DAVID **CZĘŚĆ 2**

Każdy użytkownik komputera przeżył kiedyś ten pierwszy raz, gdy usiadł przed klawiaturą i musiał nacisnąć jakiś klawisz. Siadający przed ZX Spectrum z reguły poznawali wtedy komendę **LOAD**, która pozwalała wczytać grę z kasety. Niektórzy zaczynali się później zastanawiać, do czego służą inne wypisane na klawiaturze rozkazy. Ze znajomością języka angielskiego w czasach 8-bitowych było różnie – raczej gorzej, niż lepiej – więc trudno się było nawet domyślać ich znaczenia. W moim przypadku kolejnym etapem były próby wydawania innych poleceń i obserwowanie efektów. Najłatwiejsze do zaobserwowania są oczywiście skutki wykonania rozkazów graficznych. Pamiętam, że narysowanie pierwszych linii albo okręgów było dla mnie dużym przeżyciem.



KOLOROWE PASY NA EKRANIE POWITALNYM SAMA POWSTAJĄ W WYNIKU ZMIANY DEFINICJI KOLORU ZERO (TŁA) CO 11 LINII.

Z czasem dowiedziałem się, że możliwości języka BASIC różnych komputerów pod kątem rysowania grafiki potrafią być bardzo różne. ZX Spectrum nie wypadało w tej konkurencji najgorzej – pozwalało narysować punkty, linie (proste i krzywe) czy okręgi. Do zabawy w statyczne wzorki to wystarczało, ale do napisania czegoś poważniejszego, szczególnie ruchomego, chciałoby się mieć coś więcej. To „coś" oferuje na przykład SAM BASIC.

Komputer SAM Coupé na pewno nie mógł zaimponować dużą ilością dedykowanego dla niego oprogramowania. Mógł za to zadziwić możliwościami wbudowanego BASICa i zachęcić do rozwijania umiejętności programowania w tym języku. Już na dyskietce z systemem SAMDOS, dołączonej standardowo do komputera, możemy znaleźć sporo przykładowych programów demonstracyjnych napisanych właśnie w BASICu. Wszystkie one koncentrują się na możliwościach graficznych, zarówno komputera, jak i języka.

Szczegółowy opis rozkazów graficznych zaczniemy od tych, które działają tak jak w ZX Spectrum, czyli **PLOT**, **DRAW** i **CIRCLE**. Dla przypomnienia:

PLOT *x*, *y* – rysuje na ekranie punkt w miejscu o współrzędnych ***x***, ***y***.

DRAW *x*, *y* – rysuje linię prostą od bieżącego punktu do punktu oddalonego o ***x***, ***y***.

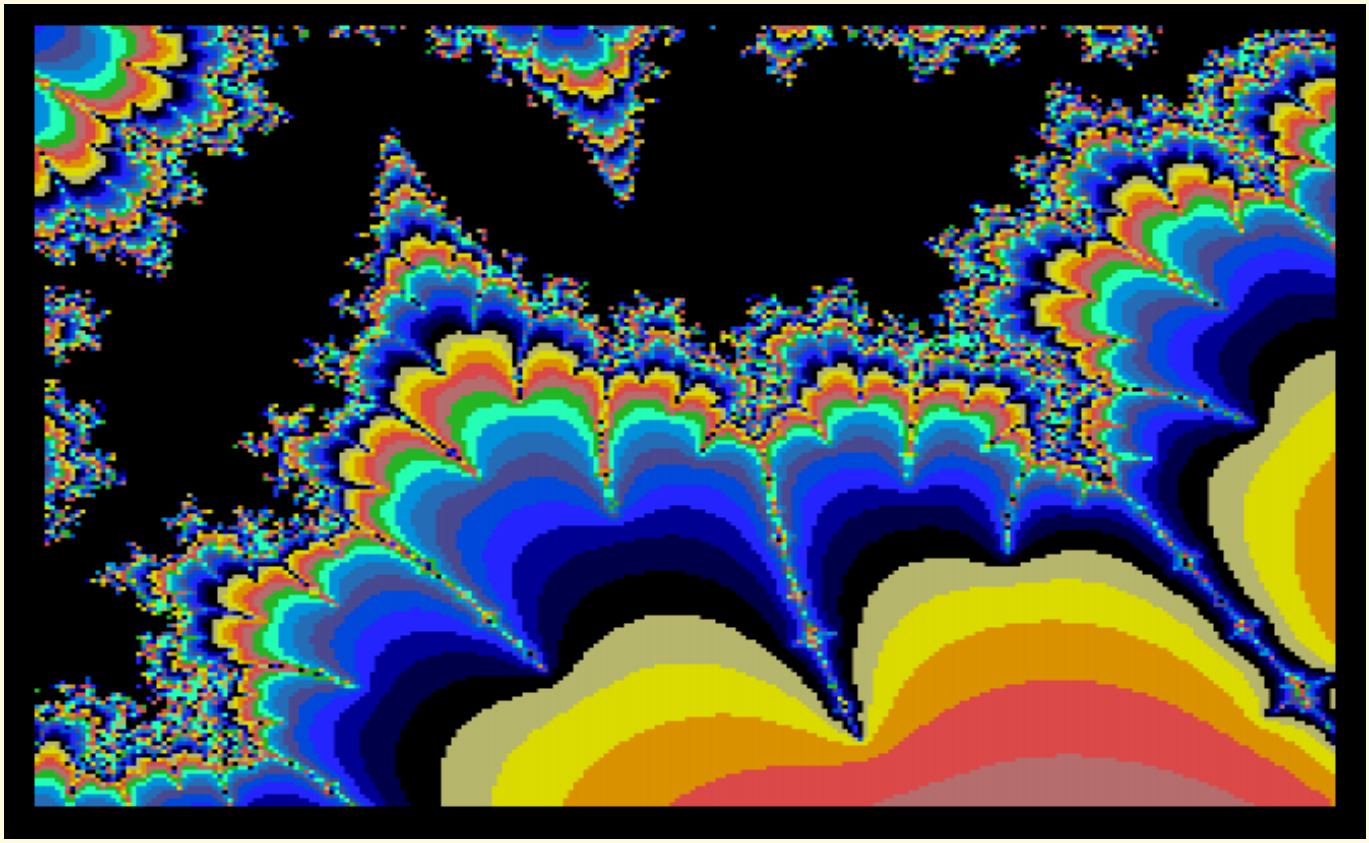
DRAW *x*, *y*, *z* – rysuje linię krzywą o kącie ***z*** od bieżącego punktu do punktu oddalonego o ***x***, ***y***.

CIRCLE *x*, *y*, *r* – rysuje okrąg o promieniu ***r*** ze środkiem w punkcie o współrzędnych ***x***, ***y***.

Warto tutaj wspomnieć, że algorytm rysowania okręgu został w SAMie zmieniony i jest kilkukrotnie szybszy niż jego poprzednik z ZX Spectrum. Ponadto w SAMie rozkaz **DRAW** może zostać użyty w postaci **DRAW TO,** a wtedy podane parametry będą oznaczać bezwzględne współrzędne ekranu końca linii a nie odległość od bieżącego punktu.

Tu pojawia się pierwsza różnica pomiędzy Spectrum a SAMem. SAM może rysować również w obszarze linii poleceń, podczas gdy Spectrum sygnalizuje błąd „B" o wartości poza zakresem. Podobnie w czasie rysowania okręgu – SAM może narysować okrąg wystający poza obszar ekranu (ale jego środek musi się mieścić się na ekranie), podczas gdy Spectrum pokazuje wspomniany błąd „B".

SAM i Spectrum są też do siebie podobne w kwestii sterowania kolorami. Mają te same rozkazy **BORDER**, **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** i **OVER**, chociaż możliwe są drobne różnice w działaniu. A dokładniej, instrukcje ze Spectrum wykonają się na SAMie identycznie (ale tylko w trybie graficznym zgodnym ze Spectrum), jednak w drugą stronę już nie zawsze. I tak na przykład **BORDER, PAPER** i **INK** w Spectrum mogą mieć argument w zakresie 0-7 a w SAMie 0-15. Działanie operacji na kolorach w SAMie zależy jednak od trybu graficznego. Dla trybów z atrybutami, czyli zgodnego z ZX Spectrum (1) i podobnego do TIMEX multikolor (2) wartości **PAPER** lub **INK** większe od 7 włączają automatycznie **BRIGHT 1**, czyli zmienią jednocześnie kolor tła i atramentu na wartości z zakresu 8-15. Jednak w drugą stronę to nie działa – wartości mniejsze od 8 nie włączają **BRIGHT 0** i trzeba tę komendę wydać bezpośrednio. Dla pozostałych trybów, gdzie nie ma ograniczeń wynikających z atrybutów, tło i atrament mogą mieć wartości z pełnego zakresu 0-15. Komenda **BRIGHT** działa w nich nadal, ale tym razem wpisanie rozkazu **BRIGHT 0** przełącza jednocześnie bieżące kolory tła i atramentu na zakres 0-7, a **BRIGHT 1** na wartości z zakresu 8-15. W trybach bez atrybutów nie działa też komenda **FLASH.** Nie oznacza to jednak, że nie da się uzyskać charakterystycznego dla Spectrum migotania, jednak robi się to w inny sposób – odpowiednio definiując kolory z palety.



PROGRAM „PALETTE DEMO" Z DYSKU „SCPDSA" DEMONSTRUJE ANIMACJĘ UZYSKANĄ JEDYNIE ZA POMOCĄ ZMIAN DEFINICJI PALETY

I tak dochodzimy do pierwszej dużej przewagi SAMa nad ZX Spectrum. Otóż SAM ma dostępną paletę 128 kolorów, a ZX Spectrum jedynie 15 (teoretycznie 16, ale kolory 0 i 8 są identyczne). Startowa definicja kolorów SAMa jest oczywiście zgodna z ZX Spectrum. Wartości kolorów do rysowania lub pisania ustawiane rozkazami **PAPER** czy **INK** (w SAMie można zamiennie stosować rozkazy **INK** albo **PEN**) muszą zawierać się w zakresie 0-15, jednak każdy z tych kolorów można zdefiniować oddzielnie z całej dostępnej palety rozkazem:

PALETTE n,k – przypisuje do koloru systemowego o numerze n kolor z całej palety z zakresu 0-127.

Zmiana definicji koloru następuje natychmiast dla całego ekranu. Okazuje się jednak, że SAM BASIC standardowo do koloru systemowego o numerze **n** przypisuje nie jedną, a dwie wartości kolorów z palety. Wydanie komendy jak wyżej powoduje jedynie, że obie te wartości są identyczne i równe **k**. Można jednak wydać taką komendę:

PALETTE *n*, *k1*, *k2* – przypisuje do koloru systemowego o numerze ***n*** dwa kolory z dostępnej palety z zakresu 0-127.

Tym razem oba kolory będą wyświetlane naprzemiennie, będą się zmieniać w tempie identycznym jak przy rozkazie **FLASH**. Nie jest to jednak zamiennik rozkazu **FLASH**, gdyż nie powoduje cyklicznej zamiany kolorów tła i atramentu. Definicja podwójnego koloru przez **PALETTE** zamienia tylko ten jeden kolor, czyli na przykład sam kolor atramentu jakiegoś tekstu, bez zmiany koloru tła. Oczywiście można kolor tła również zdefiniować jako podwójny i ostatecznie uzyskać efekt identyczny jak przy **FLASH**, jednak i tu jest różnica – definicja koloru przez **PALETTE** ma zasięg globalny, czyli działa na całym ekranie. Jeżeli na przykład napiszemy coś mrugającego w kolorze o numerze **2**, to wszystko na ekranie w kolorze **2** będzie mrugać, nie tylko nasz napis. Określenie „na całym ekranie'' też nie do końca musi być prawdą, gdyż można definiować paletę lokalnie, rozkazem:

PALETTE *n*, *k* LINE *y* (albo **PALETTE *n*, *k1*, *k2* LINE *y*) –** przypisuje do koloru systemowego o numerze ***n*** kolor (albo dwa kolory) z palety z zakresu 0-127 dla linii obrazu od wartości ***y*** do samego dołu ekranu albo kolejnej zmiany w którejś z kolejnych linii.

Lokalna zmiana koloru dotyczy również ramki. W taki właśnie sposób utworzony jest ekran powitalny SAM BASICa – poziome pasy w 16 (15 uwzględniając 2 czarne) systemowych kolorach, obejmujących również ramkę. W rzeczywistości to tylko zmiana co 11 linii definicji koloru o numerze zero. Jeżeli w momencie wyświetlania ekranu powitalnego naciśniemy klawisz BREAK z tyłu komputera (albo F11 w emulatorze), to kolorowe tło pozostanie na ekranie na dłużej.

Na koniec omawiania rozkazu **PALETTE** wypadałoby wspomnieć o paru ciekawostkach. I tak wydanie go bez parametrów przywraca wszelkie definicje kolorów do stanu początkowego. Ponadto kilka kolorów systemowych może mieć identyczną definicję palety, co można na przykład użyć do ukrycia prawdziwej zawartości ekranu, z możliwością późniejszego natychmiastowego odkrycia.

Umiejętna manipulacja samą paletą umożliwia uzyskanie ciekawych efektów graficznych, ale to dopiero początek dużych możliwości graficznych SAM BASICa. Najpierw wypadałoby coś na ekranie narysować, a ułatwiają to kolejne ciekawe komendy

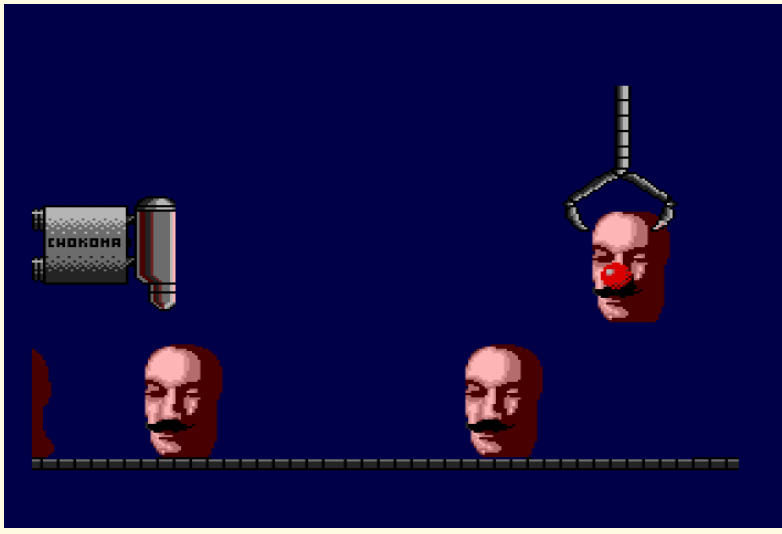
FILL PEN *k*, *x*, *y* – powoduje wypełnienie kolorem ***k*** zamkniętego obszaru ekranu, zaczynając od punktu o współrzędnych ***x*, *y***.

Przez obszar zamknięty rozumiemy sąsiadujące punkty o tym samym kolorze, co punkt startowy. Pominięcie parametru **PEN k** będzie oznaczało wypełnienie bieżącym kolorem atramentu, czyli ustawionym przez **INK** lub **PEN**. Wypełnianie nie jest procesem zbyt szybkim i w skrajnych przypadkach może trwać nawet ponad sekundę. Użycie jednolitego koloru to jednak dopiero początek. Obszar można również wypełnić zdefiniowanym wcześniej wzorem, a robi się to komendą:

FILL USING *a$*, *x*, *y* – wypełnienie obszaru zdefiniowanym w zmiennej ***a$*** wzorem, zaczynając od punktu o współrzędnych ***x*, *y***.

Zanim powiem, skąd wziąć wzór w zmiennej ***a$*** muszę wspomnieć, że SAM BASIC umożliwia umieszczanie w zmiennych tekstowych różnych rzeczy, nie tylko tekstu. Może to być na przykład fragment obrazu, ale nie tylko (o tym później). Wzór do wypełnienia jest właśnie fragmentem obrazu a można go pobrać z ekranu do zmiennej komendą:

GRAB *a$*, *x*, *y*, *szerokość*, *wysokość* – zapamiętanie do zmiennej tekstowej prostokątnego fragmentu obrazu od punktu ***x*, *y*** (lewy górny róg) o rozmiarach podanych w parametrach ***szerokość*** i ***wysokość***.



DEMO „THE CHOCOLATE FACTORY", BĘDĄCE CZĘŚCIĄ WIĘKSZEGO PROGRAMU DEMONSTRACYJNEGO „SCPDSA", JAKO JEDNO Z PIERWSZYCH DEMONSTROWAŁO DUŻE MOŻLIWOŚCI GRAFICZNE SAM BASICA.

Tak utworzony obiekt można później umieszczać na ekranie w różnych miejscach komendą:

PUT *x*, *y*, *a$* – umieszcza na ekranie w miejscu od współrzędnej ***x*, *y*** zapamiętany w zmiennej ***a$*** prostokątny fragment obrazu.

Ze względu na organizację ekranu w pamięci, w obu komendach: **GRAB** i **PUT** współrzędna ***x*** jest zaokrąglana do wartości parzystej. Tak utworzony obiekt graficzny może służyć do wypełnienia komendą **FILL**, jednak aby uzyskać pożądany efekt, musi on mieć rozmiar 16x16 punktów. A jeszcze ciekawszy efekt uzyskamy po wykonaniu komendy:

PUT *x*, *y*, *a$*, *b$* – umieszcza na ekranie w miejscu od współrzędnej ***x*, *y*** zapamiętany w zmiennej ***a$*** prostokątny fragment obrazu, używając przy tym maski zawartej w zmiennej ***b$***.

Maska z drugiej zmiennej również jest fragmentem obrazu o identycznej wielkości jak pierwszy i definiuje sposób nakładania pierwszego obiektu na tło, a dokładniej przeźroczystość tego obiektu. I tak w miejscu, gdzie w masce będzie kolor o numerze 0, pozostanie niezmienione tło. Natomiast gdy w masce będzie w danym punkcie kolor o numerze 15, narysowany zostanie punkt z obiektu ***a$***. Pośrednie wartości w masce, czyli w zakresie 1-14 spowodują nakładanie się na siebie kolorów obiektu i tła.

Wygląda imponująco, prawda? Wystarczy przygotować sobie w programie graficznym obrazek zawierający kolejne klatki animowanej postaci, utworzyć tablicę tekstową dla kolejnych klatek, pobrać je kolejno z obrazka komendą **GRAB** i już mamy w pamięci animowanego bohatera gry, który może się później poruszać na dowolnym tle. To prawie jak programowy „duszek" (ang. „sprite''), który w odróżnieniu od sprzętowych, znanych z innych komputerów, nie jest ograniczony wielkością czy liczbą kolorów. Musimy jedynie sami zadbać o wykrywanie kolizji duszków, czy też zapamiętywanie i przywracanie fragmentu tła, po którym się poruszają.

Kolejne rozkazy służą do przesuwania zdefiniowanego prostokąta w dowolnym kierunku o zadaną liczbę punktów. I tak:

SCROLL *kierunek*, *ilep*, *x*, *y*, *szerokość*, *wysokość* – przesuwa zdefiniowany przez ostatnie 4 parametry prostokąt w kierunku określonym przez pierwszy parametr, o liczbę punktów podaną w drugim parametrze.

Kierunki określone są następująco: 1=lewo, 2=góra, 3=prawo, 4=dół. Natomiast liczba punktów jest zaokrąglana w dół do liczby parzystej z wyjątkiem 1, więc gdy chcemy na przykład przesunąć obraz o 5 punktów, należy wykonać po sobie dwa przesunięcia: o 1 i 4 punkty. Możemy również ograniczyć się do dwóch pierwszych parametrów i wtedy przesunięta zostanie zawartość całego ekranu. Pusta przestrzeń, powstała po przeciwnej stronie niż kieru-

nek, zostanie wypełniona bieżącym kolorem tła, czyli ustawionym komendą **PAPER**.

Bliźniaczo podobny jest drugi rozkaz służący do przesuwania zawartości prostokąta:

ROLL *kierunek*, *ilep*, *x*, *y*, *szerokość*, *wysokość* – działa podobnie jak **SCROLL**, jednak zawartość znikająca przy jednej krawędzi pojawia się z przeciwnej strony.

Jednak najciekawsza i raczej niespotykana w innych implementacjach języka BASIC jest para rozkazów **RECORD – BLITZ**. Wspomniałem wcześniej, że w zmiennych tekstowych można oprócz tekstu przechowywać różne inne rzeczy. Kolejną jest sekwencja instrukcji graficznych zapamiętana przez instrukcję **RECORD**. Jak to działa?

Po wydaniu polecenia **RECORD TO *a$***, w zmiennej ***a$*** będą zapisywane wszystkie wykonania operacji z następującej listy: **DRAW**, **PLOT**, **DRAW TO**, **CIRCLE**, **OVER**, **PEN**, **CLS** i **PAUSE**, aż do wykonania rozkazu **RECORD STOP.** Dla przykładu można zapamiętać narysowanie jakiejś skomplikowanej figury geometrycznej. W jakim celu? Taką sekwencję można później odtworzyć poleceniem **BLITZ *a$***, na przykład w innym miejscu ekranu, w innym kolorze lub innej wielkości. Wykonanie sekwencji będzie jednak znacznie szybsze, niż to pierwsze, „nagrywane", co może bardzo efektownie wyglądać. Zapamiętywana sekwencja przypomina nieco program w BASICu w pamięci komputera, jednak bez zmiennych, pętli itp. Argumentami rozkazów są tylko krótkie, jednobajtowe liczby całkowite. Nie oznacza to, że przy nagrywaniu nie można używać zmiennych. Można jak najbardziej, jednak przy zapisie do sekwencji będą podstawiane aktualne wartości zmiennych.

Jeżeli w sekwencji nie ma zmiennych, to jak narysować zapamiętaną figurę w innym miejscu albo innej wielkości? Tu przydają się kolejne, ciekawe właściwości SAM BASICa, czyli zmienne **XOS**, **YOS**, **XRG** i **YRG**. Są to zmienne, które pojawiają się w pamięci już w czasie uruchomienia komputera i definiują organizację ekranu. Pierwsza para, czyli **XOS** i **YOS** określa położenie początku układu współrzędnych. Standardowo obie mają wartość zero, co oznacza, że początek układu współrzędnych znajduje się tuż nad linią poleceń, w skrajnej lewej kolumnie. Aby zapamiętaną w zmiennej figurę narysować w innym miejscu, wystarczy przesunąć układ współrzędnych. Robi się to oczywiście za pomocą komendy **LET**, np. **LET xos=128**. Wielkość rysowanej figury można natomiast zmienić, manipulując wartościami zmiennych **XRG** i **YRG**, które oznaczają wielkość ekranu, a dokładniej miarę wielkości ekranu. Standardowo **XRG=256**, a **YRG=192**. Wydanie na przykład polecenia **LET xrg=128** sprawi, że linia narysowana komendą **DRAW 127, 0** będzie miała długość równą szerokości całego ekranu, a nie połowy, jak przy standardowych wartościach. Podobnie linia narysowana przez **DRAW 1, 0** będzie miała długość dwóch punktów zamiast jednego. Jak wtedy narysować linię o długości jednego punktu? Można używać ułamków, czyli w tym przypadku będzie to **DRAW .5, 0**. Manipulacja wartościami zmiennych **XRG** i **YRG** pozwala płynnie zmieniać rozmiar rysowanej figury oddzielnie w poziomie i w pionie. Z jednym wyjątkiem, komenda **CIRCLE** zawsze narysuje taki sam okrąg, jak przy standardowej skali, położenie zmienia tylko jego środek.

Na koniec zostawiłem ciekawy sposób na wykorzystanie dużej ilości pamięci RAM SAMa, czyli pracę na wielu ekranach. Już ZX Spectrum 128K miał możliwość wyświetlania obrazu z dwóch obszarów pamięci, jednak nie było to możliwe z poziomu języka BASIC. Natomiast SAM BASIC umie wykorzystać cały dostępny RAM w dowolnym celu, w tym również do utworzenia dodatkowych ekranów. Teoretycznie w pamięci SAMa wyposażonego w 512KB pamięci RAM zmieści się 16 ekranów, jednak zarówno BASIC, jak i DOS muszą mieć swoje miejsce do działania, przez co użytkownikowi zostaje maksymalnie 14 ekranów. Ta liczba może jeszcze maleć zależnie od długości wykonywanego programu. Do obsługi ekranów służą następujące rozkazy:

DEMO „AIRPLANE 3D" Z DYSKU „INFOSOFT DEMODISK" UŻYWA MECHANIZMU RECORD-BLITZ DO WYŚWIETLENIA ANIMOWANEGO, TRÓJWYMIAROWEGO OBIEKTU.

OPEN SCREEN *n*, *tryb* – tworzy nowy ekran o numerze ***n*** w trybie graficznym określonym przez parametr ***tryb*** (zakres 1-4). Utworzony ekran jest na starcie czyszczony i ma ustawioną paletę identyczną jak ekran bieżący.

SCREEN *n* – przełącza się do ekranu o numerze ***n***. Od tego momentu wszystkie operacje graficzne będą wykonywane na tym ekranie, aż do przełączenia się na inny.

CLOSE SCREEN *n* – zamyka ekran ***n*** lub nie robi nic, jeżeli ekran ***n*** nie był otwarty. Zamykany ekran nie może być ekranem bieżącym.

DISPLAY *n* – włącza podgląd na ekran ***n***, podczas gdy wszystkie operacje graficzne będą wykonywane na dotychczasowym ekranie bieżącym. Aby przywrócić widok ekranu bieżącego należy wydać polecenie **DISPLAY 0**.

Numer ekranu musi się zawierać w przedziale 1-16. Ekran o numerze 1 jest otwarty już na starcie i nie można go zamknąć. Nie ma wymogu, by tworzone kolejne ekrany miały kolejne numery. Równie dobrze jako drugi można utworzyć ekran o numerze 16. Każdy z ekranów ma zdefiniowany własny tryb graficzny, bieżące wartości ustawiane przez rozkazy **PAPER**, **INK (PEN)**, **FLASH**, **BRIGHT**, **INVERSE** i **OVER** oraz własną paletę. Chociaż paletę nie do końca, gdyż lokalne definicje palety utworzone rozkazem **PALETTE LINE** są wspólne dla wszystkich ekranów. Wspólny jest również kolor ramki ustawiany rozkazem **BORDER**.

Jakie można znaleźć zastosowanie dla takiej liczby ekranów? Najłatwiej dla dwóch – aby ukryć proces rysowania na ekranie, należy rysować na jednym ekranie, a w tym czasie pokazywać drugi i przełączyć widok dopiero po zakończeniu rysowania. Większą liczbę ekranów można wykorzystać na przykład do tworzenia ciekawych efektów graficznych, aż do całoekranowych animacji. Co prawda taka animacja będzie miała maksymalnie kilkanaście klatek, ale może wyglądać bardzo efektownie.

Gorąco zachęcam do zapoznania się z możliwościami SAM BASICa i podjęcia samodzielnych prób. Własnoręczne napisanie ciekawego efektu graficznego przynosi dużo satysfakcji i motywuje do dalszej nauki programowania! ■

PRZYDATNE LINKI

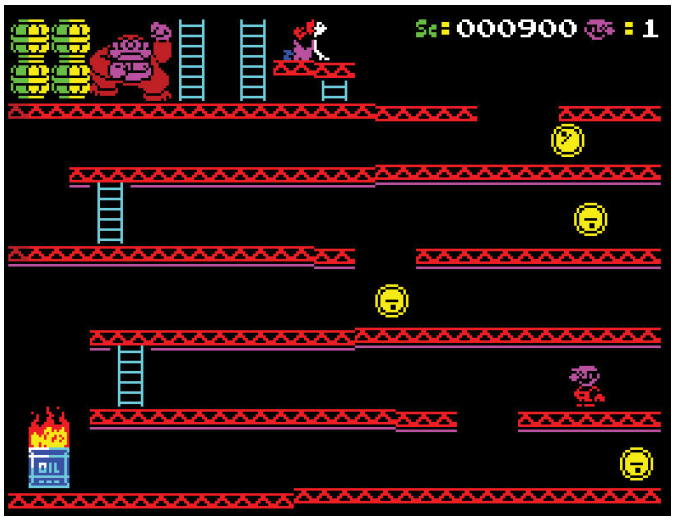
EMULATOR SIMCOUPE
http:// www.simcoupe.org
PODRĘCZNIK SAM BASICA DLA NIECO BARDZIEJ ZAAWANSOWANYCH
http://sam.speccy.cz/basic/sam-basic_complete_guide.pdf

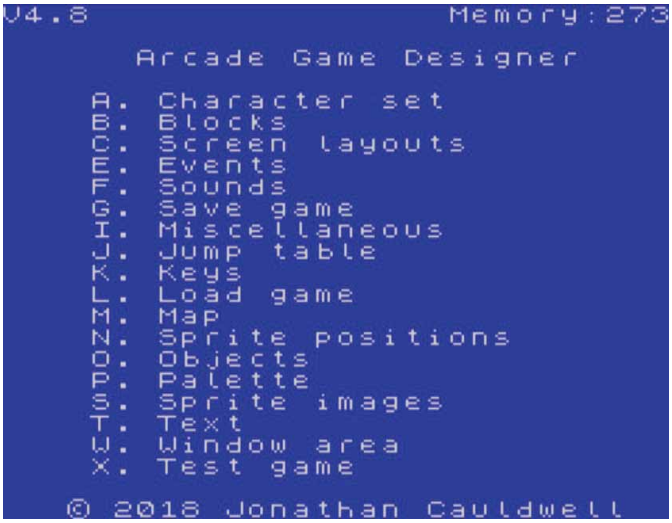
CHICADII

Lata osiemdziesiąte ubiegłego wieku kojarzą się przede wszystkim z szarością. Szaroburość otaczała wszystkich zewsząd, w szarości atrakcją było wszystko, co zachodnie (lub dalekowschodnie) i kolorowe – czy to opakowania z zagranicznych czekolad, etykiety z zagranicznych piw czy lśniące niemieckie katalogi OTTO wydrukowane w full kolor z prezentacjami niedostępnego luksusu. W czasach tych było i specjalne miejsce dla małego ZX Spectrum, którego 8 soczystych wielkoświatowych barw podstawowych było wizualnym splendorem niczym neony na Alejach Jerozolimskich w stolicy, czy w modernistycznych Katowicach. Oczywiście początkowo większość osób z racji posiadania czarno–białych telewizorów mogła cieszyć oczy wyłącznie kolorami obudowy mikrokomputera, część jednak prędzej czy później doświadczyła ich również na kolorowych odbiornikach – czasem specyficzną barwą wybuchających kineskopów Rubinów, a w późniejszych czasach na szpanerskich telewizorach JVC. Ówczesna ubogość kolorystyczna doskonale rezonuje z ubogością fachowej literatury – zarówno w języku polskim, jak i obcym – dotyczącej obsługi i programowania 8–bitowych komputerów. Portfolio literatury książkowej lat 80., dotyczącej ZX Spectrum w języku polskim kończy się raptem na kilku tytułach, poruszających najważniejsze kwestie i pozostawiających masę podstawowych zasad działania języków programowania w gestii domysłów i do własnego zgłębienia. Stąd też w owych czasach pisanie i tworzenie gier czy oprogramowania na ZX Spectrum – zwłaszcza w bardziej zaawansowanych niż BASIC językach – było rzeczywiście wiedzą tajemną i chwała tym wszystkim, którzy w latach osiemdziesiątych wiedzę tę posiedli i wykorzystywali ją dla dobra ogółu.

W pewnym sensie tak jest do dzisiaj. O ile dostępność literatury dzięki internetowi bardzo się zwiększyła i serwery wypełnione są wiedzą, dostępną we wszelkich formach – magazynów, książek, filmów, podcastów, kursów on- i off-line, o tyle paradoksalnie świat swoją dodatkową ofertą dziś tak bardzo rozprasza, że mało kto koncentruje się na zdobywaniu tego, co kiedyś było w zasadzie nieosiągalne, a dzisiaj z kolei jest w przeważającej liczbie wypadków hobbystyczną ciekawostką. Do krzewienia edukacji włączyli się również autorzy Zin80, proponując od drugiego numeru kursy programowania – redaktor naczelny Tygrys w przystępnie przedstawionym kursie Asemblera od podstaw oraz Sachy w praktycznym wykorzystaniu tego języka na potrzeby tworzenia produkcji scenowych. Dzięki swojej czytelnej architekturze, systemy oparte na procesorze Z80 dają możliwość w miarę łatwego wdrożenia się w podstawy programistyczne języków niższego poziomu, a tym samym dają możliwość zrozumienia w jaki sposób działa komputer, co do dziś jest wiedzą przydatną, znajdującą zastosowanie nawet na współczesnych platformach.

Cóż, zdaję sobie sprawę, że mimo wszystko istnieją na tym świecie osoby, którym głowę wykręca na boki na widok wszelkiego, choćby i krótkiego, BASICowego kodu, a gdyby mogli coś zaprogramować, to pewnie byłoby to zapętlenie wyświetlania komunikatu „Good bye, code!". I w zasadzie w radykalnym świecie wydawałoby się, że na tym można by skończyć artykuł, a koderzy i programiści poszliby w swoją stronę, pozostali zaś użytkownicy w drugą. Na szczęście jednak świat nie jest aż tak radykalny i okazuje się, że jest w nim miejsce zarówno dla grupy ludzi, którzy potrafią programować, jak i tych, którzy chcieliby coś stworzyć, a szamotają się bezsilnie w rytm własnej żałosnej mantry „nie umiem – a chcę, nie umiem – a chcę". Ci ostatni do tworzenia potrzebują narzędzia zamiast surowca. Narzędziem stworzonym przez tych pierwszych dla tych drugich jest *Arcade Game Designer* (dalej *AGD*). Jest to oprogramowanie służące do tworzenia oprogramowania, a ściślej rzecz ujmując, jak sama





nazwa wskazuje – gier zręcznościowych, czyli swego rodzaju enginem ułatwiający kreację.

Program występuje w dwóch wersjach – przeznaczonej dla ZX Spectrum 128K oraz Amstrada CPC464, jego autorem jest Jonathan Cauldwell. Geneza i rozwój aplikacji rozpoczyna się w 2008 roku, ostatnia wersja – numer 4.8 – ukazała się w lutym 2019 r.

W zamierzeniu *AGD* jest narzędziem przenoszącym ciężar tworzenia gier z organicznego kodowania na element kreacji i operowania na udostępnionych w aplikacji specjalistycznych modułach, z wykorzystaniem wizualnego tworzenia grafiki, czy nawet elementów fizyki w grze. Aplikacja umożliwia również definiowanie i stosowanie dźwięków – zarówno za pomocą AY, jak i z wykorzystaniem systemowego beepera.

Filozofia działania programu polega na narysowaniu i stworzeniu odpowiednich grafik postaci i obiektów, nadawaniu im określonych funkcji, tworzeniu plansz ekranów, ułożeniu ich w odpowiednią strukturę, a następnie nadaniu logiki, zdarzeń i relacji pomiędzy nimi. Jak wspomniano wcześniej, pierwsza część tych puzzli – część graficzna w dużej mierze wykonywana jest wizualnie, druga z kolei to określanie zachowań na bazie predefiniowanych ustawień lub poprzez określanie własnych. Do tego celu służy wbudowany, bardzo prosty i dobrze opisany edytor kodu. Pozwala on z jednej strony uszczegóławiać detale zdarzeń w grze, z drugiej również po zagłębieniu się i silniejszym wykorzystaniu tego aspektu – zezwala na bardzo duże dostosowywanie zachowań, które będą mogły rzutować na całkowicie inny wygląd gry, niż ten jaki standardowo otrzymuje się na wyjściu.

Część dotycząca graficznych elementów gry dzieli się na kilka modułów. **Character set**, w którym możemy zdefiniować wygląd czcionki stosowanej w grze, z edytorem umożliwiającym edycję każdego ze znaków. **Window area** to moduł określający wielkość aktywnego rejonu gry na ekranie. Kolejny to **Blocks**, w którym kreślimy i definiujemy bloki graficzne o wielkości 8x8 pikseli, z których budowana jest później częściowa grafika gier. Program udostępnia 7 typów bloków, z których każdy w inny sposób wchodzi w interakcję z graczem (pusta przestrzeń, platforma, ściana, drabina, ściana znikająca pod pewnymi warunkami, „śmiertelny" blok oraz blok z rolą do własnego zdefiniowania). Każdy z typów może zyskać odmienny wygląd. Moduł **Screen layout** to z kolei budowanie i tworzenie poszczególnych plansz, rysuje się je ze zdefiniowanych wcześniej bloków – tu najbardziej widać analogię do zabawy klockami, z których tworzy się większą całość. Same plansze są z kolei większymi klockami w module **Map layout**, w którym na obszarze siatki o wielkości 10x8 lokacji rozmieszczamy zaprojektowane wcześniej ekrany. Osobne moduły przeznaczone są do tworzenia pozostałych, aktywnych elementów grafiki. Jest to m.in. moduł **Sprite images**, który służy do tworzenia wszelkich sprite'ów występujących w grze – w tym postaci, pocisków, pojazdów, występujących w grze NPC–ów, czy przeznaczonych do zbierania „znajdek", które nie są na dalszym etapie gry wykorzystywane. Maksymalna wielkość sprite'ów to 16x16 pikseli z możliwością animacji. Narzędzie daje możliwość importu zewnętrznych plików graficznych, selekcji interesującego nas obszaru i zdefiniowania go jako klatka sprite'a. Drugi moduł dotyczący aktywnych grafik to **Objects**. Jego głównym zadaniem jest rysowanie nieruchomych, nieanimowanych obiektów o wielkości 16x16 pikseli, które mogą być również przenoszone przez gracza w różne miejsca gry. Grupę graficznych modułów zamyka **Sprite positions**, w którym rozmieszczamy sprajty na ekranach oraz **Jump table**, w którym możemy graficznie zdefiniować parabolę skoku postaci.

Drugą grupą modułów są moduły odpowiadające za definicje logiki oraz relacji pomiędzy grafiką i tym, co dzieje się na ekranie. W części tej istotnymi modułem jest **Events**, w którym określić możemy niezależne typy zachowań (odnoszące się do ruchu sprite'ów i kolizji), jak i konkretnych zdarzeń gry, np. śmierci bohatera czy ukończenia gry. Interesujące są predefiniowane, gotowe do zastosowania zachowania ruchu postaci. Można tu wykorzystać znane charakterystyki. W przypadku postaci grywalnej – do dyspozycji mamy styl zachowań jak np. *Manic Miner* (czyli sterowanie prawo — lewo + skok postaci), postać poruszająca się po drabinach (czyli np. prawo — lewo — góra — dół i skok), czy w stylu *Cybernoid* (czyli opadający pojazd z grawitacją i strzelaniem). Dla przeciwników z kolei zaproponowane zostały ruchy „patrolowe", czyli poruszającego się w relacji lewo — prawo lub góra — dół lub „podskakujące", nie wymagające szerszego opisu. Ponadto zachowania można przypisać pozostałym elementom środowiska – „znajdkom", blokom, które mogą być popychane, przemieszczającym się platformom itd. Każda z definicji zachowań posiada swoje odzwierciedlenie we w pełni edytowalnym kodzie, który możemy dostosowywać wewnątrz *AGD* do własnych potrzeb za pomocą edytora wewnętrznego języka skryptowego, z wykorzystaniem szeregu wbudowanych i opisanych szczegółowo w dokumentacji instrukcji oraz funkcji.

Należy podkreślić, że edytor ten jest mocnym narzędziem, które naprawdę udostępnia masę możliwości, skrypty zdarzeń są bowiem miejscem, w którym odbywa się cała logika gry oraz sterowanie zdarzeniami i właściwie pełna nad nimi kontrola. Komendy, funkcje i instrukcje są bardzo zróżnicowane – dają możliwość wpływu na zmiany kolorystyki, zachowań, tempa rozgrywki, sposobów wyświetlania tekstów, sterowania pociskami, definiowania efektów cząsteczkowych, zliczania punktów czy wywoływania dźwięków. W zasadzie jest to własny język programowania wbudowany w większe narzędzie.

Kolejne moduły odpowiadające za logikę i warstwę tekstową gry są już mniejsze, jak **Text**, gdzie możemy zdefiniować pojawiające się główne wiadomości w grze, np. **Game Over**, czy **Keys** gdzie określamy klawisze sterujące dla naszej produkcji.

Całości dopełniają moduły **Sounds** – służący do definiowania podstawowych efektów dźwiękowych, **Test game** – do testowania działania gry oraz **Miscellaneous**, w którym określane są globalne opcje, mające wpływ na całość lub odblokowujące wykorzystanie niektórych komend w opisanym wyżej edytorze kodu.

Program oczywiście udostępnia zapis gry na taśmie, a także jego odczyt, gdyby komuś zależało na natywnej pracy na ZX Spectrum lub Amstradzie. Oczywiście wygodniejsza będzie praca tym narzędziem na emulatorze i dużo prostsze zapisywanie obrazu pamięci.

W tym miejscu trzeba zwrócić uwagę w którą stronę rozwija się opisywany engine. W ostatnim roku Jonathan Cauldwell zdecydował o zaprzestaniu rozwoju natywnych wersji *Arcade Game Designera*. Równocześnie w ostatnich miesiącach opublikował on nowe narzędzie działające w środowisku Windows – *Multi—Platform Arcade Game Designer*. Był to długo oczekiwany moment, bowiem tworzenie oprogramowania w nowych systemach jest dużo wygodniejsze. Zasada działania *MPAGD* jest identyczna, jak *AGD*, wszystkie moduły zostały rozmieszczone odpowiednio w systemowych oknach i „menusach". Dzięki obsłudze myszki rysowanie staje się o wiele wygodniejsze, dodatkowo program daje możliwość eksportowania gier do formatów kompatybilnych z komputerami ZX Spectrum, Amstrad CPC, BBC Model B, Dragon 32/64 oraz Acorn Atom. Autor zapowiada dalszy rozwój tej platformy oraz obsługę nowych systemów.

Przerzucenie ciężaru developerskiego na wersję Windows nie oznacza, że wersja natywna umarła. Pałeczkę w rozwoju oprogramowania przejęła społeczność, która rozwija *AGD* od wersji 4.7, która występuje w dwóch wersjach: *AGDx* – zawierająca wiele dodatkowych usprawnień i precyzyjne zarządzanie wolną pamięcią oraz *AGDxMini*, która oferuje zoptymalizowany kod, obsługę sprite-'ów 8x8 pikseli oraz zwiększa możliwość ich wyświetlania do 40 na raz. Aha, no i jest jeszcze *AGD Studio*, również odpowiednik *AGD* pod Windowsa. Uff… dużo tego, i jak widać platforma nie ma zamiaru zwijać się z rynku, a wyboru narzędzia trzeba dokonać ostrożnie, bo co do zasady efekty każdej z iteracji *AGD* nie dają się otwierać w innych edytorach.

Wracając do wersji podstawowej – oczywiście program nie jest bez wad czy pewnych ciążących na nim charakterystyk. Przede wszystkim gry stworzone na bazie standardowych narzędzi i mechanizmów, jakie udostępnia, mają charakterystyczny „look" i łatwo mogą zostać rozpoznane jako stworzone za pomocą tej właśnie aplikacji. Nie jest to wadą zasadniczą, natomiast większość produkcji dostępnych w sieci opiera się na podobnym mechanizmie: plansze, platformy, drabiny, „przeszkadzajki", „znajdźki" oraz charakterystyczna kwadratowa postać głównego bohatera z trochę zbyt długim, antygrawitacyjnym skokiem. Niemniej jednak zdecydowanie należy podkreślić, że to ile można z programu wycisnąć zależy od fantazji i zagłębienia się w kod wewnętrznego języka programu. Polecam zapoznanie się z niektórymi produkcjami – zarówno porządnie wyglądającymi platformówkami, że przytoczę tu tylko np. *Nixy The Glide Sprite* czy drugą część pt. *Nixy and the Seeds of Doom*, jak również produkcjami pojawiającymi się na grupach społecznościowych – tutaj wskażę dwa nazwiska, Allan Turvey oraz Gabriele Amore.Zwłaszcza ten ostatni na swoich profilach prezentuje zaskakujące tytuły wykonane w oparciu o silnik *AGD.* Są to zarówno nowe pomysły (np. *The Legend of the Frog Prince*), jak i adaptacje znanych tytułów – takich jak np. *Popeye*, *Donkey Kong* (tu: *Donky Kong*) czy *Moon Patrol* (tu pt. *Moon Ranger*) ze scrollowaną na żywo planszą i kolorową grafiką – wyglądające o niebo lepiej niż oficjalne adaptacje. W sieci znaleźć można bardzo różnorodne gatunki wykonane przez różnych autorów, od zręcznościówek do kosmicznych shooterów. To pokazuje ile z tego narzędzia da się wycisnąć.

Arcade Game Designer jest na pewno narzędziem znacznie ułatwiającym tworzenie gier i pozwalającym porwać się na to osobom, które żyją na co dzień nieco dalej od kodu. Nie da się go całkowicie uniknąć, natomiast dzięki licznym tutorialom – również na YouTube – funkcjonującym i aktywnym forom internetowym oraz społeczności na Facebooku, przy odrobinie samozaparcia (ale i zewnętrznego wsparcia), stworzenie własnej gry na ZX Spectrum czy Amstrada staje się jak najbardziej możliwe. Do czego zapraszam – gier przecież nigdy dosyć.

■

A G D CIEKAWE TYTUŁY

ŁADNIE WYKONANE PLATFROMÓWKI

◂ NIXY THE GLIDE SPRITE,
NIXY AND THE SEEDS OF DOOM
- BUBBLESOFTGAMES.COM

WARTO ZOBACZYĆ:

PODEJŚCIA GABRIELA AMORE, M.IN:

◂ MOON RANGER
INTERPRETACJA KLASYCZNEGO ***MOON PATROL*** NA ZX SPECTRUM

◂ DONKEY KONG JR -
INTERPRETACJA KLASYCZNEGO DONKEY KONGO

◂ THE LEGEND OF THE PRINCE FROG
I INNE, GRUPA **GABAM'S ZX SPECTRUM GAMES** NA UACEBOOK-U

WYSIŁKI ALLANA TURVEY'A:

◂ PERILS OF WILLY -
KONWERSJA PRZYGÓD WILLIEGO Z VIC-20,
INVADERS RX -
KONWERSJA *SPACE INVADERS*

BRAWUROWY POLSKI AKCENT:

MYSTERIOUS DIMENSIONS - HOOY-PROGRAM, HOOYPROGRAM.I-DEMO.PL

(R)EWOLUCJA: OD TEKSTU DO BARWNYCH PRZYGÓD

GRY TEKSTOWO-PRZYGODOWE NA ZX SPECTRUM NA PRZESTRZENI OSTATNICH CZTERECH DEKAD.

TOOLOUDTOOWIDE

Zanim świat wymyślił MUDy czy też zachwycił się współcześnie grami MMORPG, przygoda była nierozerwalnie związana z tekstem, bowiem na początku, poza tekstem, w tej interaktywnej zabawie niewiele więcej mogło być, z racji ówczesnych możliwości sprzętowych.

Przygodówki, rozwijające się początkowo jako gry tekstowe, później – już w latach 80. – z towarzyszącą tekstowi grafiką, to kamienie milowe gamedevu. Programiści gier lat 70. nie mieli zbyt dużego wyboru – zwykle gry powstawały na mainframe'ach i minikomputerach z prostego powodu – do drugiej połowy lat 70. nie było komputerów domowych. Pierwsze seryjnie produkowane komputery takie jak Apple II (1977), Commodore PET (1977), czy Atari 800/400 (1979) dopiero nadchodziły, a na pierwszą maszynę domową sygnowaną przez Sinclaira musieliśmy poczekać do 29 stycznia 1980 roku.

Jaka była więc pierwsza gra przygodowo-tekstowa wydana na ZX Spectrum? Osobiście obstawiam, że były to gry z katalogu Sinclaira, ze słynnych kaset Adventure A, B, C, D. Dlaczego akurat te? Artic Computing wydało je najpierw na ZX81 – były to gry stricte tekstowe, niezbyt rozbudowane i nie zawierające ani piksela grafiki. Takie też były początki gamedevu, związanego z gatunkiem gier przygodowo-tekstowych, nie tylko na ZX Spectrum, ale praktycznie na każdej platformie. Pojawienie się wspomnianych gier w katalogu Sinclaira było prostym i szybkim rozwiązaniem – ich porty nie wymagały dużego nakładu pracy, więc mogły powstać bardzo szybko. Katalogi były dołączane do pierwszych modeli ZX Spectrum, więc gry z serii Adventure musiały być już gotowe w momencie premiery ZX Spectrum.

Artic Computing wydawało swoje gry tekstowe zanim trafiły one do katalogu Sinclaira – i może to być jeden z dowodów pierwszeństwa w dziedzinie tekstówek. Historia studia Artic Computing zaczyna się na początku lat 80., firma ta zaczynała od programów pisanych dla ZX80. Natomiast w historii ZX Spectrum zapisała się głównie jako wydawca bardzo prostych gier tekstowych, by finalnie w połowie lat 80. zacząć używać systemu *PAW*, o którym wspomnę później. W pierwszym numerze *Your Sinclair*, w krótkim opisie dwóch gier z początkowego okresu AC, oceniono je jako bardzo proste. Jednocześnie autor recenzji pochwalił użycie asemblera zamiast BASICa.

Historycznie pierwszą grą przygodowo-tekstową przeznaczoną na komputery był Adventure (przemianowany później na Colossal Cave Adventure) – przygodówka napisana w 1975 roku przez Willa Crowthera na mainframe DEC PDP-10. Przeportowana na inne platformy w 1977 przez Dona Woodsa czekała na swój debiut na ZX Spectrum… 5 lat. Ujrzała światło dzienne jako Classic Adventure (1983, Melbourne House).

PRZYGODY OD A DO H

Adventure A: **Planet of Death**
Adventure B: **Inca Curse**
Adventure C: **Ship of Doom**
Adventure D: **Espionage Island**
Adventure E: **The Golden Apple**
Adventure F: **The Eye of Bain (1984)**
Adventure G: **Ground Zero (1984)**
Adventure H: **Robin Hood (1985)**

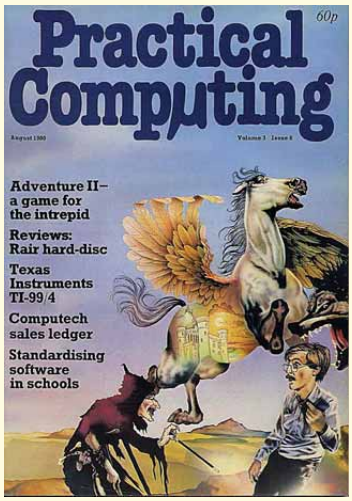
PRZYGODA NA KOMPUTERACH SINCLAIRA

Patrząc na pierwsze gry tekstowe trzeba pamiętać o narzuconych sprzętowo limitach. Do 1982 roku ceny pamięci były na tyle wysokie, że zrezygnowano z planowanego rozszerzenia RAMpack o pojemności 16K, na rzecz mniejszych rozszerzeń o pojemnościach 1KB i 3KB. Brak poważniejszych możliwości graficznych i ogromne ceny pamięci powodowały, że zarówno ZX80 jak i jego następca, ZX81, musiały zadowolić się mniejszymi produkcjami, mniej rozbudowanymi pod względem

interakcji i fabuły. Żeby zobrazować skalę tych cen wystarczy nadmienić, że w 1982 roku RAMpack 16K kosztował 49,95 GBP, a rok wcześniej musiałby kosztować niemal dwa razy tyle. W 1982 roku na rynku pojawił się 64k Memopack do ZX81 firmy Memotech kosztujący prawie 79 GBP. Dla porównania – ZX81 kosztował 49,95 GBP w wersji do samodzielnego montażu i 69,95 funta w wersji gotowej do uruchomienia.

Sytuacja zmieniła się, kiedy w 1982 roku na rynek trafiły dwa modele ZX Spectrum: 16K i 48K, wyposażone odpowiednio w 16 i 48KB pamięci. ZX Spectrum już na starcie oferował programistom znacznie więcej. Gdy dodamy do tego jego możliwości graficzne oraz opcję generowania dźwięku, łatwo pojąć, że świat tekstowych przygód właśnie zyskał nowe wymiary.

OD ADVENTURE II DO SYSTEMÓW TWORZENIA GIER TEKSTOWYCH



Practical Computing, sierpień 1980 (zdjęcie okładki)

Ken Reed: ***Adventure II - an epic game for non-disc systems***

Według wielu ówczesnych twórców gier, rewolucja nadeszła wraz z publikacją artykułu Kena Reeda w sierpniowym numerze *Practical Computing*, w którym przedstawił on metodologię pisania gry przygodowo-tekstowej w pseudojęzyku stworzonym w asemblerze.

Jak grzyby po deszczu zaczęły na platformę, która nas interesuje – najpierw ZX81 a następnie ZX Spectrum – powstawać gry przygodowo-tekstowe oraz dotyczące tej tematyki publikacje książkowe.

Dwa pierwsze systemy napisane w języku Sinclair BASIC zawarto w książkach *The ZX81 Pocket Book* (1981) oraz *Spectrum Adventure* z 1983 roku. O ile w tej pierwszej gra przygodowa i jej kod są elementem jednego z wielu rozdziałów, to druga publikacja w całości poświęcona jest logice i filozofii gier przygodowo-tekstowych.

W 1983 roku również Graeme Yeandle pisze własny system projektowania gier tekstowo-przygodowych zwany *The Quill* – wydany przez Gilsoft. Jest to pierwszy komercyjny edytor oraz system do tworzenia gier na ZX Spectrum operujący bazą danych tekstówki.

Jak sam Graeme wspomina, był on inspirowany artykułem Reeda z *Practical Computing* – przetwarzał dane gry na postać czegoś zbliżonego do bazy danych i umożliwiał kompilowanie finalnej wersji jako samodzielnej gry, niewymagającej uruchamiania *Quilla* i wczytywania danych. Gilsoft kontynuował prace nad systemem gier przygodowych, wydając dwa lata później *The Professional Adventure Writer* (*PAW*), który zawiera również moduł Illustrator, umożliwiający tworzenie oprawy graficznej na bazie wektorów. Obecnie w bazach serwisu World of Spectrum możemy znaleźć ponad 500 gier stworzonych za pomocą systemu *PAW*.

Quill czy *PAW* nie były jedynymi, dostępnymi na rynku systemami do tworzenia przygodówek w czasach świetności ZX Spectrum. Kolejne stworzone wtedy narzędzie to *Graphic Adventure Creator* (*GAC*) – prawdopodobnie największy konkurent *PAWa*. Oba systemy umożliwiały dodawanie do gier tekstowych ilustracji, które tworzone były na bazie sekwencji poleceń rysujących obiekty, co bliższe były grafice wektorowej niż bitmapowej. Co ciekawe, wśród polskich twórców statystycznie większą popularnością cieszył się *GAC*.

Grą, która dokonała rewolucji w grach przygodowych był *The Hobbit*, pierwsza popularna gra tekstowa z grafiką. *Parser*, czyli silnik przetwarzający komendy języka INGLISH (uproszczonej formy języka angielskiego) był efektowny i umożliwił zabawne formułowanie abstrakcyjnych komend w stylu „zabij Thorina zimną rybą", a wrażenie jakie towarzyszyło oglądaniu

Przez bardzo długi czas wydawało się, że ostatnia wersja ***PAW*** *przygotowana na ZX Spectrum 128K +3 zaginęła, a napęd 3" stacji dysków Tima Gilberta był niesprawny. Co więcej, w zasobach 8-bit.info Stefana Vogta był błąd – archiwum wersji programu na ZX Spectrum 128K +3 zawierało tak naprawdę wersję napisaną na CP/M. Na szczęście udało się odnaleźć zaginioną wersję w sierpniu 2017 roku. Miałem wtedy swoje 5 minut, ponieważ to w moich rękach pojawiła się właściwa, ostatnia i najbardziej poszukiwana wersja, którą przesłałem w postaci cyfrowej zarówno do Tima, jak i do Stefana Vogta. Od tamtej pory można ją pobrać ze strony http://8-bit.info/the-gilsoft-adventure-systems/*

Tim Gilberts skomentował to tak: „Zdecydowanie ważne znalezisko. Myślę, że skoro Stefan ma tego kopię, będzie to umieszczone na 8bit.info i będzie dostępne dla wszystkich".

w grze ilustracji danej lokacji potęgowało atrakcyjność gry. *The Hobbit* był sprzedawany bardzo krótko w postaci kasety w pudełku. Prawdziwą furorę zrobiło wydanie gry w zestawie z książką. W tej postaci gra była też dystrybuowana przez Sinclaira. Cóż za połączenie dwóch różnych mediów książki budującej tło gry i komputerowej gry tekstowej, która zabierała nas wprost do książkowego Hobbita.

Logiczną i chronologiczną dla historii napisanej przez Tolkiena kontynuacją *Hobbita* jest trylogia *Władca Pierścieni*. Niestety, pomimo bardzo dobrze zapowiadającego się cyklu nie otrzymaliśmy kompletu obejmującego wszystkie przygody z powieści. Podobnie jak w przypadku *Hobbita*, Melbourne

House wydał wersję zawierającą fizyczne wydanie pierwszego tomu części *Władcy Pierścieni*. Nowością w tej przygodzie była możliwość grania różnymi postaciami i przełączania się pomiędzy nimi. Melbourne House wydał również drugą część sagi pod tytułem *Shadow of Mordor*, jednak już bez takiego rozmachu jak pierwsza część zatytułowana *Lord of the Rings* – w zwykłym opakowaniu i bez książki. Można to tłumaczyć rokiem wydania – 1987 rok to już schyłek świetności i popularności ZX Spectrum na rynku gier.

Cofnijmy się jednak na chwilę do 1984 roku, kiedy to Mike Singleton tworzy kultową grę strategiczno-przygodową zatytułowaną *Lords of Midnight*. Sama w sobie nie jest typową tekstówką – to mieszanina strategii i gry przygodowej, która zdecydowanie ma swoje korzenie w grach tekstowych. Natomiast już sam sposób sterowania i wyboru działań w grze wykracza poza klasyczne tekstówki – nie wpisujemy poleceń, a wybieramy je z dostępnego menu. Warto wspomnieć o *Lords of Midnight* z kilku powodów – oprócz samego pomysłu na grę, Singleton okazał się mistrzem w zakresie optymalizacji kodu, w tym elementów graficznych, z których składa się wyświetlany obraz. *Lords of Midnight* zawiera... ponad 32 tysiące możliwych lokalizacji – kombinacji tego jak mogą być zaprezentowane wszystkie miejsca w grze. Dokonał tego za pomocą techniki nazwanej „landscaping" – metody pokazywania widoku miejsca, w którym się znajdujemy z pozycji pierwszej osoby. Landscaping polegał na przeskalowywaniu osobno różnych elementów krajobrazu, dzięki czemu uzyskiwaliśmy ciekawszy i bardziej zróżnicowany widok. Cała gra została stworzona od zera przez jedną osobę – powstała w zaledwie trzy miesiące. Doczekała się równie udanej kontynuacji *Doomarks' Revenge*.

Parser tekstowy kontra lista wyboru

Punktem startowym dla rozwoju przygodówek były silniki wykorzystujące parsery. Od tego czasu upłynęło kilka lat i gry przygodowo-tekstowe zyskały ilustracje oczywiście wykorzystujące możliwości ówczesnych komputerów. W pewnym momencie ktoś wpadł na pomysł modyfikacji sposobu interakcji gracza z grą. Zamiast wpisywania tekstu łatwiej było pokazać gotową listę wyboru czynności, z której gracz, zamiast mozolnie wpisywać polecenia (i wściekać się na parser, który reagował negatywnymi odpowiedziami) mógł wybrać konkretną akcję i wskazać dostępny przedmiot lub miejsce. Takie zmiany nastąpiły niestety głównie na nowszych, 16-bitowych platformach. Epoka gier tekstowo-przygodowych po 1986 roku na ZX Spectrum w zasadzie się skończyła, nie licząc kilku tytułów powstających w późniejszym okresie.

Jeżeli zastanowimy się nad logiką gier przygodowych to parser jest interfejsem tłumaczącym co gracz chciał zrobić na dostępne w bazie gry, możliwe w danym miejscu interakcje. To, czy klikniemy w ikonkę, czy wybierzemy z menu daną pozycję, czy też wpiszemy rozpoznawalne słowo – czasownik, oznaczający daną akcję, dla dalszej pracy silnika gry jest lub może być tożsame.

KAMIENIE MILOWE I PIONIERZY SPOZA ŚWIATA ZX SPECTRUM.

Klasyki wymieniane jednym tchem:

Colossal Cave Adventure *(1975)*

Hitchhiker's Guide to the Galaxy *(1983)*

Zork *(1977)*

Rogue *(1980)*

Mystery House *(1980)*

Akalabeth: World of Doom (1979) - *pierwsza gra RPG*

Ultima I: The First Age of Darkness *(1980)*

Adventure (1979) – *pierwsza próba przełożenia gry z gatunku interaktywnej fikcji na interfejs graficzny... konsoli Atari 2600*

Sam Mallard – The Case of the Missing Swan

Nie ma chyba lepszego przykładu gry, która jest powrotem do korzeni gatunku połączonej z powiewem świeżości od *Sam Mallard – The Case of the Missing Swan*. To gra z ilustracjami graficznymi, z dużą dozą tekstu oraz interfejsem z wyborem interakcji z dostępnej listy.

Przykład interakcji na podstawie listy dostępnych możliwości pokazuje też np. współczesna polska gra *07 zgłoś się*. Oczywiście przypomina ona w sposobie sterowania gry paragrafowe, bo w danym momencie mamy tylko możliwość wyboru pomiędzy dwiema dostępnymi opcjami, co mocno ogranicza interakcję.

Gry, o których wcześniej nie wiedzieliśmy

Dwa lata temu jeden z użytkowników forum speccy.pl znalazł na zakupionej przez siebie kasecie grę zatytułowaną *Tumitak z podziemnych korytarzy*. Powstała w 1987 roku, jednak większość posiadaczy ZX Spectrum o niej nie słyszała (w zasadzie poza posiadaczem kasety, którym jest KrisZX). Okazuje się, że dwóch Polaków w latach 80. za pomocą systemu *GAC*, stworzyło grę na podstawie opowiadania Charlesa R. Tannera, o tym samym tytule. Co ciekawe, gra była nagrana na składance gier pozbawionych loadera. Szczęśliwie udało się odczytać jej zapis na kasecie (choć na początku samo przewijanie kasety BASF 90 zawierającej kopię *Tumitaka* sprawiało problem) i jest ona dostępna w archiwum speccy.pl. *Tumitak z podziemnych korytarzy* jest typową tekstową przygodówką z ilustracjami wektorowymi, czyli wykorzystuje to, co miał do zaoferowania system *GAC*. Gra powstała na bazie zmodyfikowanej wersji jego silnika, gdzie słowa angielskie zastąpiono polskimi odpowiednikami, co czasem wymuszało ograniczenia w zakresie liczby znaków dla danego polskiego słowa.

Postscriptum

W 2019 roku podczas **Pixel Crunch! 8bit Game Jam**, który odbył się w trakcie festiwalu Pixel Heaven 2019, wraz z Tygrysem i Voyagerem stworzyliśmy grę – czy raczej grywalny mockup gry – z trzema niezależnymi zakończeniami. Grę tekstową, z grafiką oraz... pierwszym w historii ZX Spectrum streamem wideo z karty SD. Po raz kolejny granica tego, co można zrobić w trakcie około 16 godzin wspólnej pracy oraz tego co można wycisnąć z ZX Spectrum, z wykorzystaniem trochę nowszych technologii została przesunięta. Więcej informacji o grze można znaleźć na stronie FB projektu (https://www.facebook.com/speccyraiders/) oraz na portalu speccy.pl.

■

POLSKIE GRY TEKSTOWO-PRZYGODOWE NA ZX SPECTRUM

1986 **KAYLETH**, *Stefan Ufnowski*

MASTERS OF THE UNIVERSE - THE SUPER ADVENTURE, *Stefan Ufnowski*

PUSZKA PANDORY, *M. Borkowski*

REBEL PLANET, *Stefan Ufnowski*

1987 **CONAN - SPOTKANIE W KRYPCIE**, *T. Kopiejć, Z. Wymysłowski, M. Statkiewicz*

HIBERNATUS, *B. Wozniak*

METROPOLIS 1, *T. Kopiejć, Z. Wymysłowski*

MOSCOW-PARIS, *M. Kiszakiewicz, J. Mering*

SMOK WAWELSKI, *P. Kucharski, K. Piwowarczyk*

1989 **MÓZGPROCESOR**, *P. Kucharski, W. Florek, K. Piwowarczyk*

TUMITAK Z PODZIEMNYCH KORYTARZY, *S. Gębka i L. Szamocki*

1993 **SMOK**, *Yerzmyey*

2004 **UCIECZKA ZE SPEJS-SZIPU**, *Hellboj, Yerzmyey*

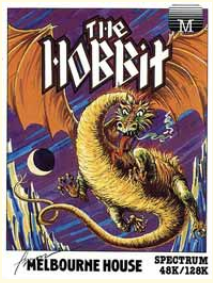
2010 **PAC-TXT**, *Factor 6, Mister Beep, Yerzmyey*

2016 **07 ZGLOS SIĘ**, *Yerzmyey*

2017 **TAJEMNICA CARMEN STRASSE 35**, *Smok Wawelski*

2019 **ASCII**, *Tygrys, Voyager, Tooloud*

10 GIER PRZYGODOWO-TEKSTOWYCH W KTÓRE WARTO ZAGRAĆ NA ZX SPECTRUM



HOBBIT (1983)
Melbourne House

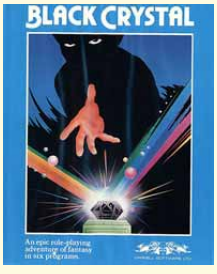
LORD OF THE RINGS
Melbourne House, 1986

PUSZKA PANDORY (1986)

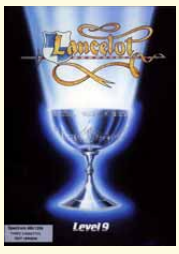
MÓZGPROCESOR (1989)
CAS

TUMITAK Z PODZIEMNYCH KORYTARZY (1989)

THE SECRET DIARY OF ADRIAN MOLE (1985)
Level 9 Computing, Ltd.,



BLACK CRYSTAL (1982)
Carnell Software Ltd

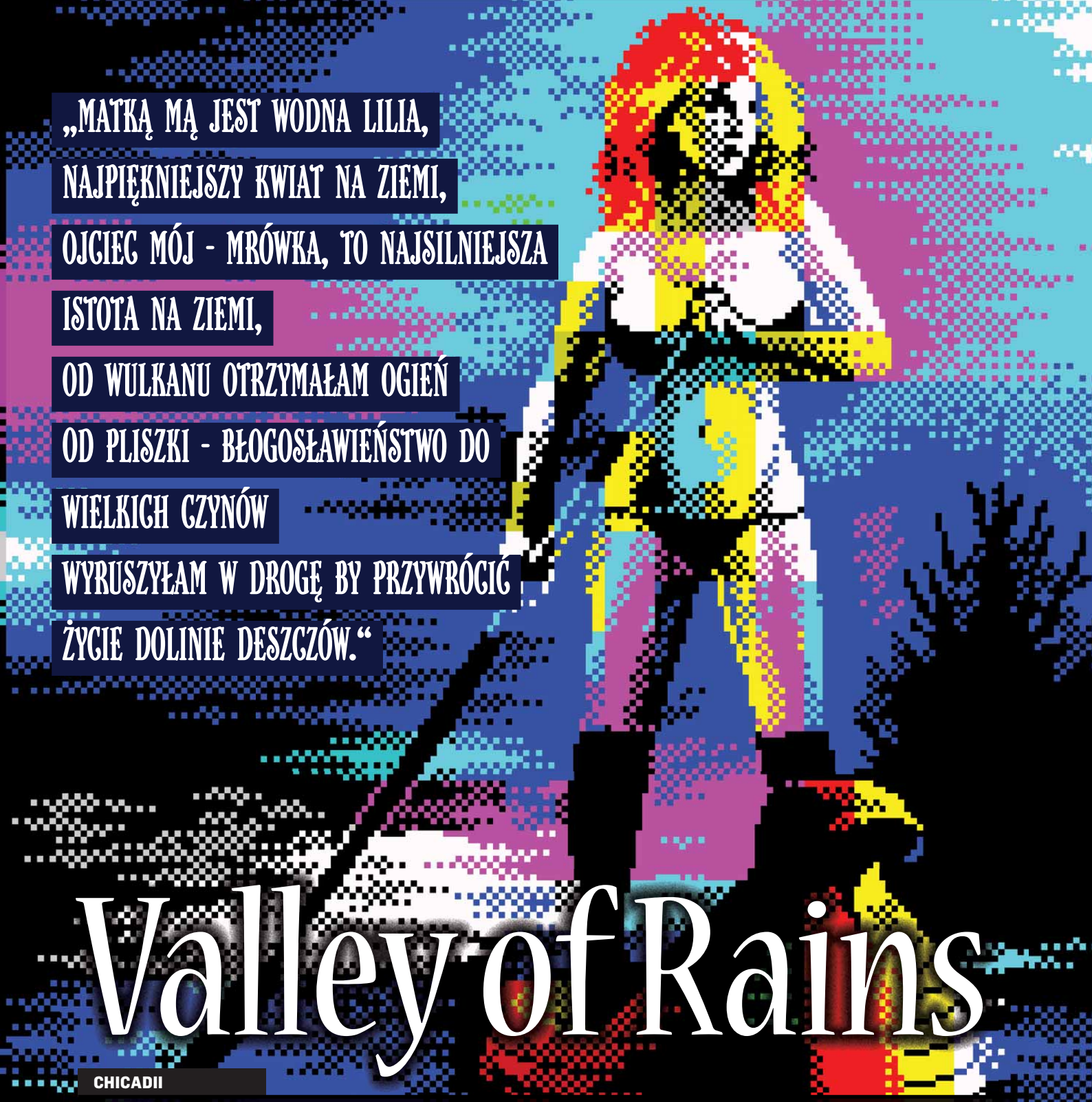


LANCELOT (1988)
Level 9 Computing, Ltd.,

THE SAGA OF ERIK THE VIKING (1984)
Level 9 Computing, Ltd.,

LORDS OF MIDNIGHT (1984)
Beyond

„MATKĄ MĄ JEST WODNA LILIA, NAJPIĘKNIEJSZY KWIAT NA ZIEMI, OJCIEC MÓJ - MRÓWKA, TO NAJSILNIEJSZA ISTOTA NA ZIEMI, OD WULKANU OTRZYMAŁAM OGIEŃ OD PLISZKI - BŁOGOSŁAWIEŃSTWO DO WIELKICH CZYNÓW WYRUSZYŁAM W DROGĘ BY PRZYWRÓCIĆ ŻYCIE DOLINIE DESZCZÓW.“



Valley of Rains

CHICADII

CYTOWANE SŁOWA SĄ IDEOLOGICZNĄ PODBUDOWĄ WYPOWIADANĄ PRZEZ RUDOWŁOSĄ PIĘKNOŚĆ (DO ZAKOCHANIA) I RÓWNOCZEŚNIE GŁÓWNĄ BOHATERKĘ NOWEJ PRODUKCJI ZATYTUŁOWANEJ VALLEY OF RAINS. POWSTAŁA ONA W RAMACH ROSYJSKIEJ INICJATYWY KONKURSOWEJ YANDEX RETRO GAMES BATTLE 2019. ZGŁOSZENIE ANONIMOWEJ GRUPY ZOSYA ENTERTAINMENT PODBIŁO SERCA GRACZY I FANÓW ZX SPECTRUM I ZOSTAŁO ZWYCIĘZCĄ „BITWY” Z 18 INNYMI PRODUKCJAMI (KTÓRYCH POZIOM W WIĘKSZOŚCI PRZYPADKÓW BYŁ PONADPRZECIĘTNY).

Znawcom historii ZX-a styl graficzny gry natychmiast przywodzi na myśl jeden tytuł – *Savage* firmy Firebird. I rzeczywiście, z filozofii tej pozycji Valley of Rains czerpie pełnymi garściami. Czuć że jest to mentalny wzorzec, który „Dolina…” ulepsza, cyzeluje i poprawia. Duże sprajty postaci, generalna rozwałka w korowodzie eksplodujących barw, ciągły pościg w prawo czy w lewo, bez wytchnienia ani chwili spokoju, ciągły bieg i walka – to clou nowego tytułu. Jak wspomniałem, gra duchowo czerpie z *Savage*, natomiast zdecydowanie poprawia niemal wszystkie jego aspekty - jest to wzorcowy przykład implementacji i przerośnięcia swojego mistrza. Nie jest natomiast zróżnicowana pod względem rozrywki (chodzi o element para-FPP w *Savage*), natomiast niemal wszystko poprawia i robi lepiej. Nie odcina się od poprzednika, doskonale to widać w okolicach końcówki pierwszego poziomu, kiedy musimy walczyć z orłami strzegącymi przejścia do dalszego poziomu (forma orła była jedną z grywalnych postaci w Savage).

Dla tych, co nie znają poprzednika – sama gra to charakterystyczny kolorowy run&shoot z dużą postacią główną na ok. 2/5 ekranu, która biegając, skacząc oraz siadając dokonuje generalnej rozwałki napotkanych okoliczności przyrody, spotykając od czasu do czasu groźniejszych przeciwników, którym musi stawić czoła. Generalnie bohaterka uważa, że ma misję, która polega na tym, żeby przywrócić życie Dolinie Deszczów. Winą za zły stan tego miejsca obarcza - jak rozumiem - jej mieszkańców... Rozwałki dokonujemy za pomocą różnego rodzaju broni, które zdobywamy w ramach rozgrywki i związanego z nimi magicznego strzelania. Uzbrojenie wydaje się być kluczowe dla utrzymania skutecznego tempa i efektywnej rozgrywki. Maksymalne pokrycie obszaru przed sobą siłą ognia, pozwala na właściwe tempo/kierunek biegu. Strzelanie wymaga konsekwentnego naciskania klawisza strzału, więc w trakcie gry na klawiaturze kciuk jest w ciągłym ruchu. Znajomy osteopata twierdzi, że to dobrze, no cóż gry zręcznościowe zawsze wyrabiały refleks, tak jest i z tym tytułem.

Valley of Rains wszystko robi pięknie, elegancko i lepiej od Savage, który – głównie poprzez efekty występujące w eksplozjach – sprawiał wrażenie nieco chaotycznego i bałaganiarskiego.

Gra jest szybka i dynamiczna, ale wymagająca zaangażowania. Nie za trudna, balans rozgrywki został dość dobrze dobrany. Wrogowie są zróżnicowani, dosłownie się od nich roi, atakują ze wszystkich stron i na wszystkich wysokościach, co skłania do wykorzystywania pełnego repertuaru ruchów—przysiadów, podskoków, wycofań oraz oczywiście parcia naprzód. Pojawiający się bossowie są nieco trudniejsi, ale nie na poziomie wielkiego wyzwania. Rozgrywka jest satysfakcjonująca, ograniczeniami są czas i energia postaci. Po drodze natrafiamy na power-upy, które pozwalają wydłużyć czas potrzebny na ukończenie etapu, podnoszą wartość energii, a także modyfikują uzbrojenie.

O grze warto pisać przede wszystkim dlatego, że jest graficznie bardzo ładna – jest to jedna z ładniejszych gier na platformę ZX. Zarówno sama grafika, jak i jakość wykonania całości — włącznie z animacjami są małym wizualnym, efektownym majstersztykiem. Od strony jakości rozwiązań programistycznych gra jest również modelowym przykładem do implementacji w innych produkcjach. Bardzo ładny efekt paralaksy, płynne przewijanie ekranów, minimalizacja color clashu, długość rozgrywki skompresowana w niewielkim pliku, super tła – mój ulubiony poziom *Amanita* (czyli muchomor) – *Valley of Rains* wszystko robi pięknie, elegancko i lepiej od *Savage*, który – głównie poprzez efekty występujące w eksplozjach – sprawiał wrażenie nieco chaotycznego i bałaganiarskiego.

Udźwiękowienie na dobrym poziomie, autorzy nie zdecydowali się na wykorzystanie modułu AY, poprzestając na bazowym beeperze, dzięki temu gra uruchomi się w pełnej, przewidzianej krasie nawet na 48KB. Tytułowa melodyjka jest zacna i dobrze brzmi. A więc szacun! Autorzy, notabene, odwołują się do odtwarzania w tle soundtracku, którym jest płyta grupy Tiurula o tym samym tytule. Informują również o możliwości zakupu wersji kasetowej, póki co jednak niestety pod wskazanym adresem widnieje tylko wizytówka.

Jako gra jest to bardzo dobra pozycja, na pewno w latach świetności platformy podbiłaby rynek i przyniosła twórcom miliony, polecam szczególnie fanom zręcznościówek.

■

Valley of Rains
Developer: Zosya Entertainment
Rok wydania: 2019
Platforma: ZX Spectrum 48

Jesień/Zima 2020*

Warszawa

speccy.pl/party

*) W ZALEŻNOŚCI OD SYTUACJI W KRAJU