VGARM, czyli ARM + monitor komputerowy

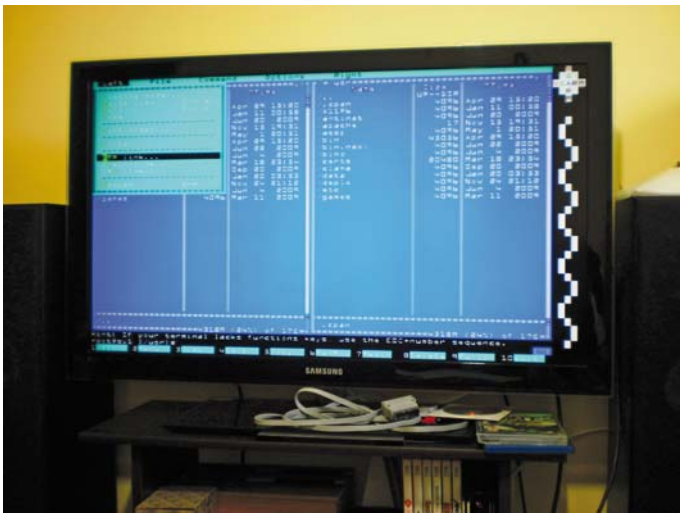
część 1

Niewielki moduł, który w prosty sposób pozwoli dodać do każdego projektu kolorowy wyświetlacz alfanumeryczny w postaci dowolnego monitora VGA lub telewizora! Zobacz, co potrafi nowoczesny mikrokontroler ARM Cortex M3 i garstka innych elementów.

Pomysł na ten projekt zrodził się kilka lat temu. Odkąd pierwszy raz zetknąłem się z „szybkimi" w porównaniu do C51, MSP430 czy AVR-ów mikrokontrolerami ARM7, zastanawiałem się nad możliwością generowania obrazu na monitorze komputerowym przy ich użyciu. Kilku moich kolegów podejmowało w tamtym czasie pewne próby, ale uzyskiwane rezultaty nie były specjalnie porywające.

Tymczasem w EdW 02/2008 pojawił się artykuł autorstwa mojego kolegi **Michała Wysockiego** pod tytułem „AVRTV" opisujący wykorzystanie mikrokontrolera AVR do generowania obrazu na ekranie telewizora (PDF z artykułem jest dostępny w Elportalu wśród materiałów dodatkowych do bieżącego numeru i opisywanego dalej projektu). Poprzednikiem tamtego układu była gra „Snake" tego samego autora (PDF z artykułem także jest dostępny w Elportalu), przy czym AVRTV był wielkim krokiem naprzód, bo co prawda zamiast 3 „kolorów" (czarny, biały i szary) generował tylko 2, jednak z 40x25 dużych kwadratowych pikseli przeszedł do takiej samej liczby pełnoprawnych znaków alfanumerycznych, tym samym uzyskując 64-krotnie większą rozdzielczość (po 8 razy w pionie i poziomie) 320x200 punktów.

Zainspirowany zastosowanym w AVRTV sprytnym i prostym zarazem rozwiązaniem układowym, przystąpiłem do obmyślania jego adaptacji z zastosowaniem procesora ARM7. Niestety z obliczeń wynikało, że pomimo swej znacznie większej mocy obliczeniowej, nie podoła on zadaniu wyświetlenia kolorowego obrazu na ekranie monitora, a przynajmniej nie przy zakładanych przeze mnie parametrach. Pomysł odłożony na półkę doczekał się kolejnej generacji mikrokontrolerów ARM, tym razem z o wiele bardziej dopracowanym rdzeniem Cortex M3. Tu dodam, że podobne urządzenia zazwyczaj konstruowane są w oparciu o układy FPGA,



Fot. 1

jednak jednym z celów przedstawianego projektu było udowodnienie sobie i innym, że podobny efekt można osiągnąć przy pomocy samego mikrokontrolera.

Możliwości układu

Mając dostęp do szybkich „maszynek", udało się opracować układ, którego możliwości dalece przewyższają te z AVRTV – w **tabeli 1** zawarłem zgrubne porównanie tych konstrukcji. Uznałem, że nie ma sensu na nowo opracowywać całej koncepcji takiego modułu i postanowiłem, że VGARM będzie miał te same wymiary co AVRTV i będzie z nim niemal 100% kompatybilny w zakresie wykorzystywanych komend do komunikacji z układem. Przy tym moduł jest w stanie wyświetlić na ekranie monitora, bądź nowoczesnego telewizora z wejściem VGA, obraz składający się w domyślnym trybie z 40 linii po 80

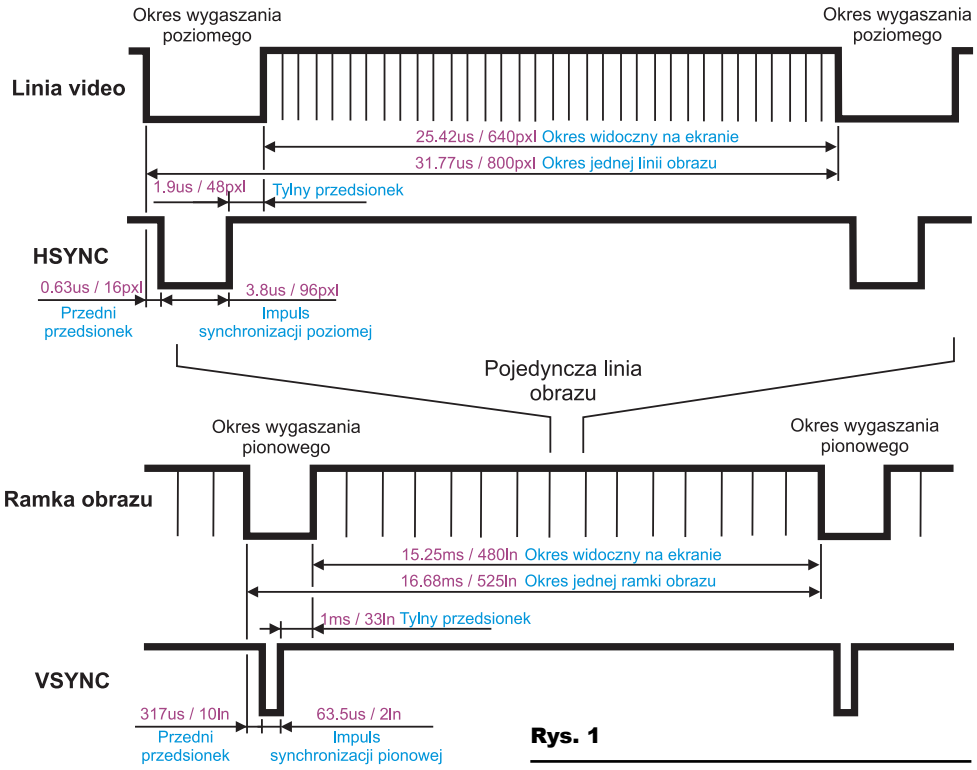
Tabela 1

	AVRTV	VGARM
Rozdzielczość	40 kolumn x 25 wierszy znaków 8x8 pikseli (320x240), obraz czarno-biały	(domyślnie) 80 kolumn x 40 wierszy znaków 8x12 pikseli (640x480), 15 kolorów
Czcionka	Do 256 znaków, w tym podstawowe znaki ASCII, znaki specjalne i polskie znaki diakrytyczne	Do 256 znaków, w tym podstawowe znaki ASCII, znaki specjalne i polskie znaki diakrytyczne
Sterowanie	Interfejs UART (RS232), poziom napięć TTL 5V, prędkość możliwa do ustawienia zworkami	Interfejs UART (RS232), poziom napięć 3.3V (tolerancja 5V), prędkość możliwa do ustawienia zworkami
Wyjście video	Composite 1Vpp/75ohm	VGA RGB, DB15
Wyjście audio	-	mini jack, 300mW, mono, 8bit, ~31kHz
Wymiary	57x40mm	57x40mm
Zasilanie	5V / ~35mA	5V / ~90mA

znaków i to wszystko w 15 kolorach tła i znaku.

Jednym z założeń, jakie przyjąłem, była możliwość wyświetlenia co najmniej 80x25 znaków tak, aby układ mógł generować obraz rodem z tekstowych trybów takich systemów operacyjnych jak Linux czy DOS. Tak, tak, to nie żart, co widać na **fotografii 1**, pokazującej na przykładzie „sztucznie" wygenerowanego obrazu, jak mógłby wyglądać popularny linuksowy program *Midnight Commander* wyświetlony za pomocą naszego modułu VGARM.

Osiągana rozdzielczość obrazu to 640x480 czyli pełne VGA. Szerokość czcionki to klasyczne 8 pikseli, jednak ze względu na sporo miejsca w pionie postanowiłem, że wysokość czcionki będzie większa, a konkretnie 12 pikseli. Dzięki temu czcionka może być bardziej smukła, mieć bardziej naturalne proporcje i zdecydowanie ułatwiona jest kwestia wszelkich „ogonków". Czcionka o tej wysokości pozwala na uzyskanie 40 linii obrazu, a jeżeli to komuś nie wystarcza, to dość łatwo można zmienić wymiary czcionki np. na 8x8 i w ten sposób uzyskać 80x50 znaków (obrazem VGA dużo łatwiej manipulować w pionie niż w poziomie). Dla każdego znaku można niezależnie dobrać kolor tła i samego znaku.



Rys. 1

Opis układu

Na początek kilka słów o użytym mikrokontrolerze. W chwili tworzenia prototypu, najszybszymi dostępnymi w handlu układami z rdzeniem Cortex M3 mogła pochwalić się firma NXP. Na rynku, od hobbystów do średnich firm, głównymi graczami, jeśli chodzi o te układy, są wspomniane NXP oraz ST. W wypadku wielkoseryjnych produkcji sprawy wyglądają zawsze inaczej i w grę wchodzą firmy, o których przeciętny hobbysta mógł nawet nie słyszeć. Firma NXP, pomimo drobnego opóźnienia w stosunku do swojego głównego konkurenta, od razu sięgnęła po drugą rewizję wspomnianego rdzenia. Ten zabieg pozwolił początkowo na osiągnięcie prędkości taktowania do 100MHz! Jako ciekawostkę dodam, że w chwili pisania tego artykułu obie firmy mają w ofercie mikrokontrolery ARM Cortex M3 taktowane jeszcze szybszymi zegarami, np. 150MHz czy 180MHz, a nawet więcej! (NXP niedawno wprowadziło do sprzedaży dwurdzeniowe mikrokontrolery z rdzeniami Cortex: pokrewnym M3 rdzeniem M4 i prostszym M0 w jednej obudowie, oba taktowane zegarem do 204MHz).

Z pracy ze starszymi procesorami ARM7 wyniosłem znacznie lepsze wrażenia w wypadku konstrukcji firmy NXP, niż jej ówczesnego głównego konkurenta, czyli firmy Atmel (nawiasem mówiąc, o ile AVR wyszedł im świetnie, to niestety ARM7 już niekoniecznie, za to np. ARM9 w ich wydaniu jest już całkiem w porządku). Tak więc niewiele myśląc, sięgnąłem po mikrokontroler LPC1768, który miałem pod ręką na uniwersalnej makiecie dydaktycznej Akai Kaba mojego autorstwa (zainteresowanych konstrukcją odsyłam pod adres *www.livelights.pl/akaikaba*). Mikrokontroler ten może według producenta pracować z maksymalną prędkością zegara wynoszącą 100MHz. Na potrzeby projektu został on jednak odrobinę „podkręcony" do około 100,7MHz. Dlaczego? Już wyjaśniam.

Zasada tworzenia obrazu VGA. Tworzenie analogowego obrazu VGA jest podobne, ale jednak odmienne od tworzenia czarno-białego obrazu TV. O ile mnie pamięć nie myli, ten temat był już kiedyś poruszany na łamach EdW, jednak jest to kluczowa sprawa dla projektu VGARM, dlatego należałoby tę wiedzę gruntownie odświeżyć.

Na **rysunku 1** dość wyczerpująco przedstawiony jest timing sygnałów VGA. Cały widoczny obraz składa się z 480 linii po 640 pikseli w każdej z nich. Zarówno dla każdej ramki (klatki) obrazu, jak i dla pojedynczej linii, występuje impuls synchronizacji na odpowiedniej linii – VSYNC dla synchronizacji ramki i HSYNC dla synchronizacji pojedynczej linii obrazu. Obraz na ekranie jest odświeżany z częstotliwością 60Hz. Czas jednej ramki obrazu, na który składa się 480 linii, oraz okres wygaszania pionowego jest oczywiście równy 1/60s, czyli około 16,7ms, co w przeliczeniu na linie obrazu daje wartość 525. Posługiwanie się liczbą linii zamiast czasem bywa wygodniejsze przy pracy z generowaniem obrazu VGA, podobnie jak posługiwanie się pikselami dla opisu czasu w wypadku pojedynczej linii.

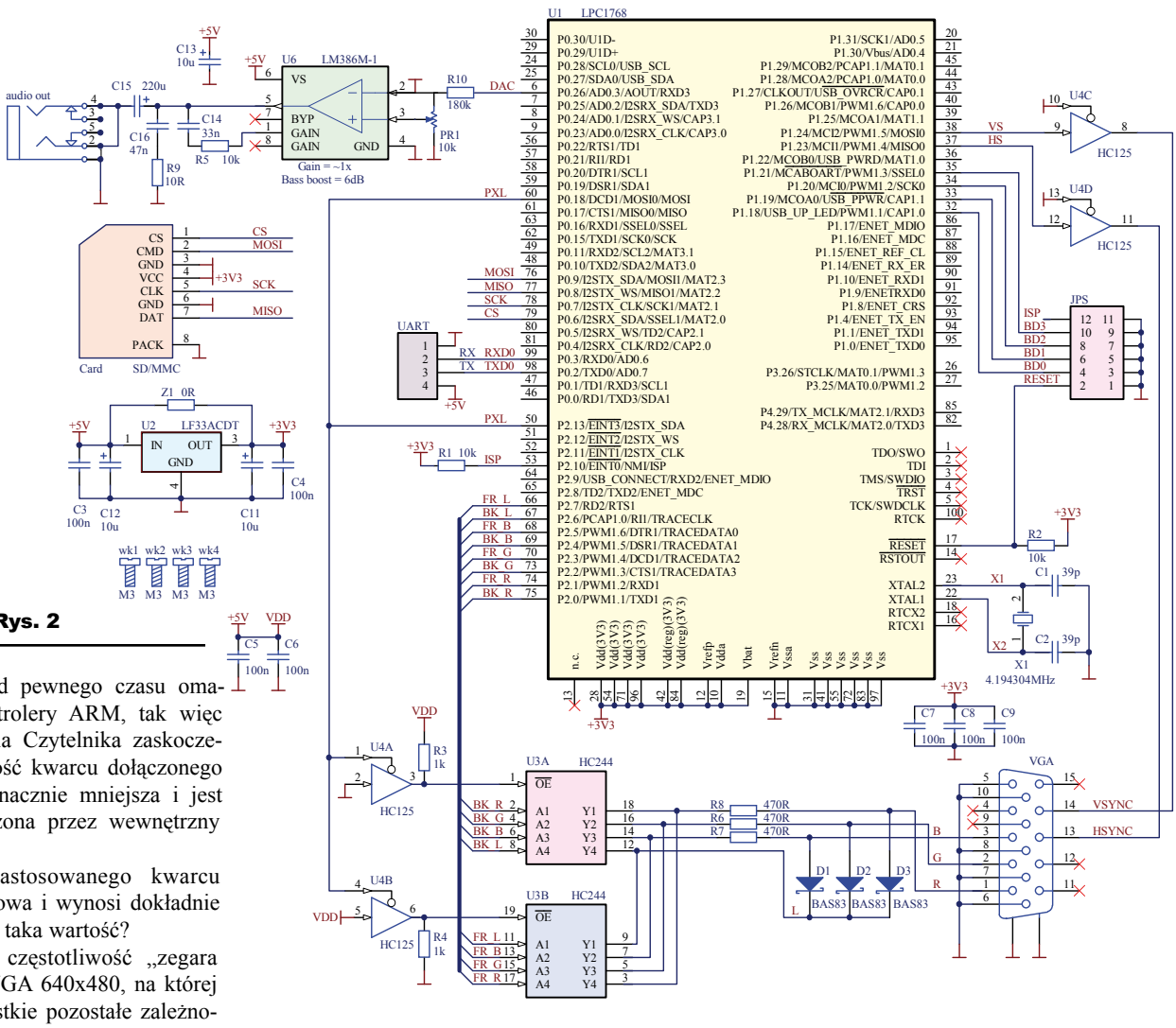
W czasie trwania wygaszania pionowego, po okresie 10 linii, zwanym przednim przedsionkiem, następuje stosunkowo krótki (2 linie) impuls synchronizacji pionowej, czyli na około 64us na linii VSYNC pojawia się stan niski. Potem występuje dalsza część wygaszania, zwana tylnym przedsionkiem, trwająca 33 linie, po czym następuje generacja poszczególnych 480 linii obrazu.

Przy generowaniu linii obrazu występują podobne okresy, jak w wypadku ramki. Także tutaj mamy okres wygaszania, tym razem poziomego, który rozpoczyna się od przedniego przedsionka trwającego w przeliczeniu 16 pikseli. Cała pojedyncza linia obrazu trwa około 31,77us, co odpowiada 800 pikselom. Następnie występuje impuls synchronizacji poziomej trwający 3,8us (96 pikseli) i tylny przedsionek (1,9us / 48 pikseli).

Poza wspomnianymi dwoma liniami synchronizacji, interfejs VGA ma 3 analogowe linie obrazu, po jednej na każdy z podstawowych kolorów, tj. czerwony (R), zielony (G) i niebieski (B). Na 480 widocznych na ekranie pikseli przypada czas 25,42us. W tym okresie na liniach RGB jednocześnie zmieniają się poziomy napięć, tworząc pojedynczy piksel o wybranym kolorze. Można zatem powiedzieć, że piksele są wysyłane na ekran z częstotliwością 25,175MHz. Dość szybko, nieprawdaż?

Ale pamiętajmy, że mowa tu o zupełnie podstawowym obrazie o rozdzielczości 640x480, która niejednokrotnie nie dorównuje nawet dzisiejszym telefonom komórkowym, nie wspominając już o telewizorach czy monitorach.

Sprzęt. Jak widać na schemacie z **rysunku 2**, dominującą rolę w układzie odgrywa opisany już częściowo mikrokontroler LPC1768. Zintegrowana pamięć flash pozwala na zapisanie do 512KB programu i danych, a całkowita pojemność pamięci RAM to 64KB, co ma niemałe znaczenie w omawianym projekcie. Najistotniejsza jest jednak prędkość działania, w tym wypadku wynosząca dokładnie 100,663296MHz. Na łamach EdW od pewnego czasu omawiane są mikrokontrolery ARM, tak więc nie powinno być dla Czytelnika zaskoczeniem, że częstotliwość kwarcu dołączonego do procesora jest znacznie mniejsza i jest ona następnie mnożona przez wewnętrzny blok PLL.



Rys. 2

Częstotliwość zastosowanego kwarcu jest wybitnie nietypowa i wynosi dokładnie 4,194304MHz. Skąd taka wartość?

Jak już wiemy, częstotliwość „zegara pikseli" w grafice VGA 640x480, na której można oprzeć wszystkie pozostałe zależności czasowe, wynosi 25,175MHz. Standard dopuszcza drobne odchyłki od tej częstotliwości, tak więc częstotliwość działania procesora musiała być możliwie bliska wielokrotności „zegara pikseli". Ze względu na zalecane maksymalne taktowanie procesora, wynoszące 100MHz, teoretycznie nie powinniśmy przekraczać trzykrotności „zegara pikseli", czyli około 75,5MHz, jednak jak łatwo zauważyć, czterokrotność wynosząca 100,7MHz powoduje jedynie nieznacznie przetaktowanie procesora i z dużym prawdopodobieństwem można przyjąć, że będzie on działał w pełni poprawnie.

Poszukiwania kwarcu skupiły się zatem na wartościach, które, po pomnożeniu w granicach możliwości bloku PLL, dałyby wynik zbliżony do 100,7MHz. Ostatecznie wybór padł na stosunkowo łatwo dostępny, wspomniany kwarc (stosowany w zegarach, 2^{22} = 4194304). Po pomnożeniu przez PLL otrzymujemy taktowanie procesora 4,194304MHz * 24 = 100,663296MHz. Różnica pomiędzy idealną a otrzymaną częstotliwością to 36,7kHz, co może na pierwszy rzut oka wydawać się dużą wartością, jednak jest to jedynie 0,036% i nie stanowi żadnego problemu dla odbiorników sygnału VGA.

Z powyższych kalkulacji wynika, że na każdy piksel obrazu przypadają 4 (słownie cztery!) cykle procesora. Myślę, że choćby nie wiem jak dopracowany technologicznie był procesor, to 4 cykle najwyczajniej nie wystarczyłyby na wysłanie informacji o obrazie z pamięci wprost na porty przy zachowaniu zakładanych parametrów (choć dodam, że istnieją projekty na mikrokontrolery AVR generujące kolorowy obraz TV, gdzie w przeliczeniu na jeden piksel przypada 6 cykli zegara). I tutaj dochodzimy do tego sprytnego rozwiązania, o którym pisałem we wstępie. W AVRTV jednobitowa informacja o pikselach wysyłana była na ekran niemal wprost z linii MOSI interfejsu SPI (jedynie po odpowiednim kondycjonowaniu napięć). W projekcie VGARM ta koncepcja została nieco zmodyfikowana.

Otóż przy czarno-białym obrazie, generowanym przez sekwencję „zer" i „jedynek", można także mówić o pikselach „zgaszonych" lub o „kolorze tła" (oczywiście w tym wypadku czarnym) oraz pikselach „zaświeconych" lub „kolorze znaku" (w tym wypadku białym). W takim prostym rozwiązaniu wysyła się poprzez interfejs SPI jednobitową informację o kolorze tła i znaku (w zasadzie po „pół bitu" na tło i znak). Interfejs SPI, a konkretnie jego rozbudowana wersja, zwana tu SSP, w procesorze LPC1768 teoretycznie potrafi działać z połową częstotliwości taktowania procesora, a z jedną czwartą tej prędkości na pewno radzi sobie wyśmienicie. Tak więc wiemy już, że przy użyciu samego interfejsu SPI nasz mikrokontroler jest (przynajmniej teoretycznie) w stanie wyświetlić obraz o rozdzielczości 640 x 480 pikseli w kolorze czarnym i białym.

A gdyby tak można było co jakiś czas zmieniać te kolory dla całego obrazu? A może da się to zrobić dla każdego znaku i jego tła? Jak już obliczyliśmy, procesor jest zbyt wolny, aby mógł wysłać informację o kolorze na swój port z częstotliwością 25,175MHz, czyli dla rozdzielczości poziomej 640 pikseli, w czasie 25,4us. Niemniej jednak w tym czasie jest to w stanie zrobić dla mniejszej liczby większych pikseli, przykładowo dla 80.

I teraz wyjaśni się cała tajemnica działania układu. Bufor typu HC244 jest wewnętrznie

podzielony na dwie połowy po 4 bity (U3A i U3B). Do wejść pierwszej części doprowadzona jest 4-bitowa informacja o kolorze tła znaku, a do drugiej podobna informacja o kolorze samego znaku. Na pierwszych 3 bitach kodowana jest barwa poprzez wszystkie 8 możliwych kombinacji, natomiast najstarszy bit określa jasność. Czyli mamy 8 kolorów i każdy z nich może być jasny bądź ciemny, oczywiście poza kolorem czarnym – stąd 15, a nie 16 kolorów.

W pierwszej wersji mojego projektu do wejść kluczujących buforów U4A i U4B trafiał sygnał z wyjścia MOSI. To rozwiązanie miało jednak pewien mankament, podobny jak w AVRTV. Otóż interfejs SPI (SSP) nie potrafi wysyłać danych ciągłym strumieniem – między każdą, w tym wypadku 8-bitową, paczką danych występuje krótka przerwa, która w dodatku na zmianę trwa krócej i dłużej, co na ekranie dawało efekt odstępu między znakami, raz wynoszącego 1, a raz 2 piksele. Pomimo to efekt był zadowalający i wszystko było gotowe, łącznie z artykułem.

Jednak tuż przed wysłaniem kompletu materiałów do Redakcji w rozmowie z kolegą Michałem Wysockim przypadkiem pojawił się pomysł zastąpienia interfejsu SPI przez I2S (to nie błąd w druku – I2S nie I2C!). Jest to stosunkowo nowy interfejs, zapewne mało znany większości Czytelników. Dedykowany on jest przede wszystkim do współpracy z przetwornikami AC i CA w aplikacjach audio. Transmisja odbywa się szeregowo, synchronicznie, bardzo podobnie do tej w SPI (linia danych, linia zegarowa), jednak dane wysyłane są ciągłym strumieniem, a stan na dodatkowej linii WS informuje o tym, dla którego kanału audio (R lub L) są przeznaczone. Nie sposób zwięźle przedstawić wszystkie niuanse związane z tym interfejsem, dlatego zainteresowanych odsyłam przede wszystkim do dokumentacji procesora (UM10360). Na schemacie można zobaczyć, że linia MOSI została połączona z wyjściem I2STX_SDA, dzięki czemu można stosować te interfejsy zamiennie.

Wróćmy teraz do opisu sposobu generowania obrazu. Wejście U4A jest na stałe połączone z masą, natomiast wejście U4B z dodatnią szyną zasilania. Dzięki temu sygnał na wyjściu U4B jest zanegowany, a na wyjściu U4A ma ten sam stan, co na wejściu obu tych buforów. Zastosowanie prostego negatora tylko na jednej linii doprowadziłoby do „bardzo dużego" przesunięcia w czasie (fazie) obu sygnałów ze względu na opóźnienie bramek dochodzące nawet do 20ns, a czas trwania jednego piksela na ekranie to niespełna 40ns. Dzięki dwóm buforom, w identyczny sposób propagującym sygnały, otrzymujemy takie samo opóźnienie. Rezystory R3 i R4 ustalają odpowiednie poziomy napięcia przy zatkanych buforach (kiedy wyjście jest w stanie wysokiej impedancji).

Wejścia zezwolenia buforów U3A i U3B są zatem sterowane sygnałami o przeciwnych poziomach logicznych. Odpowiadające sobie wyjścia buforów A i B są ze sobą połączone. Dzięki temu, że nigdy oba bufory nie będą otwarte jednocześnie, a wyjścia w stanie zatkanym mają wysoką impedancję, nie nastąpi tu żadna kolizja. Tym oto sposobem z połączenia obu pomysłów otrzymujemy 4-bitową informację kodującą na zmianę kolor znaku i jego tła z częstotliwością ~25,175MHz, czyli dla 640 pikseli w linii ;-).

Jedyne, co pozostaje, to przetworzyć informację bitową na postać zrozumiałą dla analogowego interfejsu VGA. Rezystory R6, R7 i R8 tworzą wraz z rezystorami na końcach linii RGB w monitorze 3 dzielniki napięcia. Tym samym przykładowo poziom wysoki na wyjściu Y1 bufora U3 odpowiadający napięciu 5V zostanie podzielony do poziomu około 1V, co odpowiada pełnej jasności koloru czerwonego. Oczywiście poziom niski będzie odpowiadał napięciu 0V i całkowicie wygaszonemu kolorowi. Tym sposobem otrzymujemy 8 kombinacji. Czwarty bit kodujący dwupoziomową jasność został dodany dzięki obecności diod Schottky'ego D1, D2 i D3.

Kiedy na wyjściu Y4 panuje stan wysoki, odpowiada to kolorowi jasnemu. Diody są spolaryzowane zaporowo i nie przewodzą, a tym samym informacja o kolorze jest pobierana wprost z niższych 3 bitów. Sytuacja zmieni się, kiedy na wyjściu Y4 pojawi się stan niski. W wypadku wystąpienia stanu wysokiego na którejś linii składowego koloru, odpowiednia dioda zacznie przewodzić i odłoży się na niej w przybliżeniu 0,3–0,5V. Przykładowo stan wysoki na wyjściu Y1 spowoduje przewodzenie diody D3, a tym samym na linii koloru czerwonego pojawi się napięcie znacząco niższe od tego oznaczającego pełny kolor. Dzięki temu wszystkie 3 składowe mogą zostać „przycięte" jednocześnie, a tym samym otrzymujemy ciemniejszy odcień wszystkich 7 możliwych kombinacji kolorów składowych. Sygnały RGB są doprowadzone do odpowiednich pinów złącza VGA.
Ot i cała filozofia ;-)

Do tego złącza doprowadzone są także sygnały synchronizacji poziomej i pionowej. Sygnały te po wyjściu z procesora trafiają na identyczne bufory, jak sygnał z linii MOSI/I2STX_SDA. Ma to jedynie na celu dodanie podobnego opóźnienia i uniknięcie kłopotliwego dostrajania synchronizacji do sygnału wizyjnego.

Złącze, nazwane UART, służy do doprowadzenia sygnałów komunikacji RX i TX oraz zasilania 5V, które następnie jest stabilizowane do wartości 3,3V przez układ U2. Goldpiny opisane jako JPS służą do ustalenia prędkości komunikacji za pomocą czterech jumperów, a ponadto dostępne są tu sygnały RESET i ISP, które w połączeniu z liniami RX i TX służą do programowania procesora.

Tak w zasadzie wygląda główna część hardware'u, jednak w trakcie rysowania płytki uznałem, że można by jeszcze na niej zmieścić złącze karty SD/MMC. Karty takie mają bardzo dużą pojemność, szczególnie w odniesieniu do rozmiaru pojedynczej „klatki" obrazu VGA czyli maksymalnie 6800 bajtów w trybie naprzemiennego wysyłania znaku i koloru (szczegóły w części opisującej oprogramowanie). Tak więc, aby cały ogrom miejsca na karcie się nie marnował, postanowiłem dodać do projektu „poboczną" funkcjonalność w postaci odtwarzania (prawie)nieskompresowanego dźwięku z nietypową częstotliwością próbkowania nieco ponad 31kHz. Próbki odczytywane z karty są cyklicznie wysyłane na wbudowany w procesor przetwornik CA. Analogowy sygnał jest następnie wzmacniany w układzie U6, będącym wzmacniaczem audio, który przy dostępnym napięciu zasilania dysponuje mocą maksymalną około 300mW. Potencjometr PR1 służy do regulacji głośności, a rezystor R10 ogranicza poziom sygnału podawany na wejście wzmacniacza. Wzmocniony sygnał audio jest dostępny na standardowym złączu jack 3,5mm stereo (oba kanały R i L połączone).

Oprogramowanie

Oprogramowanie układu powstało w darmowym środowisku *LPCXpresso*, będącym produkcją firmy *code_red* na zlecenie NXP, zatem z pełnym wsparciem producenta mikrokontrolerów LPC. Jakiś czas temu na łamach EdW spotkałem się z bardzo niepochlebną opinią na temat tego środowiska. Uważam jednak, że pomimo drobnych wad w niektórych wersjach, jest to dość spójne i przyjazne środowisko – wystarczy tylko trochę się do niego przyzwyczaić. Program jest oparty na popularnym Eclipsie, z którym współpracują rozmaite kompilatory na bodaj każdą możliwą platformę. Środowisko można pobrać za darmo po zarejestrowaniu na stronie:
www.code-red-tech.com

Praktycznie cały program powstał w języku C, a jego napisanie zajęło mi równo miesiąc. Złożyło się na to wiele czynników, takich jak długotrwałe testy różnych rozwiązań i użycie elementów, których tak naprawdę nigdy wcześniej nie potrzebowałem, na przykład DMA, o czym dalej.

Zacznijmy od krótkiego opisu funkcji *main*, gdyż nie jest to w zasadzie najistotniejsza część programu. Po starcie procesor ląduje w pliku *cr_startup_lpc17.c*, gdzie dokonuje się podstawowa konfiguracja pamięci, a następnie przechodzi do funkcji *main*. Pierwszą ważną rzeczą jest wywołanie funkcji *frequency*, która jest odpowiedzialna za ustalenie źródła sygnału zegarowego (po resecie aktywny jest wewnętrzny oscylator

4MHz) i jego odpowiednie przemnożenie, a także rozdystrybuowanie do poszczególnych części procesora. Użyłem w tym celu własnej funkcji, aby mieć absolutną pewność, że wszystko jest jak należy, bowiem częstotliwość pracy rdzenia i peryferii ma w tym projekcie kluczowe znaczenie.

Dalej następuje inicjalizacja pozostałych części procesora i nie ma sensu specjalnie rozpisywać się na ten temat. Przejdźmy teraz do opisu fragmentu, gdzie pojawia się cała „magia" tworzenia obrazu. Jak nietrudno się domyślić, ma to miejsce w pliku *vga.c*. Przy tworzeniu obrazu najbardziej istotną sprawą jest prawidłowa synchronizacja, a w szczególności synchronizacja pozioma, ponieważ występuje ona przy każdej z 480 linii obrazu (a w zasadzie w każdej z 525 linii).

Aby w mikrokontrolerze zachować ścisłe zależności czasowe, należy oczywiście użyć timera. Ponieważ w każdej linii występuje impuls synchronizacji na wyjściu HSYNC, postanowiłem jako licznika użyć dość specyficznego komponentu, jakim jest specjalny licznik dedykowany do tworzenia sygnałów PWM. Sam impuls synchronizacji poziomej jest tworzony właśnie jako sygnał PWM na wyjściu procesora PWM1.4 (P1.23). Poza tym przy każdym przepełnieniu licznika wywoływane jest przerwanie, którego obsługą zajmuje się najważniejsza funkcja w programie o nazwie *PWM1_IRQHandler*.

Funkcja ta jest wywoływana z częstotliwością 31,469kHz (co 31,77us) na początku generowania każdej linii obrazu. Po wejściu w procedurę obsługi przerwania sprawdzany jest numer aktualnej linii i na tej podstawie podejmowane są różne akcje np. software'owe generowanie impulsu synchronizacji pionowej. Odbywa się to bardzo prosto – jeżeli wyświetlana jest linia o numerze 491, na wyjściu VSYNC (P1.24) pojawia się stan niski i jest tam utrzymywany aż do linii o numerze 493, czyli przez czas dokładnie odpowiadający dwóm liniom obrazu.

W ostatniej linii o numerze 524 występuje poboczna funkcja ładowania kolejnej próbki audio do przetwornika DAC – tu wyjaśnia się nietypowa częstotliwość próbkowania sygnału audio. Jedynym sposobem na utrzymanie stałej częstotliwości próbkowania bez zakłócania generowania obrazu było właśnie umieszczenie wysyłania próbek wewnątrz procedury generacji obrazu. Zajmuje się tym funkcja *audio*. Na koniec zerowany jest numer linii, odpowiednio ustawiany jest wskaźnik na tablicę z treścią do wyświetlenia (szczegóły dalej) oraz kasowana jest flaga przerwania.

Dla wszystkich linii o numerach od 0 do 479 funkcja przechodzi do właściwego procesu ładowania i wyświetlania obrazu. Na tym etapie warto wspomnieć, że cała „klatka" obrazu jest przechowywana w trzech tablicach. Kolor znaków i tła jest umieszczony w tablicy *kolor*, natomiast wygląd znaków w tablicach *obraz_a* i *obraz_b*, przy czym w pierwszej tablicy umieszczone są znaki z górnej, a w drugiej z dolnej połowy ekranu.

Taki sposób podziału jest związany ze strukturą pamięci RAM w mikrokontrolerze. Z grubsza można przyjąć, że jest ona podzielona na dwie części po 32KB, pomiędzy którymi nie jest zachowana ciągłość adresów. Innymi słowy początek drugiej części RAMu znajduje się znacznie dalej w przestrzeni adresowej niż koniec pierwszej części. Kompilator domyślnie korzysta z pierwszej połowy i tam umieszcza automatycznie wszystkie zmienne.

Ilość pamięci, potrzebna do przechowania wyglądu znaków, to: 480 x 80 = 38400 bajtów (480 linii obrazu po 80 znaków w linii, przyjmując, że na każdy znak potrzebny jest jeden bajt w linii. W trybie SPI domyślnie 85 znaków w linii, czyli 40800 bajtów). Od razu widać, że tak duża tablica nie zmieści się w całości w jednej części RAM-u i dlatego została podzielona na pół. Umieszczenie danych w drugiej połowie RAM-u może odbyć się na kilka sposobów np. przez modyfikację skryptu linkera. Ja jednak przyjąłem bardzo proste i mam nadzieję przejrzyste rozwiązanie, polegające na zdefiniowaniu adresu początku tej pamięci w stałej *RAM2_BASE* i następnie przypisanie tego adresu do wskaźnika *obraz_b*, którego dalej w programie można użyć jak najzwyklejszej tablicy. Obie tablice obrazu zadeklarowane są w pliku *font.c*, o którym za chwilę. Tablica przechowująca kolor ma rozmiar: 80 x 40 = 3200 bajtów (dla trybu SPI 85x40=3400). Ona również występuje w istocie jako wskaźnik na adres w drugiej części RAM-u, aby nie zajmować przestrzeni wykorzystywanej bezpośrednio przez kompilator. Umieszczona została tuż za końcem tablicy *obraz_b*.

Przejdźmy teraz do opisu procesu wyświetlania obrazu. Jak już wiemy, informacja o „zaświeconym" bądź „zgaszonym" stanie pikseli wysyłana jest przez interfejs I2S (lub SPI, a konkretniej SSP0). Teoretycznie cykliczne ładowanie kolejnych znaków do wysłania mogłoby zadziałać, ale nie zapominajmy, że musimy jeszcze podać informację o kolorze. Zrobienie tego w jednej chwili byłoby trudne, szczególnie bez schodzenia do poziomu asemblera, który w tym wypadku do łatwych nie należy (także ze względu na trudne do określenia zależności czasowe w wykonywaniu instrukcji). Z pomocą przychodzi wbudowany w procesor układ DMA, czyli blok tak zwanego bezpośredniego dostępu do pamięci.

Poza wieloma dodatkowymi parametrami, najważniejsze dla układu DMA jest źródło i przeznaczenie danych. W tym wypadku źródłem danych jest oczywiście pamięć RAM, a konkretniej adres zawarty we wskaźniku *line_addr*, który w zależności od numeru wyświetlanej linii zawiera adres początku tablicy *obraz_a* dla linii o numerze 0 lub *obraz_b* dla 240. Adres ten na końcu procedury wyświetlania jest cyklicznie powiększany o 80 (liczba znaków w linii).

Przeznaczeniem danych jest rejestr FIFO I2S (lub nadawczo-odbiorczy interfejsu SSP0). Dla układu DMA nie stanowi to problemu, gdyż jest on w stanie zarządzać bezpośrednim (czyli bez ingerencji rdzenia procesora) transferem zarówno pomiędzy różnymi częściami pamięci RAM, jak i wybranymi peryferiami. W trakcie testów wystąpił problem z początkowymi danymi w kolejce FIFO I2S, ładowanymi przez DMA. Dlatego pierwsze dwa bajty obrazu danej linii są najpierw ładowane „ręcznie". Trzecim istotnym parametrem jest liczba danych do przesłania, wynosząca tutaj równo 80 bajtów, czyli tyle, ile znaków występuje w jednej linii.

W ten oto sposób otrzymujemy w pełni autonomiczne wysyłanie wyglądu znaków na ekran, czyli mamy już obraz czarno-biały. Nałożenie koloru następuje metodą dość brutalną, mianowicie wysłanie koloru każdego znaku to osobny zestaw 3 instrukcji. Ilość wygenerowanego kodu nie stanowi żadnego problemu, natomiast unikamy problemów z określeniem czasu wykonywania pętli, a wręcz otrzymujemy możliwość trymowania opóźnień dla każdego znaku z osobna.

Dokładniej rzecz ujmując, dla koloru każdego znaku występuje odpowiednie opóźnienie, najczęściej w postaci makra *NOP_9*, które jak nietrudno się domyślić, generuje 9 „pustych" cykli procesora, podczas których nie są wykonywane żadne operacje, oczywiście poza tymi autonomicznymi – sprzętowymi jak PWM i DMA. Samo wysłanie informacji o kolorze polega na zapisaniu do 8 wybranych bitów portu odpowiedniej wartości kodującej kolor tła i znaku. Porty w procesorach ARM są najczęściej 32-bitowe, ale w naszym projekcie wykorzystywana jest możliwość dostępu do jednej z czterech grup 8-bitowych, tj. P2.0 do P2.7. Możliwy jest także dostęp 16-bitowy (2 grupy).

Wartość koloru, jak już wspominałem, pobierana jest z globalnej tablicy o nazwie *kolor*. Jednak dla uproszczenia procedury adresowania, przed wyświetleniem kolejnej linii, adres z tej globalnej tablicy, odpowiadający informacji o kolorach aktualnej grupy 80 (dla SPI 85) znaków, jest ładowany do statycznego lokalnego wskaźnika *kolor1*. Następnie podczas wyświetlania kolejnych kolorów wskaźnik ten jest inkrementowany. Ładowanie tego samego adresu odbywa się łącznie 12 razy, bo ze względu na wysokość czcionki, wynoszącą 480 / 40 = 12, należy właśnie 12 razy wyświetlić ten sam zestaw kolorów.

Na końcu procedury wyświetlania, inkrementowany jest numer aktualnej linii i ewentualnie następuje „przełączenie" na drugą

tablicę obrazu. W pierwotnej wersji artykułu w tym miejscu umieściłem dość szeroki opis problemu nieciągłości danych wysyłanych przez SPI i związanymi z tym parametrami obrazu, takimi jak liczba znaków, wynosząca nietypowo 85 i szerokość czcionki, wynosząca 6 pikseli. Ponieważ zastosowanie interfejsu I2S wyeliminowało ten problem, proponuję przyjąć tylko do wiadomości, że taki problem występuje dla SPI. W pliku *main.h* umieszczona jest definicja, której zmiana z wartości *I2S_MODE* na *SPI_MODE* spowoduje kompilację programu w trybie SPI i umożliwi bardziej wnikliwe zapoznanie się z różnicami.

W projekcie zastosowałem dość „surową" i nieco „futurystyczną" czcionkę, która jest zdefiniowana w pliku *font.h*. W pliku *font.c* natomiast znajdują się procedury „renderowania" fragmentu tekstu lub pojedynczego znaku. Jest tu oczywiście tworzony jedynie kształt znaku, natomiast kolor jest nakładany w funkcjach z pliku *vga.c*. Ze względu na uproszczenie w projektowaniu płytki drukowanej, poszczególne bity kodujące kolor są poprzesuwane względem formatu podawanego przy wprowadzaniu tekstu z zewnątrz i z tego powodu w funkcjach *vga_render_xy* i *vga_char_xy* wykonywane są odpowiednie przetasowania na bajcie koloru, polegające na podstawieniu wcześniej obliczonej wartości, odpowiadającej przesunięciu bitów. Tym zadaniem zajmuje się makro *COLOR_TRANS*.

Funkcja *font_char_xy* zapisuje wygląd pojedynczego znaku do odpowiednich komórek w pierwszej bądź drugiej tablicy obrazu. Jak widać, cały proces jest dość prosty i polega na odczycie kolejno każdej z 12 linii znaku i przepisaniu tych danych z tablicy *font* do tablicy *obraz_a* lub *obraz_b*. Ta sama procedura wykorzystywana jest w funkcjach *font_render* i *font_render_xy*, przy czym ładowanie danych zachodzi dla wielu znaków, czyli dla fragmentu bądź dla całego dostępnego tekstu (3200/3400 znaków).

Bezpośrednio w programie użytkownikowi, zamiast funkcji z pliku *font.*c, wygodniej będzie posłużyć się wspomnianymi *vga_render_xy* i *vga_char_xy*, które w parametrach przyjmują tekst (lub pojedynczy znak), jego pozycję na ekranie oraz kolor i najpierw wywołują odpowiednie procedury renderujące z pliku *font.c,* a następnie wypełniają odpowiednio tablicę koloru.

Na koniec opisu oprogramowania jeszcze kilka słów na temat odtwarzania audio. Jest to część eksperymentalna i należałoby ją rozbudować o dodatkową funkcjonalność, która obecnie ogranicza się do ciągłego odtwarzania próbek zapisanych na karcie SD pomiędzy dwoma adresami określonymi w strukturze *audio_ctrl* jako *start_addr* i *end_addr*.

Za ładowanie próbek do jednego z dwóch globalnych buforów *audio_data* odpowiedzialna jest funkcja *audio_load*. Jest ona wywoływana w głównej pętli programu w czasie wolnym od innych operacji wyznaczonym przez wartość flagi *free_time*. Flaga ta jest ustawiana na 1 na początku 481 linii obrazu, a następnie jest zerowana w linii o numerze 525. W tym okresie, w przerwaniu *PWM1_IRQHandler* praktycznie nie są wykonywane żadne zadania, poza ustawieniem linii synchronizacji pionowej i ładowaniem próbki dźwięku do przetwornika, co daje możliwość oddania kontroli nad programem do głównej pętli na czas około 1,4ms, w którym można wykonywać dowolne inne operacje.

Wspomniana funkcja *audio_load* odczytuje dane z kolejnej strony karty SD/MMC i ładuje je naprzemiennie do jednego z buforów. Odczyt z karty jest dość szybki i ma szansę w całości wykonać się w wolnym czasie. Mimo to zastosowany został podwójny bufor, ze względu na konieczność ciągłego odtwarzania dźwięku także w trakcie odczytu kolejnych jego próbek. Tak więc kiedy jeden bufor zapełniany jest nowymi danymi z karty, drugi zawiera te aktualnie odtwarzane.

Samych funkcji obsługi karty pamięci nie będę szczegółowo omawiał – wszystkie są zawarte w pliku *sd_mmc.c*. Dodam jedynie, że po poprawnej inicjalizacji karta pracuje z prędkością 16,6 MHz. Dla dźwięku odtwarzanego bezpośrednio z karty bardzo istotny w tym zastosowaniu okazuje się jeden z jej parametrów, a mianowicie wielkość strony, wynosząca 512 bajtów.

Aby poprawnie odtworzyć dźwięk, należy ładować przetwornik DAC kolejnymi próbkami z taką samą częstotliwością, z jaką dźwięk był wcześniej próbkowany. Tu pojawia się pierwsza trudność. Próba zastosowania dźwięku o jednej ze standardowych częstotliwości próbkowania, jak np. 24kHz, 32kHz czy 44,1kHz, wymaga zastosowania osobnego licznika, i przerwania, które wywoływane z tą częstotliwością będzie ładowało kolejną próbkę.

Nic prostszego, ale nie zapominajmy, że procesor jest przez większość czasu mocno obciążony przez generowanie obrazu, a ten proces jest niezwykle czuły na wszelki odchyłki czasowe. Tak więc wywołanie przerwania w trakcie trwania generacji obrazu zupełnie by go zrujnowało i jedynym sensownym rozwiązaniem (może poza zabiegami sprzętowymi, dostępnymi w niektórych mikrokontrolerach) wydaje się synchronizacja odtwarzania dźwięku z generowanym obrazem.

I tak dochodzimy do nietypowej częstotliwości próbkowania dźwięku 31,469kHz, czyli dokładnie tyle, ile synchronizacja pozioma. Wynika to oczywiście z faktu, że kolejne próbki ładowane są poprzez funkcję *audio*, a ta wywoływana jest zawsze na końcu aktualnej linii obrazu. W tym miejscu pojawia się drugi znacznie istotniejszy problem: pamiętamy, że na jednej stronie karty SD zapisanych jest 512 bajtów. Jednak linii w obrazie jest 525... Chcąc uzyskać ciągły dźwięk, musielibyśmy w trakcie generowania obrazu odczytywać kolejną stronę karty. Niestety czasu wolnego pomiędzy końcem jednej i początkiem kolejnej linii jest bardzo mało (mniej niż 1us) i taka operacja byłaby bardzo trudna, jeżeli w ogóle możliwa do przeprowadzenia.

Skoro w czasie trwania jednej klatki obrazu dysponujemy 512 próbkami, a do odtworzenia mamy ich 525, skąd wziąć brakujące 13 próbek? Cóż, możliwości jest wiele i w większości sprowadzają się one do pojęcia interpolacji, czyli odtworzenia brakujących danych na podstawie tych dostępnych np. dla uzyskania większej dokładności. W tym wypadku można to także traktować jako formę lekkiej stratnej kompresji informacji o stopniu upakowania 0,975, czyli tracimy i próbujemy odzyskać około 2,5% danych.

Jak już pisałem, odtwarzanie dźwięku jest w tym projekcie jedynie funkcją dodatkową i dlatego nie spędziłem zbyt wiele czasu nad zastanawianiem się nad algorytmami interpolacji. Zastosowałem bardzo prosty zabieg. Wewnątrz funkcji *audio* kolejne próbki dźwięku są ładowane do przetwornika, przy czym następuje to jedynie wtedy, gdy numer próbki nie jest podzielny bez reszty przez 36. Innymi słowy, co 36 próbek numer kolejnego bajtu nie jest inkrementowany, więc do przetwornika ładowana jest po raz drugi ta sama wartość, co w efekcie daje dwukrotne wydłużenie czasu odtwarzania poprzedniej próbki. Na 525 wartości załadowanych do przetwornika mamy zatem 14 „sztucznych" próbek. Sumując to z 512 bajtami na stronie, otrzymujemy o jedną próbkę za dużo, tak więc ostatni bajt na każdej stronie nie jest wykorzystywany.

Takie rozwiązanie może nie jest zbyt eleganckie i wymaga sporo dodatkowych zabiegów przeprowadzonych wstępnie na plikach audio, ale pozwala uzyskać przyzwoite efekty przy małym obciążeniu procesora. Choć zdecydowanie nie jest to sprzęt dla audiofilów, to na pewno wyjście audio można wykorzystać, jeżeli nie do odtwarzania muzyki, to przynajmniej do rozmaitych dźwięków kontrolnych itp.

Za miesiąc, w drugiej części artykułu opisane zostaną zagadnienia związane z montażem i uruchomieniem.

Filip Rus
filip.rus@livelights.pl