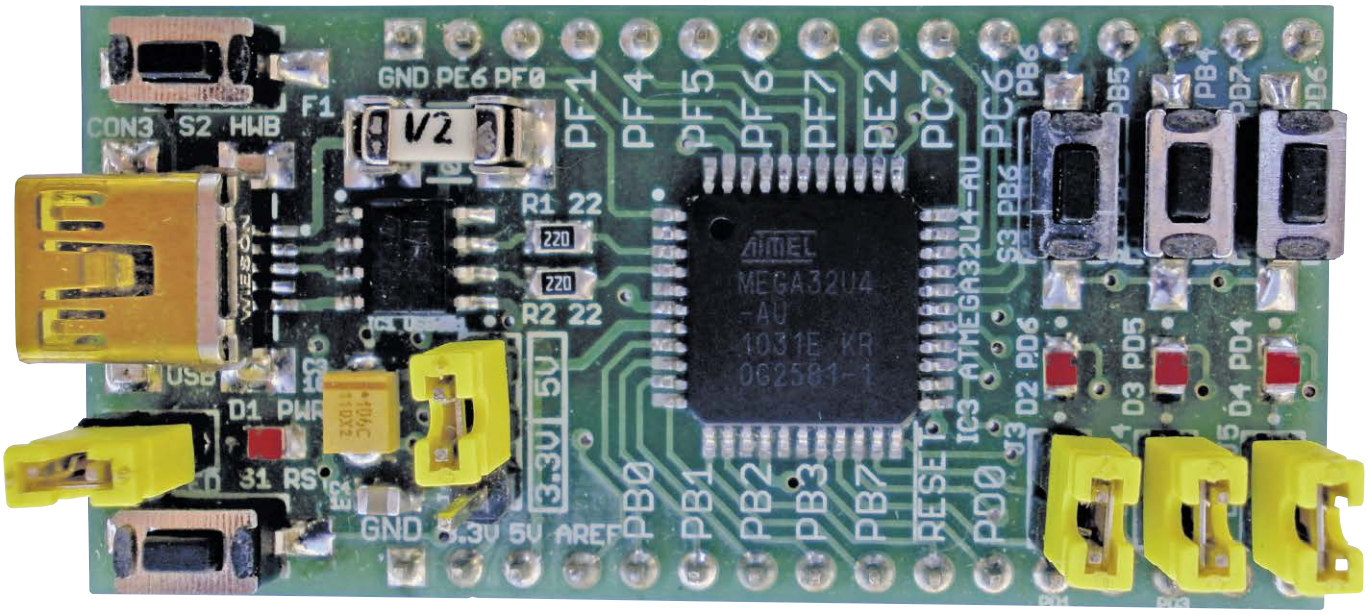


ATmega32U4-DIPMOD

AVT 5468

„Breakout board" dla mikrokontrolerów AVR ATmega16U4/32U4

Prezentowany projekt powstał w związku z chęcią poznania możliwości komunikacyjnych mikrokontrolerów AVR z wbudowanym interfejsem USB. W Elektronice Praktycznej były już opisywane projekty wykorzystujące AVR-y oraz USB, jednak z dotychczas przedstawionych wszystkie miały pewne niedociągnięcia. Między innymi wykorzystywały programowy stos USB (ograniczający maksymalną prędkość transmisji i pozostawiający niewiele mocy obliczeniowej na pozostałe zadania), albo miały niekonwencjonalny interfejs sprzętowy, który czasami może mieć problemy ze zgodnością ze standardem. Przedstawione właściwości klasyfikują te rozwiązania jako możliwe do wykorzystania raczej w mniej wymagających aplikacjach lub zabawkach, bardziej niż w profesjonalnych urządzeniach, które muszą być przede wszystkim niezawodne. Rozwiązaniem opisanych problemów jest zastosowanie mikrokontrolera mającego sprzętowy interfejs USB.

***Rekomendacje:** doskonały projekt do nauki technik komunikowania się oraz testowania rozwiązań.*

Dzięki sprzętowemu USB zyskuje się zgodność ze standardem oraz prostsze i pewniejsze sposoby oprogramowania tego interfejsu. W projekcie zastosowano mikrokontroler ATmega32U4, można również użyć jego odpowiednika o mniejszej pamięci Flash, tj. ATmega16U4. Płytkę drukowaną zaprojektowano w ten sposób, że pasuje do standardowych płytek stykowych umożliwiając szybką budowę prototypów.

Opis urządzenia

Projektując urządzenie starałem się uczynić go jak najbardziej bezpiecznym i uniwersalnym, pamiętając jednocześnie, że jest głównie przeznaczony do umieszczania na płytce stykowej. Nie do przecenienia jest fakt, że układ może być zasilany i programowany bezpośrednio z portu USB komputera, co w wielu sytuacjach upraszcza i uprzyjemnia pracę.

W ofercie AVT*
AVT-5468 A AVT-5468 B
AVT-5468 C

Podstawowe informacje:

- Napięcie zasilania: 5 V z USB lub 3,3 V, lub 7…9 V z zasilacza zewnętrznego.
- Mikrokontroler ATmega16U4 lub ATmega32U4.
- Programowanie za pomocą USB.
- Pasuje do płytek stykowych.

Dodatkowe materiały na FTP:

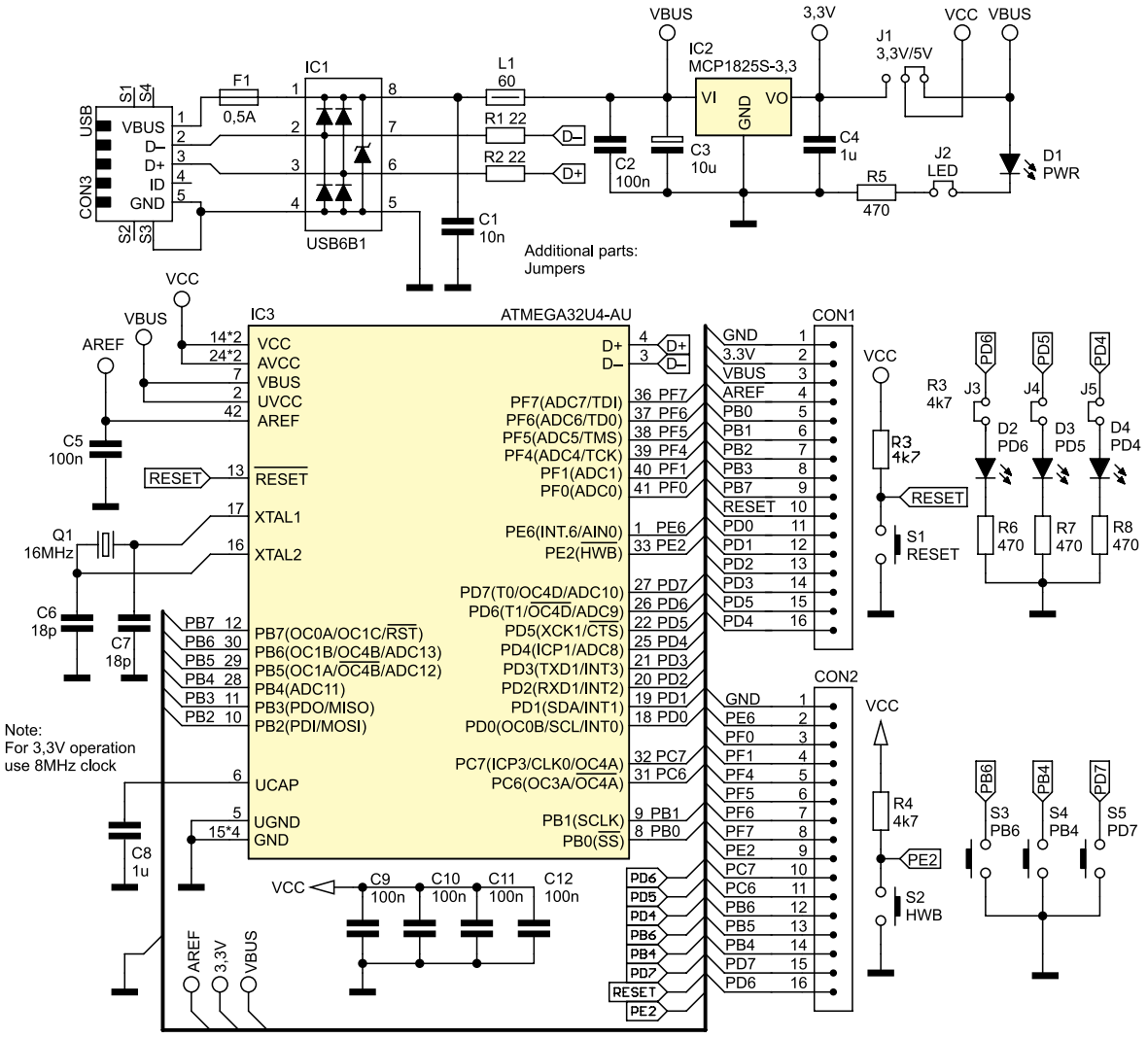
ftp://ep.com.pl, user: 42850, pass: 3063yuhc

- wzory płytek PCB

Projekty pokrewne na FTP:
(wymienione artykuły są w całości dostępne na FTP)

AVT-1795	Minimoduł STK_Mega32USB (EP 4/2014)
AVT-1777	TinyMini861 – miniaturowy moduł ATtiny861 (EP 10/2013)
AVT-1752	ATmega128 na płytce ewaluacyjnej AVT5311 (EP 8/2013)
AVT-1665	Moduł wyświetlacza LCD z mikrokontrolerem Atmega8 (EP 2/2012)
AVT-1622	Minimoduł z ATmega8 (EP 6/2011)
AVT-1610	Minimoduł z ATtiny2313 (EP 3/2011)
AVT-1430	ATmega8 w AVT-992 (EP 7/2006)

* Uwaga:
Zestawy AVT mogą występować w następujących wersjach:
AVT xxxx UK to zaprogramowany układ. Tylko i wyłącznie. Bez elementów dodatkowych.
AVT xxxx A płytka drukowana PCB (lub płytki drukowane, jeśli w opisie wyraźnie zaznaczono), bez elementów dodatkowych.
AVT xxxx A+ płytka drukowana i zaprogramowany układ (czyli połączenie wersji A i wersji UK) bez elementów dodatkowych.
AVT xxxx B płytka drukowana (lub płytki) oraz komplet elementów wymieniony w załączniku pdf
AVT xxxx C to nic innego jak zmontowany zestaw B, czyli elementy wlutowane w PCB. Należy mieć na uwadze, że o ile nie zaznaczono wyraźnie w opisie, zestaw ten nie ma obudowy ani elementów dodatkowych, które nie zostały wymienione w załączniku pdf
AVT xxxx CD oprogramowanie (nieczęsto spotykana wersja, lecz jeśli występuje, to niezbędne oprogramowanie można ściągnąć, klikając w link umieszczony w opisie kitu)
Nie każdy zestaw AVT występuje we wszystkich wersjach! Każda wersja ma załączony ten sam plik pdf! Podczas składania zamówienia upewnij się, którą wersję zamawiasz! (UK, A, A+, B lub C). *http://sklep.avt.pl*



Rysunek 1. Schemat ideowy układu ATmega32U4-DIPMOD

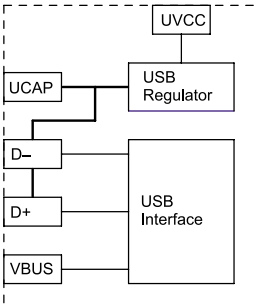
Na **rysunku 1** pokazano schemat ideowy płytki prototypowej. Przyłącza się ją do komputera z wykorzystaniem złącze CON3 typu USB Mini B. Bezpiecznik F1 o wartości 500 mA ma na celu ochronę portu USB komputera w wypadku zwarcia na płytce, gdy jest ona z niego zasilana (500 mA to maksymalna wartość natężenia prądu, którą zgodnie ze standardem można pobierać z portu USB). Wartości rezystancji R1 oraz R2 wzięto z karty katalogowej mikrokontrolera.

Układ IC1 (USB6B1) produkcji STMicroelectronics chroni przed przepięciami mogącymi pojawiać się na linii transmisyjnej. Dzięki wbudowanym diodom zapewnia on ograniczenie wysokich napięć mogących wystąpić na przewodach sygnałowych i zasilających. Poza interfejsem USB, układ nadaje się m.in. do ochrony interfejsu RS-485, a jego podstawową zaletą jest mała ilość miejsca zajmowanego na płytce w porównaniu z rozwiązaniami opartymi o elementy dyskretne.

Kondensatory C1 i C2, C9…C12 i dławik L1 filtrują zaburzenia mogące wystąpić na szynie zasilania. Dławik L1 to tzw. koralik przeciwzakłóceniowy (*ferrite bead*), którego rolą jest absorbowanie prądu o wysokiej częstotliwości i wydzielanie go w postaci ilości ciepła. Podana na schemacie wartość 60 oznacza, że koralik ma impedancję 60 Ω przy częstotliwości 100 MHz (tę wartość warto zawsze konsultować z dokumentacją danego modelu). Wartości elementów C1, L1, C2 zostały dobrane zgodnie z zaleceniami odszukanymi w dokumentacji firm FTDI [1] oraz Texas Instruments [2].

Płytkę wyposażono w stabilizator napięcia 3,3 V, dzięki któremu jest możliwy wybór napięcia zasilającego mikrokontroler między +3,3 V ze stabilizatora a +5 V pobieranym z USB. Na blok zasilania składają się elementy kondensatory C3 i C4, układ scalony IC2, rezystor R5, dioda D1, złącza J1 i J2. Pojemność kondensatora C3 (10 µF) dobrano zgodnie z zaleceniami dokumentacji mikrokontrolera. Jeśli miałaby być większa, dla zgodności ze standardem USB niezbędne może okazać się zastosowanie *soft startu* w projektowanym urządzeniu. Pojemność C4 (1 µF) to minimalna zalecana dla IC2. Układ IC2 (MCP1825) wybrano ze względu na relatywnie mały (maks. 220 µA) pobierany prąd, co może mieć znaczenie przy testowaniu urządzeń USB wykorzystujących tryb czuwania, gdy pobierany prąd nie powinien przekraczać wartości 500 µA. Biorąc pod uwagę prąd pobierany przez wbudowane w mikrokontroler rezystory podciągające magistralę USB (ok. 200 µA) dla mikrokontrolera pozostaje około 80 µA. Rola zworki J2 wydaje się oczywista – umożliwia ona odłączenie diody D1 (sygnalizującej zasilanie), pobierającej w tym wypadku znaczny prąd.

Wyboru napięcia zasilającego mikrokontroler dokonuje się zworką J1. Układ

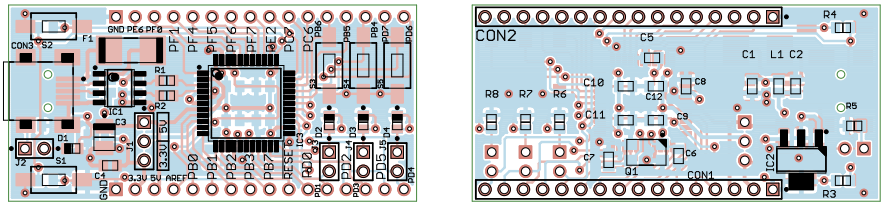


Rysunek 2. Schemat (częściowy) bloku zasilania mikrokontrolera ATmega32U4

zasilania mikrokontrolera wymaga kilku słów wyjaśnienia, w czym pomocny będzie **rysunek 2** przedstawiający fragment budowy wewnętrznej bloku USB mikrokontrolera ATmega32U4. Zgodnie ze standardem, napięcie zasilające interfejsu USB wynosi około +5 V, zaś napięcie poziomu wysokiego na liniach sygnałowych (D–, D+) wynosi około +3,3 V, dlatego mikrokontroler ma wbudowany stabilizator napięcia +3,3 V, umożliwiający odpowiednie zasilanie linii sygnałowych. Wejście (UVCC) oraz wyjście (UCAP) tego stabilizatora są dostępne na wyprowadzeniach mikrokontrolera. W prezentowanym urządzeniu, wejście stabilizatora (UVCC) przyłączono na stałe do napięcia VBUS (tj. +5 V z USB), natomiast do wyjścia (UCAP) jest przyłączony jedynie kondensator filtrujący C8. Dzięki takiej konfiguracji, mikrokontroler może generować wymagane poziomy napięć interfejsu USB niezależnie od napięcia zasilającego (VCC). Dzięki zworce J1, napięcie VCC mikrokontrolera może zostać ustalone na +5 V z USB lub +3,3 V ze stabilizatora IC2. Warto zaznaczyć, że mikrokontroler może pracować w różnych konfiguracjach zasilania, które omówiono szerzej

Wykaz elementów

Rezystory: (SMD 0805)
R1, R2: 22 Ω
R3, R4: 4,7 kΩ
R5…R8: 470 Ω
Kondensatory: (SMD 0805)
C1: 10 nF
C2, C5, C9…C12: 100 nF
C3: 10 µF (SMD „A")
C4, C8: 1 µF
C6, C7: 18 pF
Półprzewodniki:
D1…D4: diody LED SMD
IC1: USB6B1 (SO-8)
IC2: MCP1825S-3.3 (TO-225)
IC3: ATmega32U4-AU lub ATmega64U4-AU (TQFP-44)
Inne:
CON1, CON2: goldpin 16×1
CON3: gniazdo USB „A" (SMD)
F1: bezpiecznik polimerowy 0,5 A
J1: goldpin 1×3+zworka
J2: golpin 1×2+zworka
L1: 60 µH
Q1: kwarc 16 MHz
S1…S5: mikroprzycisk SMD



Rysunek 3. Schemat montażowy układu ATmega32U4-DIPMOD

w nocie katalogowej układu. Widoczne na rys. 2 wyprowadzenie VBUS umożliwia mikrokontrolerowi detekcję przyłączenia do USB.

Rezonator kwarcowy Q1 oraz kondensatory C6 i C7 ustalają częstotliwość taktowanie mikrokontrolera. Częstotliwość Q1 jest krytyczna i powinna wynosić 8 lub 16 MHz, ponieważ tylko te wartości zapewniają poprawną pracę interfejsu USB. Dodatkowo, na maksymalną częstotliwość taktowania ma wpływ napięcie zasilające: dla 3,3 V wynosi ona 8 MHz, dla 5 V może to 16 MHz. W trakcie użytkowania nie stwierdziłem niestabilnej pracy mikrokontrolera dla częstotliwości 16 MHz przy zasilaniu 3,3 V, jednak nie należy tego przyjmować za regułę.

Na płytce znalazło się również miejsce dla mikroprzełączników S1…S5 oraz diod LED D2…D4. Przycisk S1 służy do zerowania mikrokontrolera, zaś S2 do uruchamiania bootloadera (sposób programowania opisano dalej). Przyciski ogólnego przeznaczenia S3…S5, podobnie jak diody D2…D4 mogą być wykorzystane dowolnie. Zworki J3…J5 pozwalają na odłączenie diod od mikrokontrolera. Linie przycisków S1 i S2 zostały podciągnięte do plusa zasilania za pomocą rezystorów R3 oraz R4, zaś przyciski S3…S5 mogą być podciągnięte za pomocą rezystorów wbudowanych w mikrokontroler.

Na złącza CON1 i CON2 (goldpin) wyprowadzono wszystkie porty mikrokontrolera, sygnał RESET, zasilanie oraz wejście napięcia odniesienia przetwornika A/C. Rozstaw złączy na płytce drukowanej wynosi 22,86 mm (9×2,54 mm), co pozwala na jej łatwe umieszczenie w otworach płytki stykowej.

Montaż i uruchomienie

Schemat montażowy pokazano na **rysunku 3**. Urządzenie składa się głównie z elementów SMD, w związku z tym wymaga nieco wprawy w ich montażu. Na samym początku przestrzegam przed rozpoczęciem montażu od elementów przewlekanych, w tym przypadku złączy CON1, CON2 oraz zworek J1…J5 (należy je montować na końcu). Ponadto, przydatne będą plecionka z topnikiem oraz topnik w pisaku.

Montaż najwygodniej przeprowadzić zaczynając od elementów SMD na stronie górnej. Na tym etapie nie należy montować przycisków S1…S5, bezpiecznika F1, kondensatora C4 oraz złącza USB CON3. Następnie należy przylutować elementy SMD na stronie dolnej. Nieco uwagi wymaga montaż Q1, gdyż jego obudowa zasłania pola lutownicze. Montując go należy ocynować jedno z pól i przylutować do niego jedno z wyprowadzeń Q1. Następnie zaaplikować sporą ilość topnika i przylutować pozostałe wyprowadzenia. W dalszej kolejności należy przylutować złącze USB, lutując najpierw wyprowadzenia obudowy, a potem pozostałe. Teraz można już przylutować wszystkie pozostałe elementy strony górnej.

Złącza CON1 oraz CON2 należy montować w ten sposób, żeby ich wyprowadzenia znajdowały się od strony dolnej płytki. Na początku należy delikatnie (bez wciskania) umieścić płytkę w płytce stykowej, wyrównać złącza i przylutować po dwa przeciwległe wyprowadzenia. Następnie wyjąć płytkę i solidnie przylutować pozostałe wyprowadzenia. Czynność tę warto wykonać bardzo starannie, co zaoszczędzi późniejszych problemów z umieszczaniem płytki w płytce stykowej oraz wyłamywaniem się wyprowadzeń.

Zmontowany układ należy sprawdzić pod kątem występowania zwarć, szczególnie na szynie zasilania USB. Na **listingu 1** zamieszczono program umożliwiający sprawdzenie działania przycisków S2…S5 oraz diod D2…D4. Działanie programu polega na zaświeceniu diody znajdującej się pod przyciskiem w odpowiedzi na jego wciśnięcie. Naciśnięcie przycisku S2 (HWB) powoduje zaświecenie wszystkich diod.

Programowanie

Mikrokontroler ATmega32U4 został fabrycznie wyposażony w bootloader umożliwiający programowanie z wykorzystaniem USB, bez potrzeby korzystania z zewnętrznego programatora. Bootloader w AVR jest umieszczany na końcu pamięci Flash, w tzw. sekcji bootloadera. Zaraz po starcie, mikrokontroler może uruchomić program wgrany przez użytkownika lub bootloader. To, która aplikacja zostanie uruchomiona (główna czy bootloader) zależy od programu i konfiguracji mikrokontrolera.

W fabrycznie nowym mikrokontrolerze ATmega32U4 bootloader uruchamia się wymuszając odpowiednią sekwencję poziomów logicznych na wejściach RESET

Listing 1. Program testujący działanie przycisków i diod LED

```
#include <avr/io.h>
int main( void )
{
    // Piny przycisków jako wejścia z rezystorami podciągającymi.
    DDRB  &= ~( (1<<PB6) | (1<<PB4) ); // S3 & S4
    PORTB |=  ( (1<<PB6) | (1<<PB4) ); // S3 & S4
    DDRD  &= ~(1<<PD7); // S5
    PORTD |=  (1<<PD7); // S5
    DDRE  &= ~(1<<PE2); // S2 (HWB)
    // Piny diod LED jako wyjścia.
    DDRD |= ( (1<<PD6) | (1<<PD5) | (1<<PD4) ); // D2, D3 & D4.
    // Pętla zapalająca diodę w odpowiedzi na naciśnięcie przycisku.
    while( 1 )
    {
        if( !( PINB & (1<<PB6) ) ) // S3 & D2.
        {
            PORTD |= (1<<PD6);
        }
        else
        {
            PORTD &= ~(1<<PD6);
        }
        if( !( PINB & (1<<PB4) ) ) // S4 & D3.
        {
            PORTD |= (1<<PD5);
        }
        else
        {
            PORTD &= ~(1<<PD5);
        }
        if( !( PIND & (1<<PD7) ) ) // S5 & D4.
        {
            PORTD |= (1<<PD4);
        }
        else
        {
            PORTD &= ~(1<<PD4);
        }
        if( !( PINE & (1<<PE2) ) ) // S2 (HWB) & D2, D3 & D4.
        {
            PORTD |= ( (1<<PD6) | (1<<PD5) | (1<<PD4) );
        }
        else
        {
            PORTD &= ~ ( (1<<PD6) | (1<<PD5) | (1<<PD4) );
        }
    }
}
```

i HWB mikrokontrolera, które na płytce przyłączono do przycisków S1 oraz S2. Sekwencja ta wygląda następująco: naciśnięcie S1 (stan niski na wejściu RESET), naciśnięcie S2 (wyzerowane wejście HWB), zwolnienie S1 (ustawione wejście RESET), zwolnienie S2 (ustawione wejście HWB). Po uruchomieniu bootloadera można przejść do programowania układu – w systemie Windows można skorzystać z aplikacji *Atmel FLIP*, a pod Linuksem z aplikacji *dfu-programmer*.

W systemie Windows jest niezbędne zainstalowanie sterownika bootloadera. Jeśli *FLIP* został zainstalowany w domyślnym katalogu, to pliki sterownika znajdują się w katalogu *C:\Program Files\Atmel\Flip 3.4.5\usb*. Po poprawnym zainstalowaniu sterownika można połączyć się z płytką. W tym celu należy uruchomić program *FLIP*, z menu *Device* wybrać opcję *Select*, wybrać docelowy mikrokontroler tj. ATmega32U4, a następnie z menu *Settings* wybrać opcję *Communication*, dalej *USB* i *Open*. Wygląd programu po tych operacjach pokazano na **rysunku 4**. Aby wgrać własny program do pamięci mikrokontrolera, z menu *File* należy wybrać opcję *Load HEX File* i wskazać plik programu w formacie Intel HEX, a następnie nacisnąć przycisk *Run*. Wgrany program można uruchomić naciskając przycisk *Start Application* w programie *FLIP* lub restartując mikrokontroler za pomocą przycisku S1 na płytce. Opisu pozostałych opcji programu *FLIP* należy szukać w pliku pomocy dostępnym w menu *Help*.

Rysunek 4. Wygląd okna programu FLIP

Programowanie płytki w systemie Linux przy wykorzystaniu programu *dfu-programmer* sprowadza się do wydania następujących poleceń:

```
$ dfu-programmer atmega32u4
erase
$ dfu-programmer atmega32u4 flash
plik_programu.hex
```

Ostatecznie, program mikrokontrolera można uruchomić poleceniem:

```
$ dfu-programmer atmega32u4
start
```

Wyjaśnienia dodatkowych opcji programu *dfu-programmer* należy szukać w jego manualu.

Listing 2. Reguły demona *udev* pozwalające programować mikrokontroler użytkownikom należącym do grupy *plugdev* (dotyczy systemu Linux)

```
SUBSYSTEM==”usb”,ACTION==”add”,ATTRS{idVendor}==”03eb”,ATTRS{idProduct}==”2ff4”,GROUP=”plugdev”
SUBSYSTEM==”usb”,ACTION==”add”,ATTRS{idVendor}==”03eb”,ATTRS{idProduct}==”2104”,GROUP=”plugdev”
```

Listing 3. Dostosowanie pliku *makefile* projektu LUFA do prezentowanej płytki

```
// Przed zmianą:
MCU = at90usb1287
BOARD = USBKEY
F_CPU = 8000000
LUFA_PATH = ../..
// Po zmianie:
// Użyty mikrokontroler.
MCU = atmega32u4
// Pliki sterowników będą definiowane przez programistę.
BOARD = USER
// Częstotliwość kwarcu. Należy wpisać zgodnie z użytym kwarcem oraz ustawieniem fusebitu CKDIV8.
F_CPU = 16000000
// Ścieżka do biblioteki.
LUFA_PATH = ../../LUFA-120219
```

W systemie Linux ważną rolę odgrywa posiadanie odpowiednich uprawnień, pozwalających na dostęp do urządzenia USB, w tym przypadku bootloadera prezentowanej płytki, więc pierwsze próby warto przeprowadzić na koncie użytkownika *root*, aby od razu wykluczyć problemy z uprawnieniami. Dodatkowo, na **listingu 2** przedstawiono dwie reguły demona *udev*, dzięki którym pliki urządzeń tworzone dla bootloadera USB lub programatora AVRISP mkII będą należały do grupy *plugdev*, umożliwiając bezproblemowy dostęp wszystkim użytkownikom należącym do tej grupy. W przypadku mojej dystrybucji (Slackware 13.37) okazało się niezbędne odmontowanie systemu plików *usbfs* z katalogu */proc/bus/usb*.

Na zakończenie warto zaznaczyć, że bootloader nie umożliwia zmiany fusebitów, ale pozwala na zmianę lockbitów oraz nie umożliwia wymiany siebie samego. Ograniczenia te nie dotyczą programowania z wykorzystaniem standardowego interfejsu ISP, jednak w tym przypadku jest niezbędny zewnętrzny programator.

Listing 4. Dostosowanie pliku *LEDs.h* do prezentowanej płytki

```
(...)
/** LED mask for the first LED on the board. */
#define LEDS_LED1   (1<<PD6)
/** LED mask for the second LED on the board. */
#define LEDS_LED2   (1<<PD5)
/** LED mask for the third LED on the board. */
#define LEDS_LED3   (1<<PD4)
/** LED mask for the fourth LED on the board. */
#define LEDS_LED4   (1<<PD4)

(...)

static inline void LEDs_Init(void)
{
    DDRD  |=  LEDS_ALL_LEDS;
    PORTD &= ~LEDS_ALL_LEDS;
}

static inline void LEDs_Disable(void)
{
    DDRD  &= ~LEDS_ALL_LEDS;
    PORTD &= ~LEDS_ALL_LEDS;
}

static inline void LEDs_TurnOnLEDs(const uint8_t LEDMask)
{
    PORTD |= LEDMask;
}

static inline void LEDs_TurnOffLEDs(const uint8_t LEDMask)
{
    PORTD &= ~LEDMask;
}

static inline void LEDs_SetAllLEDs(const uint8_t LEDMask)
{
    PORTD = ((PORTD & ~LEDS_ALL_LEDS) | LEDMask);
}

static inline void LEDs_ChangeLEDs(const uint8_t LEDMask, const uint8_t
ActiveMask)
{
    PORTD = ((PORTD & ~LEDMask) | ActiveMask);
}

static inline void LEDs_ToggleLEDs(const uint8_t LEDMask)
{
    PORTD ^= LEDMask;
}

static inline uint8_t LEDs_GetLEDs(void) ATTR_WARN_UNUSED_RESULT;
static inline uint8_t LEDs_GetLEDs(void)
{
    return (PORTD & LEDS_ALL_LEDS);
}

(...)
```

LUFA

Interfejs USB, w przeciwieństwie do np. RS-232, nie należy do najprostszych w oprogramowaniu. W jego wypadku najwygodniej skorzystać z gotowych bibliotek ułatwiających obsługę. Jedną z chyba najbardziej popularnych bibliotek USB dla mikrokontrolerów AVR jest LUFA (*Lightweight USB Framework for AVRs*), której podstawy użycia przedstawię korzystając z omawianej płytki. W przykładach wykorzystano bibliotekę LUFA o oznaczeniu 120219, którą można pobrać spod adresu *http://goo.gl/NrWlmv*. Wszystkie przykłady były kompilowane oraz testowane w systemie operacyjnym Linux (Slackware) przy wykorzystaniu kompilatora AVR-GCC (w wersji *Atmel AVR 8-bit Toolchain 3.4.0 – Linux 32-bit*). Należy zaznaczyć, że LUFA została napisana pod kątem kompilatora GCC i z dużym prawdopodobieństwem nie ma prostej możliwości wykorzystania innego.

Podstawy konfiguracji LUFA

Biblioteka została skonstruowana w ten sposób, że możliwe jest szybkie uruchomienie wybranego przykładu tak na własnej płytce, jak i wielu popularnych płytkach rozwojowych (np. AT90USBKEY, XPLAIN). Spośród pokaźnej ilości przykładowych projektów wiele może zostać wykorzystanych praktycznie bez większych zmian w kodzie. Na potrzeby niniejszego artykułu przyjmijmy, że źródła biblioteki zostały rozpakowane do katalogu *ep-avr-usb* katalogu domowego, zaś przykładowe projekty będą kopiowane do katalogu *projects* znajdującego się w katalogu *ep-avr-usb*. Kopiowanie projektu nie jest konieczne, jednak zapobiega tworzeniu bałaganu w źródłach biblioteki. Układ katalogów będzie następujący:

- ~/ep-avr-usb/LUFA-120219 – oryginalne pliki źródłowe biblioteki,
- ~/ep-avr-usb/projects/przykladowy_projekt_1 – pierwszy projekt,
- ~/ep-avr-usb/projects/przykladowy_projekt_2 – drugi projekt.

Listing 5. Dostosowanie pliku *Buttons.h* do prezentowanej płytki

```
(...)
#define BUTTONS_BUTTON1      (1<<PB6)
#define BUTTONS_BUTTON2      (1<<PB4)
#define BUTTONS_ALL_BUTTONS  ( BUTTONS_BUTTON2 | BUTTONS_BUTTON1 )
(...)
static inline void Buttons_Init(void)
{
    DDRB  &= ~BUTTONS_ALL_BUTTONS;
    PORTB |=  BUTTONS_ALL_BUTTONS;
}

static inline void Buttons_Disable(void)
{
    DDRB  &= ~BUTTONS_ALL_BUTTONS;
    PORTB &= ~BUTTONS_ALL_BUTTONS;
}

static inline uint8_t Buttons_GetStatus(void) ATTR_WARN_UNUSED_RESULT;
static inline uint8_t Buttons_GetStatus(void)
{
    return ( (PINB & BUTTONS_ALL_BUTTONS) ^ BUTTONS_ALL_BUTTONS );
}
(...)
```

Biblioteka zapewnia spójne API oraz spójną strukturę katalogów, dzięki czemu dostosowanie wybranego przykładu na potrzeby prezentowanej płytki sprowadza się do wykonania kilku prostych czynności. Czynności te są takie same dla wszystkich przedstawionych przykładów i polegają na:

- skopiowaniu wybranego przykładu do katalogu *projects*,
- dostosowaniu pliku *makefile* danego przykładu,
- skopiowaniu i uzupełnieniu plików sterowników urządzeń.

Każdy projekt wchodzący w skład biblioteki posiada plik *makefile*, który należy dostosować do posiadanego sprzętu. Na **listingu 3** przedstawiona została część zmian, dostosowujących wybrany projekt do prezentowanej płytki. Celowo napisałem część zmian, gdyż różne przykłady mogą wymagać dodatkowych, drobnych zmian. Warto wiedzieć, że plik *makefile* pozwala również na szybkie zaprogramowanie mikrokontrolera, wystarczy jedynie wydać polecenie *make dfu* lub *make flip* w katalogu z przykładową aplikacją (oczywiście bootloader mikrokontrolera powinien być wtedy uruchomiony).

Niektóre aplikacje LUFA mogą wymagać przyłączenia do mikrokontrolera np. przycisków bądź diod LED. W celu zapewnienia spójności, biblioteka zawiera szablonowe pliki sterowników tych elementów, które należy skopiować do podkatalogu *Board* katalogu danego przykładu, a następnie uzupełnić ich zawartość. Szablony sterowników znajdują się w katalogu: *~/ep-avr-usb/LUFA-120219/LUFA/CodeTemplates/DriverStubs*, zaś po skopiowaniu ich ścieżka powinna wyglądać tak:

- ~/ep-avr-usb/projects/przykladowy_projekt_1/Board/LEDs.h.
- ~/ep-avr-usb/projects/przykladowy_projekt_1/Board/Buttons.h.
- itd.

Na **listingu 4** pokazano fragment pliku *LEDs.h* uzupełniony o definicje wymagane dla prezentowanej płytki. Makrom LEDS_LED1… LEDS_LED4 zostały przypisane numery pinów, do których przyłączone są poszczególne diody. Ponieważ na płytce mamy do dyspozycji trzy diody LED, a plik wymaga podania czterech, programowym diodom 3 i 4 odpowiada jedna fizyczna dioda D4. Aby wszystkie diody działały prawidłowo, powinny być przyłączone do jednego portu mikrokontrolera. Funkcje odpowiedzialne za obsługę diod (LEDs_*) uzupełnione zostały na podstawie pliku *~/ep-avr-usb/LUFA-120219/LUFA/Drivers/Board/AVR8/USBKEY/LEDs.h*. Jeżeli nie potrzebujemy sygnalizacji wykorzystującej LED-y, makrom LEDS_LED1..LEDS_LED4 możemy przypisać wartość 0, a ciała funkcji pozostawić puste (jedynie w funkcji LEDs_GetLEDs() należy zwrócić wartość 0, aby uniknąć ostrzeżeń o braku zwracania wartości w czasie kompilacji).

Na **listingu 5** przedstawiono fragment pliku *Buttons.h*, uzupełniony o definicje wymagane dla prezentowanej płytki. Makrom BUTTONS_BUTTON1 oraz BUTTONS_BUTTON2 przypisane zostały numery pinów, do których przyłączone są odpowiednio przyciski S3 oraz S4. Funkcje odpowiedzialne za obsługę przycisków (Buttons_*) uzupełnione zostały na podstawie pliku *~/ep-avr-usb/LUFA-120219/LUFA/Drivers/Board/AVR8/USBKEY/Buttons.h*. W przypadku dwóch lub więcej przycisków, powinny one być przyłączone do jednego portu mikrokontrolera (stąd brak uwzględnienia przycisku S5, przyłączonego do portu D). Alternatywnie, należałoby napisać własną obsługę, uwzględniającą np. likwidację drgań styków.

W dalszej części artykułu przedstawione zostaną przykłady wykorzystania opisywanej płytki w połączeniu z biblioteką LUFA. W celu kompilacji danego projektu należy wydać polecenie *make*. I tu jedna uwaga - jeśli wcześniej kompilowaliśmy jakiś inny projekt, warto przed poleceniem *make* wydać polecenie *make clean*, które „oczyści" bibliotekę z plików obiektowych poprzedniego projektu. Niestosowanie się do tej reguły może powodować dziwne i ciężkie do usunięcia błędy kompilacji.

Bootloader

Wcześniej wspomniałem, że mikrokontroler ATmega32U4 ma preinstalowany bootloader umożliwiający wgranie własnego programu. Problem w tym, że nie mając dostępu do kodu źródłowego tego bootloadera, jesteśmy skazani na konfigurację producenta (np. dotyczącą sposobu uruchamiania). Wyjściem z tej sytuacji może być wykorzystanie któregoś z bootloaderów dostępnych w bibliotece LUFA.

Do wyboru mamy trzy wersje bootloadera: HID, CDC oraz DFU. Wersja HID korzysta z klasy USB HID, tj. urządzenia interfejsu człowiek – komputer, co oznacza, że system operacyjny posiada wbudowany sterownik do jego obsługi (ten sam, który jest używany np. dla klawiatury czy myszy). W bibliotece, wraz z bootloaderem jest dostarczana aplikacja do jego obsługi (*hid_bootloader_cli*). Wersja CDC korzysta z klasy USB CDC, tj. urządzenia przeznaczonego do komunikacji i jest widziana w systemie jako wirtualny port szeregowy. Podobnie jak w przypadku bootloadera HID, sterowniki dla bootloadera CDC są wbudowane w system operacyjny. Bootloader ten korzysta z protokołu AVR109, zaś programowanie możliwe jest m.in. za pomocą aplikacji *avrdude*. Ostatni bootloader, to jest w wersji DFU, korzysta z klasy USB DFU, przeznaczonej dla urządzeń wspierających aktualizację firmware. Co ważne, jest zgodny z fabrycznym bootloaderem Atmela, w związku z tym programowanie odbywa się w identyczny sposób (opisany w sekcji **Programowanie**), przy wykorzystaniu programów *FLIP* lub *dfu-programmer*. Temu bootloaderowi przyjrzymy się bliżej.

Pracę rozpoczynamy od skopiowania katalogu *~/ep-avr-usb/LUFA-120219/Bootloaders/DFU* do katalogu *~/ep-avr-usb/projects/DFU*. Następnie w katalogu *DFU* tworzymy katalog*Board* i kopiujemy do niego plik *~/ep-avr-usb/LUFA-120219/LUFA/CodeTemplates/DriverStubs/LEDs.h*. Plik *LEDs.h* uzupełniamy zgodnie z list. 4. W przypadku tego przykładu nie jest wymagany plik *Buttons.h*. Plik *makefile* projektu modyfikujemy zgodnie z list. 3, dodatkowo zmieniając definicję FLASH_SIZE_KB = 128 na FLASH_SIZE_KB = 32 (określa ona wielkość pamięci Flash użytego mikro-

Listing 6. Przykład kodu uruchamiającego bootloader
```
// Pin PE2 (S2, HWB) jako wejście, włączenie/wyłączenie rezystora podciągającego nie ma znaczenia
// bo PE2 jest podciągnięty za pomocą R4 na płytce.
DDRE &= ~(1<<PE2);
// Sprawdź stan przycisku i ustaw flagę RunBootloader na true, jeśli przycisk naciśnięty,
// w przeciwnym razie ustaw flagę na false.
RunBootloader = (!(PINE & (1<<PE2)));
// Uruchom aplikację użytkownika, jeśli bootloader nie ma być uruchomiony.
if( !RunBootloader ) AppStartPtr();
```

Listing 7. Dodatkowe zmiany w pliku *makefile* przykładu programatora AVRISP mkII
```
// Przed zmianą:
LUFA_OPTS += -D AUX_LINE_MASK="(1 << 4)"
#LUFA_OPTS += -D NO_VTARGET_DETECT
// Po zmianie:
LUFA_OPTS += -D AUX_LINE_MASK="(1 << 7)"
LUFA_OPTS += -D NO_VTARGET_DETECT
```

Listing 8. Zmiana w pliku *V2Protocol.c* przykładu programatora mkII
```
// Linijka 56 przed zmianą:
#if defined(ADC)
// Linijka 56 po zmianie:
#if (defined(ADC) && !defined(NO_VTARGET_DETECT))
```

kontrolera). W pliku *BootloaderDFU.h* makru SECURE_MODE można przypisać wartość *true*, co spowoduje, że jedyną możliwą operacją na pamięci po uruchomieniu bootloadera będzie jej skasowanie (w szczególności, przed skasowaniem, niemożliwe będzie jej odczytanie). Zabieg ten (wraz z odpowiednim ustawieniem lockbitów) ma na celu uniemożliwienie skopiowania zawartości pamięci osobom postronnym.

Ostatnia modyfikacja dotyczy sposobu uruchamiania bootloadera (opisany zostanie sposób wykorzystujący przycisk, ale może to być dowolny inny, wymyślony przez programistę). Założenie jest takie, że bootloader będzie uruchamiany w taki sam sposób jak fabryczny bootloader, czyli poprzez odpowiednią sekwencję naciskania przycisków S1 (RESET) oraz S2 (HWB). Zmiany wprowadzimy w głównym pliku aplikacji, tj. *BootloaderDFU.c*. To, czy bootloader zostanie uruchomiony, zależy od wartości zmiennej *RunBootloader*, której początkowa wartość wynosi *true*. Na **listingu 6** przedstawiony został kod odpowiedzialny za sprawdzenie stanu przycisku S2 oraz odpowiednie ustawienie zmiennej *RunBootloader*. Kod ten należy umieścić w pliku *BootloaderDFU.c* na początku funkcji main(), przed wywołaniem funkcji SetupHardware(). Po skompilowaniu projektu, płytkę należy zaprogramować plikiem *BootloaderDFU.hex*, zaś sam bootloader uruchomić zgodnie z opisem w sekcji **Programowanie**.

W celu wymiany bootloadera niezbędny jest osobny programator, gdyż bootloadery najczęściej nie wspierają wymiany siebie samego. Ja użyłem drugiej płytki ATmega32U4-DIPMOD z aplikacją AVRISP mkII, której uruchomienie i wykorzystanie zostało opisane w następnej sekcji.

W czasie aktualizacji bootloadera warto zwrócić uwagę na odpowiednie ustawienie fusebitów mikrokontrolera, w szczególności HWBE, BOOTRST oraz CKDIV8. W **tabeli 1** przedstawiony został sposób uruchamiania mikrokontrolera w zależności od ustawienia bitów HWBE i BOOTRST oraz stanu panującego na pinie HWB (PE2). Uważam, że bootloader powinien być uruchamiany jako pierwszy (BOOTRST ustawiony) i decydować czy skoczyć do programu głównego, czy przejść do trybu programowania. Tym sposobem, programowanie będzie możliwe nawet w przypadku uszkodzenia programu głównego. Bit CKDIV8 (domyślnie włączony) odpowiedzialny jest za wewnętrzny podział częstotliwości taktującej przez 8, tzn. jeśli zastosujemy kwarc 8 MHz to rdzeń (i nie tylko) będzie taktowany częstotliwością 1 MHz. Należy mieć to na uwadze przy ustalaniu wartości F_CPU w pliku makefile (poza fusebitem CKDIV8, istnieje również możliwość programowego wyłączenia tego podziału za pomocą funkcji *clock_prescale_set()* – funkcjonalność ta jest często wykorzystywana w bibliotece LUFA, o czym też należy pamiętać). Bit ten nie ma wpływu na działanie interfejsu USB, gdyż ten korzysta z PLL i jest taktowany osobno (jakkolwiek z tego samego kwarcu).

Tabela 1. Wpływ ustawienia fusebitów HWBE i BOOTRST oraz stanu na pinie HWB (PE2) na proces uruchamiania mikrokontrolera

BOOTRST	HWBE	HWB (PE2)	Uruchamiany program
1	X	X	Bootloader
0	0	X	Program główny
0	1	0	Bootloader
0	1	1	Program główny

W przypadku fusebitów: 1 – ustawiony, 0 – nieustawiony, X – dowolny
W przypadku pinu PE2: 1 – poziom wysoki, 0 – poziom niski, X – dowolny

Programator AVRISP mkII

Można się kłócić, czy warto robić własny programator, czy lepiej kupić gotowy. Zwolenników pierwszej opcji zachęcam do dalszej lektury, zaś zwolenników drugiej... również, gdyż czasem może się zdarzyć, że uszkodzimy jedyny programator, jakim dysponujemy, a wtedy będziemy w stanie szybko zrobić coś swojego.

Biblioteka LUFA zawiera projekt programatora AVRISP mkII, będącego w znacznym stopniu kompatybilnym, jeśli chodzi o firmware, bo sprzęt jest o wiele uboższy, z programatorem Atmela o tej samej nazwie. Programator obsługuje trzy interfejsy programowania, tj. ISP (np. ATmega), PDI (np. ATxmega) oraz TPI (np. ATtiny o małej liczbie wyprowadzeń) oraz może być obsługiwany z poziomu *AVRStudio* lub *avrdude*. W przeciwieństwie do programatora Atmela, programator z biblioteki LUFA wymaga trzech osobnych wtyków do każdego z wymienionych interfejsów programowania. Po szczegółowe informacje odsyłam do dokumentacji LUFA, zaś poniżej opiszę jak dostosować programator do prezentowanej płytki i wykonać przykładowe połączenia interfejsu ISP.

Pliki źródłowe programatora znajdują się w katalogu *~/ep-avr-usb/LUFA-120219/Projects/AVRISP-MKII* i należy je skopiować do katalogu *~/ep-avr-usb/projects/AVRISP-MKII*. Następnie w katalogu *AVRISP-MKII* tworzymy katalog *Board* i kopiujemy do niego plik *~/ep-avr-usb/LUFA-120219/LUFA/CodeTemplates/DriverStubs/LEDs.h*, który należy uzupełnić zgodnie z listingiem 4. W przypadku tego przykładu nie jest wymagany plik *Buttons.h*. Plik *makefile* projektu modyfikujemy zgodnie z listingiem 3 oraz **listingiem 7**. Znaczenie wybranych opcji konfiguracyjnych jest następujące:

- AUX_LINE_* (PORT, PIN, DDR, MASK) – określenie pinu odpowiedzialnego za obsługę linii RESET programowanego mikrokontrolera. Pin ten warto wybrać na końcu, aby nie kolidował z pozostałymi pinami interfejsu programowania (nie może to być również pin SS modułu SPI mikrokontrolera)
- NO_VTARGET_DETECT – wyłączenie pomiaru napięcia w układzie programowanym. Programator posiada opcję pomiaru tego napięcia, z której

Listing 9. Definicje obsługi przetwornika A/C dla przykładu joysticka
```
#ifndef JOYSTICK_H
#define JOYSTICK_H
#include <LUFA/Drivers/Peripheral/ADC.h>

// Ilość próbek na kanał.
#define JOY_ADC_SAMPLES_NUM  8
// Kanał ADC dla osi Y.
#define JOY_ADC_CHANNEL_Y    ADC_CHANNEL0
// Kanał ADC dla osi X.
#define JOY_ADC_CHANNEL_X    ADC_CHANNEL1
#endif
```

Listing 10. Definicja zmiennych zawierających aktualne położenie joysticka

```
static int8_t y_ax;
static int8_t x_ax;
```

nie będziemy korzystać w prezentowanym przykładzie

- LIBUSB_DRIVER_COMPAT – włączenie trybu kompatybilności z *libusb*, z którego korzysta m.in. *avrdude*. Niestety, jednocześnie tracona jest kompatybilność z *AVRStudio*. Jakkolwiek, nie stwierdziłem błędnego działania *avrdude* (wersja 5.11.1, Linux) przy braku ustawienia tej opcji

Ze względu na błąd w pliku *Lib/V2Protocol.c*, niezbędne jest jego poprawienie zgodnie z **listingiem 8** (dotyczy wersji LUFA-120219 i prawdopodobnie wcześniejszych).

Zwracam uwagę na fakt, że interfejs programowania ISP korzysta z modułu SPI mikrokontrolera, zaś interfejsy PDI oraz TPI korzystają z USART-u mikrokontrolera, co wiąże się z koniecznością wykonania połączeń w odpowiedni sposób. Na **rysunku 5** pokazano schemat połączeń w przypadku programowania mikrokontrolerów z serii ATmega za pomocą interfejsu SPI. Należy pamiętać o zachowaniu kompatybilności poziomów logicznych między płytką, a układem programowanym (tj. oba układy powinny być zasilane napięciem 5 V lub 3,3 V). Bardziej rozbudowane wersje interfejsu programującego, zawierające między innymi konwerter poziomów oraz interfejsy PDI oraz TPI, można znaleźć w Internecie pod hasłem „mkII clone" (na *Elportalu* można znaleźć projekt o nazwie „Programator USBTiny MKII PL". Na pinie OC1A (PB5) mikrokontrolera jest generowany sygnał o częstotliwości 4 MHz, który można wykorzystać w celu odblokowania mikrokontrolera AVR zablokowanego nieprawidłowym ustawieniem fusebitów. Sygnał ten należy doprowadzić do pinu XTAL1 odblokowywanego mikrokontrolera, zaś częstotliwość interfejsu ISP powinna być ustawiona na 125 kHz.

Po skompilowaniu przykładu, płytkę należy zaprogramować plikiem *AVRISP-MKII.hex* (w tym celu można skorzystać z bootloadera i polecenia *make dfu*). Po przyłączeniu do komputera, płytka będzie widziana jako programator AVRISP mkII. W systemie Windows należy zainstalować sterownik dostępny w *AVRStudio*, w przypadku Linuksa, programator powinien być od razu gotowy do użycia.

Joystick USB

Joystick to przykład urządzenia wykorzystującego klasę HID USB, czyli urządzenia stanowiącego interfejs między człowiekiem a komputerem. Oryginalny przykład joysticka z biblioteki LUFA w celu aktualizacji współrzędnych XY wykorzystuje joystick oparty o styki, co uniemożliwia ich płynną zmianę (możliwe są tylko skrajne położenia). Poniżej pokażę jak zmodyfikować przykład, aby korzystał z przetwornika A/C wbudowanego w mikrokontroler, zaś zmiana współrzędnych odbywała się za pomocą dwóch potencjometrów o rezystancji 10 kΩ.

Podobnie jak poprzednio, pracę należy rozpocząć od skopiowania przykładu, który znajduje się w katalogu *~/ep-avr-usb/LUFA-120219/Demos/Device/ClassDriver/Joystick*, następnie należy utworzyć katalog sterowników *Board* i skopiować do niego pliki *LEDs.h* oraz tym razem *Buttons.h*, oba uzupełniając zgodnie z list. 4 oraz 5. Dodatkowo należy utworzyć plik *Joystick.h* o treści jak na **listingu 9**. Gdybyśmy chcieli uruchomić oryginalny przykład, plik ten pochodziłby z katalogu *DriverStubs* i odpowiadałby za joystick oparty o styki. W naszym przypadku, w pliku tym określiliśmy kanały przetwornika A/C, do których będą przyłączone potencjometry osi X i Y oraz ile próbek z danego kanału należy pobrać przed właściwą aktualizacją współrzędnych (zabieg ten ma na celu eliminację zakłóceń mogących pojawiać się na wejściu przetwornika A/C). Plik *makefile* projektu modyfikujemy zgodnie z listingiem 3, poza tym nie wymaga on dodatkowych zmian.

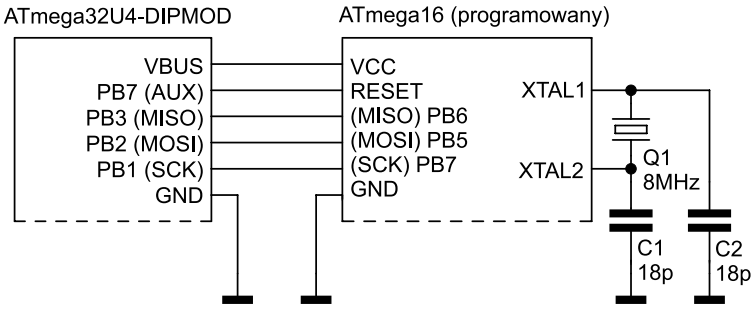
W prezentowanym przykładzie zadowolimy się 8-bitową rozdzielczością przetwornika, która jest również wystarczająca dla wielu zastosowań praktycznych. O minimalnej oraz maksymalnej wartości wychylenia joysticka należy poinformować komputer, co zrobi za nas biblioteka. Wartości te podajemy w pliku *Descriptors.c* w linii 56. Omawiany przykład operuje na współrzędnych X i Y, które są typu int8_t (istnieje możliwość zmiany tego typu), czyli o zakresie wartości –128 – +127, zaś zakres wartości przetwornika A/C to 0…255 (*uint8_t*), więc w programie niezbędne będzie dokonanie prostego przekształcenia zakresu wartości typu *uint8_t* na *int8_t*. Linia 56 będzie wyglądała następująco: HID_DESCRIPTOR_JOYSTICK(2, –128, 127, –1, 1, 2).

Teraz pozostaje już tylko dodać obsługę i odczyt przetwornika A/C w głównym pliku aplikacji (*Joystick.c*). Po pierwsze, dyrektywą *#include <LUFA/Drivers/Peripheral/ADC.h>* dołączamy funkcje obsługi przetwornika, następnie w miejscu przed wywołaniem funkcji main() dodajemy dwie zmienne globalne, które będą zawierały aktualne położenie potencjometrów joysticka (**listing 10**). W funkcji SetupHardware() zamiast funkcji Joystick_Init() należy wstawić wywołanie funkcji ADC_Init(ADC_SINGLE_CONVERSION | ADC_PRESCALE_128), które skonfiguruje przetwornik do pracy w trybie pojedynczej konwersji, przy częstotliwości taktowania równej F_CPU / 128.

W funkcji main() dodajemy właściwy odczyt wartości z przetwornika. Biblioteka udostępnia kilka różnych funkcji, przy czym prezentowany przykład korzysta z następujących:

- ADC_SetupChannel() – wybór kanału pomiarowego,
- ADC_GetChannelReading() – wykonanie i odczyt pomiaru z oczekiwaniem na zakończenie konwersji.

Dokładny opis parametrów wywołania funkcji przetwornika A/C można znaleźć w dokumentacji biblioteki, zaś opis parametrów użytych w przykładzie znajduje się w komentarzach. Na **listingu 11** pokazano zmodyfikowaną wersję pętli głównej programu Joysticka, uzupełniona o odczyt przetwornika. Jak wcześniej wspomniałem, w celu likwidacji zakłóceń mogących pojawiać się na wejściu przetwornika, wartość napięcia z potencjometru joysticka jest mierzona kilkukrotnie w pętli *for*, kolejne odczyty są sumowane, a następnie uśredniane poprzez dzielenie. Na sam koniec, od średniej wartości odczytu odejmowana jest wartość 128, dzięki czemu liczba bez znak o zakresie wartości 0 – 255 (uint8_t) zostaje zamieniona na liczbę ze znakiem o zakresie wartości –128 – +127 (int8_t). Zwracam uwagę na fakt, że możliwe jest bezpośrednie wykorzystanie 10-bitowego wyniku pomiaru przetwornika A/C z zakresu 0 – 1023 (bez znaku), bez konieczności jego zamiany na liczbę ze znakiem. W przedstawio-



Rysunek 5. Schemat połączeń interfejsu ISP

Listing 11. Zmodyfikowana pętla główna przykładu Joysticka

```
for (;;)
{
    // Suma kolejnych próbek poszczególnych kanałów ADC.
    int16_t y_ax_sum = 0;
    int16_t x_ax_sum = 0;

    // Pętla pomiarowa. Napięcie poszczególnych kanałów zostanie zmierzone
    // tyle razy, ile określono w stałej JOY_ADC_SAMPLES_NUM.
    for( uint8_t i = 0; i < JOY_ADC_SAMPLES_NUM; i++ )
    {
        // Wybór kanału zapisanego w stałej JOY_ADC_CHANNEL_Y.
        ADC_SetupChannel( JOY_ADC_CHANNEL_Y );
        // Pomiar napięcia wybranego kanału ADC. Parametr wywołania oznacza, że:
        // a) napięcie odniesienia przetwornika pobierane jest z pinu AVCC (ADC_REFERENCE_AVCC)
        // b) wynik pomiaru jest wyrównany do lewej strony (ADC_LEFT_ADJUSTED)
        // c) wybrany zostaje kanał określony w stałej JOY_ADC_CHANNEL_Y
        // Funkcja ADC_GetChannelReading() czeka w pętli na zakończenie konwersji. Wynik, przed dodaniem
        // do zmiennej sumy próbek jest przesuwany o 8 bitów w prawo, ponieważ parametr ADC_LEFT_ADJUSTED
        // oznacza, że 8 najbardziej znaczących bitów 10 bitowego wyniku pomiaru jest umieszczanych w
        // bardziej znaczącym bajcie 16 bitowego wyniku zwracanego przez funkcję. Dzięki tym operacjom,
        // wynik przetwarzania jest 8 bitowy, bazując na 10 bitowym przetworniku.
        y_ax_sum += ADC_GetChannelReading( ADC_REFERENCE_AVCC | ADC_LEFT_ADJUSTED | JOY_ADC_CHANNEL_Y ) >> 8;

        ADC_SetupChannel( JOY_ADC_CHANNEL_X );
        x_ax_sum += ADC_GetChannelReading( ADC_REFERENCE_AVCC | ADC_LEFT_ADJUSTED | JOY_ADC_CHANNEL_X ) >> 8;
    }

    // Obliczenie pozycji potencjometrów osi X-Y. Po pierwsze, wynik pomiaru (suma próbek)
    // poszczególnych kanałów jest uśredniany (dzielony przez JOY_ADC_SAMPLES_NUM). Następnie
    // wartość bez znaku z zakresu 0-255 jest zamieniana na wartość ze znakiem z
    // zakresu -128 - +127, poprzez odjęcie wartości 128 (255 - 128 = 127, 0 - 128 = -128).
    y_ax = (y_ax_sum / JOY_ADC_SAMPLES_NUM) - 128;
    x_ax = (x_ax_sum / JOY_ADC_SAMPLES_NUM) - 128;


    HID_Device_USBTask(&Joystick_HID_Interface);
    USB_USBTask();
}
```

nym przykładzie zrezygnowałem jednak z takiego rozwiązania, gdyż zmienne wyniku pomiaru są współdzielone między pętlą główną a przerwaniem, a jak wiadomo, w przypadku 8-bitowego mikrokontrolera oraz zmiennych większych niż jeden bajt, sytuacja taka może prowadzić do złej interpretacji wartości zmiennej, jeśli się odpowiednio nie zabezpieczymy. Zmienne położenia (*x_ax* oraz *y_ax*) są aktualizowane w pętli głównej wartością z przetwornika A/C, zaś ich odczyt następuje w funkcji *CALLBACK_HID_Device_CreateHIDReport()* (**listing 12**), która może być wywoływana w kontekście przerwania, a jej zadaniem jest przygotowanie informacji o stanie joysticka w celu wysłania do komputera. Ciało funkcji z list. 12 różni się od wersji oryginalnej tym, że kod odpowiedzialny za odczyt stanu joysticka bazującego na stykach zastąpiony został kodem aktualizującym współrzędne na podstawie położenia potencjometrów (tj. wartości zmiennych globalnych *y_ax* oraz *x_ax*). Nieznacznie zmieniony został również kod odczytu stanu drugiego przycisku (w celu odzwierciedlenia połączeń na płytce).

Schemat przyłączenia potencjometrów osi X i Y do płytki przedstawiony został na **rysunku 6**. **Uwaga!** W tym przypadku, zworka J1 ustalająca napięcie pracy mikrokontrolera musi być ustawiona w pozycji 5 V (VBUS), gdyż przy połączeniu zgodnie z rysunkiem 6, takie jest napięcie odniesienia potencjometrów i takie też powinno być napięcie odniesienia przetwornika A/C.

Po wgrania programu (*Joystick.hex*), płytka będzie widziana w systemie operacyjnym jako dwuosiowy joystick, mający dwa przyciski.

Listing 12. Ciało funkcji odpowiedzialnej za tworzenie raportu HID joysticka

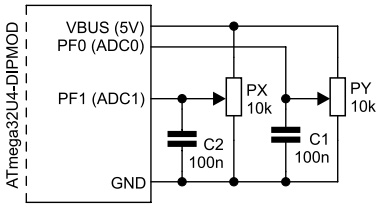
```
USB_JoystickReport_Data_t* JoystickReport = (USB_JoystickReport_Data_t*)ReportData;
uint8_t ButtonStatus_LCL = Buttons_GetStatus();

JoystickReport->Y = y_ax;
JoystickReport->X = x_ax;
if (ButtonStatus_LCL & BUTTONS_BUTTON2)
  JoystickReport->Button |= (1 << 1);
if (ButtonStatus_LCL & BUTTONS_BUTTON1)
  JoystickReport->Button |= (1 << 0);
*ReportSize = sizeof(USB_JoystickReport_Data_t);
return false;
```

Podsumowanie

Dzięki pokaźnej liczbie przykładowych aplikacji, biblioteka LUFA pozwala na szybkie i łatwe rozpoczęcie przygody z mikrokontrolerami AVR z interfejsem USB. W połączeniu z prezentowaną płytką, ich uruchomienie nie zajmie więcej niż kilkanaście – kilkadziesiąt minut.

W artykule nie poruszyłem problemu tworzenia własnej aplikacji korzystającej z biblioteki LUFA od podstaw. W tym celu niezbędna jest szersza wiedza, tak o standardzie USB jak i strukturze samej biblioteki. Jakkolwiek, przedstawione przykłady dają ogólny zarys zagadnienia i myślę, że przy odrobinie wysiłku czytelnicy będą w stanie sami osiągnąć zamierzone cele, np. przerobić aplikację demonstracyjną wirtualnego portu szeregowego (*~/ep-avr-usb/LUFA-120219/Demos/Device/ClassDriver/VirtualSerial*) na kartę przekaźników sterowaną z terminala. A może to być dopiero początek fascynującej wyprawy w świat interfejsu USB.

Na zakończenie jeszcze raz zachęcam do przejrzenie przykładów oraz dokumentacji projektu LUFA. W większości przypadków do ich uruchomienia wymagane będą jedynie modyfikacje przedstawione w artykule, ewentualnie garść dodatkowych elementów umieszczonych na płytce stykowej.



Rysunek 6. Sposób dołączenia potencjometrów w aplikacji joysticka

Zmodyfikowane wersje źródeł prezentowanych przykładów można znaleźć w materiałach dodatkowych, dołączonych do artykułu.

W czasie przekazywanie tego materiału do redakcji, dostępna była już nowsza wersja projektu LUFA. Jednak w trakcie stawiania pierwszych kroków, zalecam korzystanie z wersji opisanej w artykule, w celu uniknięcia przykrych niespodzianek spowodowanych wprowadzonymi zmianami.

Andrzej Telszewski
atelszewski@gmail.com

Bibliografia:

1. *AN146 – USB Hardware Design Guidelines for FTDI ICs, FTDI*
2. *SPRAAR7 – USB 2.0 Board Design and Layout Guidelines, Texas Instruments*