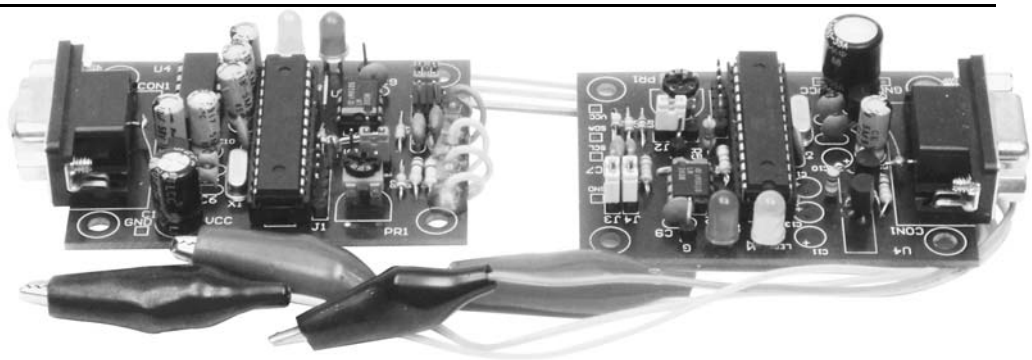
kit 2899 AVT



Analizator I²C

Wielokrotnie podczas budowania różnorodnych urządzeń elektronicznych stosuje się układy scalone korzystające z magistrali I²C. Są to na przykład zegary RTC, przetworniki ADC i DAC, cyfrowe termometry, akcelerometry, przeróżne procesory dźwięku i obrazu. Często mamy problemy z komunikacją poprzez I²C, w skrócie I². Pół biedy, gdy stosujemy od lat znane i przetestowane mikrokontrolery i układy I². Wtedy problemy ograniczają się do niedziałania transmisji z powodu zwarć/przerw na płytce, zbyt dużej częstotliwości taktowania szyny, nieodpowiedniego adresu czy nieodpowiadania układu (brak potwierdzania danych sygnałem ACK). Jednak naprawdę poważnych problemów przysparzają urządzenia budowane od podstaw, na nowo poznanych podzespołach. O tym, jakie cuda potrafi odebrać procesor/układ peryferyjny, mimo że nigdy nie zostały wysłane przez magistralę I²C, przekonałem się sam już niejeden raz. I co ciekawe, problemy najczęściej były powodowane przez źle skompilowany program (brak niektórych rejestrów w bibliotekach kompilatora), dziwaczne efekty kompilacji (niekiedy poprawne, ale naprawdę dziwaczne – np. generowanie jednego taktu zegara przed sygnałem I2CStart w BASCOM-ie). Takie problemy w żaden sposób nie dadzą się wykryć w jakiś programowy sposób (poprzez umieszczenie punktów kontrolnych w uruchamianym programie). Niezbędny jest zewnętrzny analizator I², którego opis przedstawiam w artykule. O ile podsłuchanie na przykład sygnałów RS232 jest wręcz banalne, to nie można tego powiedzieć o I². Dlatego przedstawiony projekt aż prosi się o zastosowanie mikrokontrolera.

Układ jest dość prosty, a dzięki temu to i bardzo tani. Zbudowany jest z popularnych podzespołów i możliwy do wykonania także przez początkujących elektroników. Odbiór jest realizowany na drodze programowej, a wyniki są przesyłane do komputera poprzez interfejs RS232. Układ nie jest tylko prostym 2-kanałowym analizatorem stanów logicznych. Urządzenie samo wykrywa impulsy Start, Stop, przesyłane dane oraz sygnały potwierdzenia Ack/Nack.

Układ potrafi analizować szyny o prędkościach od 0 do 600kHz (900kHz) o poziomach napięć od kilkudziesięciu mV do 30V.

Program zawarty w mikrokontrolerze jest dość krótki, co nie znaczy, że był łatwy do napisania. Łamie on chyba większość zasad wyznawanych przez prawie wszystkich programistów (ale nie przeze mnie).

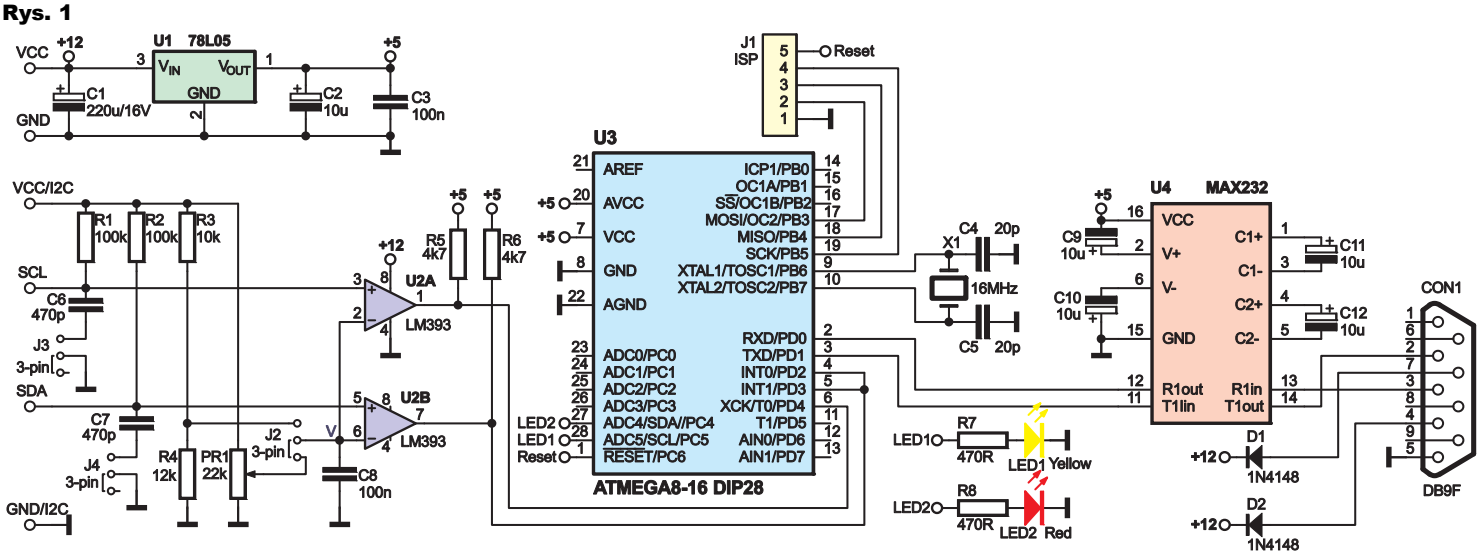
Niezachowywanie stanu rejestrów podczas wchodzenia do obsługi przerwania (absolutnie żadnych – nawet rejestru statusu SREG!), brak instrukcji powrotu z przerwania, wyskakiwanie ze środka przerwania za pomocą instrukcji skoku czy fragmenty kodu, które absolutnie nigdy nie zostaną wykonane, to standard. To, że w ten sposób napisany program może działać, a przy tym zapewnić niespotykaną stabilność i szybkość działania, każdy jednak będzie musiał sprawdzić sam – bo zapewnie mało kto uwierzy mi na słowo.

Opis układu

Część sprzętowa

Schemat układu jest przedstawiony na **rysunku 1**. Napięcie zasilające o wartości 8...15V jest podawane na punkty VCC i GND. Jest ono filtrowane przez kondensator C1 i trafia do stabilizatora U1. Napięcie z wyjścia stabilizatora zasila mikrokontroler oraz układ MAX232. Z kolei niestabilizowane napięcie z przed stabilizatora zasila kompaktory U2A i U2B. Układ można zasilić o wiele większym napięciem – do 30V. Wtedy jednak trzeba zastosować kondensator C1 o większym napięciu pracy.

Rys. 1



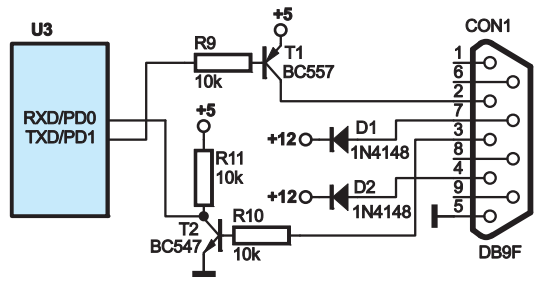
Punkty VCC/ I2C, SCL, SDA i GND/ I2C podłącza się do badanej szyny I^2. Uwaga! Napięcie podawane na punkt VCC/I2C nie powinno być większe niż napięcie zasilania (podawane na punkt VCC).

Napięcie pobierane z punku VCC/ I2C nie służy do zasilania analizatora. Służy ono do generowania napięcia referencyjnego, odpowiedzialnego za rozstrzyganie, czy sygnały na liniach SCL i SDA są logiczną jedynką, czy logicznym zerem. Dzięki temu sygnały na liniach SCL/SDA nie muszą mieć poziomów 5V (takim napięciem jest zasilany mikrokontroler w analizatorze).

Domyślnie napięcie jest dzielone mniej więcej na pół i trafia na odwracające wejścia kompaktorów. Z kolei na wejścia nieodwracające są kierowane sygnały SCL i SDA. Tak więc sygnały z wyjść kompaktów odzwierciedlają sygnały na ich wejściach ($0 \rightarrow 0$, $1 \rightarrow 1$), tyle że poziom napięć jest już równy 5V. Zapewniają to rezystory podciągające R5 i R6. Mała wartość tych rezystorów pozwala na konwertowanie nawet bardzo szybkich sygnałów. Rezystory R1 i R2 wstępnie podciągają wejścia kompaktorów do plusa zasilania badanego urządzenia. Nie są absolutnie konieczne, ale dzięki nim w momencie, gdy któraś z końcówek SCL/SDA nie jest podłączona, układ nie odbiera przydźwięku sieci i innych zakłóceń.

Jak już wcześniej napisałem, napięcie z punktu VCC/I2C jest dzielone MNIEJ WIĘCEJ na pół. Mniej więcej, bo logiczna jedynka w magistrali I^2 zwykle jest bardzo zbliżona do napięcia zasilania (dzięki rezystorom podciągającym), z kolei logiczne 0 już nie bardzo odzwierciedla poziom masy. Powodem jest napięcie nasycenia tranzystorów (w przypadku tranzystorów bipolarnych) lub rezystancja $R_{DS(on)}$ (w przypadku FET). Szczególnie daje się to odczuć przy małej wartości rezystorów podciągających, gdy tranzystory nie są w stanie dostatecznie zewrzeć szyn sygnałowych do masy (nie chodzi o rezystory podciągające R1 i R2, ale o rezystory znajdujące się w badanym urządzeniu – zwykle mają one wartość 220Ω…10kΩ).

Dlatego lepiej, aby napicie referencyjne było trochę wyższe od połowy napięcia zasilania. Stąd rezystory R3 i R4 o nierównych wartościach. Lepiej, by było wyższe, ale niektóre układy scalone mogą mieć napięcie progowe inne niż połowa VCC, stąd też zwora J2 i potencjometr PR1. Pozwala on dowolnie ustawić napięcie referencyjne. Można je zmierzyć w punkcie V, wyprowadzonym na stronę elementów płytki drukowanej w formie jednego goldpina (wyprowadzona jest także masa w postaci punktu G). Dowolnie ustawiane napięcie referencyjne pozwoli sprawdzić, czy faktycznie za nieprawidłową komunikację może odpowiadać zbyt wysokie napięcie w stanie niskim. Pozwoli to także przetestować szyny I^2, w których stosuje się konwertery poziomów napięć (dwukierunkowe oczywiście) lub bardzo długie przewody połączeniowe. Często po jednej stronie (cienkiego) kabla napięcia są prawidłowe, ale po drugiej już nie. Dokładniej mówiąc, ustawiane napięcie referencyjne pozwoli sprawdzić, przy jakim napięciu progowym sygnały odbierane przez analizator zaczynają być błędne. Wtedy możemy porównać wyznaczone napięcie progowe z danymi katalogowymi badanych układów scalonych i ocenić, dlaczego układ nie pracuje, lub gdy pracuje – jak duży jest margines bezpieczeństwa.



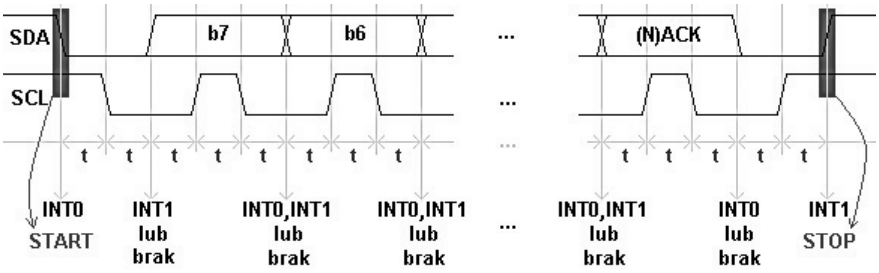
Rys. 2

Kondensator C9 filtruje napięcie referencyjne z wszelkich zakłóceń, które będą się generowały w przewodzie połączeniowym I2C/VCC. Szczególnie w długim przewodzie połączeniowym. Zakłócenia będą się generowały także w przewodach doprowadzających sygnały SCL i SDA, ale filtracja napięcia referencyjnego jest niejako ważniejsza.

Jednak nasz układ, jako przystosowany do pracy także z bardzo szybkimi sygnałami, odbiera wszelkie bardzo krótkie impulsy z szyn SCL, SDA. Podczas testów prototypu nie stwierdziłem żadnych problemów w działaniu urządzenia, jednak w bardzo zakłóconych środowiskach (ale zakłócenia generuje także sam badany układ), szczególnie przy analizowaniu sygnałów o małych prędkościach, konieczne może się okazać dołączenie kondensatorów C6, C7. Realizuje się to, przekładając zworki J3–J4 do pozycji „2". Często po dołączeniu tych kondensatorów analizator zacznie odbierać prawidłowe dane. A nawet badane urządzenie może zacząć działać! Oznacza to tyle, że badane urządzenie odbiera wszelkie „śmieci" z szyn SCL i SDA. Najczęściej jest tak, gdy rezystory podciągające szyny SCL i SDA mają dużą wartość, powiedzmy powyżej 10kΩ, oraz układ generuje ogromną ilość zakłóceń. Może tak być, gdy układ zawiera jakąś przetwornicę impulsową elementy przełączające duże prądy (np. sterowanie PWM silnika dużej mocy).

Sygnał SDA (już o poziomie 5V) trafia jednocześnie na piny PD2 i PD3 (INT0 i INT1). Dlaczego tak? Bo dzięki temu mamy możliwość osobnego i zarazem bardzo prostego wykrywania zarówno zbocza narastającego, jak i opadającego. Natomiast sygnał SCL trafia na PD4. Transmisja ze specjalnym programem napisanym w Delphi odbywa się poprzez port RS232. Konwersja poziomów napięć jest realizowana przez dobrze znany układ MAX232. W tym miejscu muszę się przyznać, że jest to moje pierwsze urządzenie wykorzystujące MAX232. Zbudowałem już kilkadziesiąt urządzeń korzystających z transmisji RS232 i zawsze wykorzystywałem prosty konwerter na dwóch tranzystorach. Nigdy nie miałem żadnych problemów, mimo że układy współpracowały z przeróżnymi komputerami. Płytka drukowana jest przystosowana do zastosowania takiego prostego konwertera. Układ połączeń będzie wtedy wyglądał tak jak na **rysunku 2**.

Układ z MAX232 i ATmega8 pobiera od 32 do 45mA. Z T1, T2, R3 i R4 = 1,5kΩ i ATmega88 od 18 do 22mA (z rezonatorem 16MHz). Cały analizator pobiera więc niewiele prądu, jednak w wersji podstawowej urządzenia zrezygnowałem z zasilania układu za pomocą sygnałów DTR i RTC ze złącza CON1. Powód jest prosty: wiele nowoczesnych komputerów nie ma złączy RS232. Istnieją konwertery USB-RS232 do kupienia za grosze, ale one często nie posiadają sygnałów DTR i RTC. Dlatego też zasilanie jest realizowane poprzez punkty VCC i GND. Jednak na płytce drukowanej jest miejsce na wlutowanie dwóch diod (D1 i D2). Program sterujący (ten na komputer) ustawia piny DTR i RTC w stan wysoki (tzn. ustawia na nich +12V). Możliwe więc będzie zasilanie układu z portu COM. Jednak, aby takie zasi-



Rys. 4

lanie było możliwe, układ musi pobierać mało prądu (trzeba zastosować ATmega88, rezonator maksymalnie 16MHz, zamiast MAX232 wykorzystać prosty konwerter na T1 i T2 oraz zwiększyć wartość rezystorów R7 i R8 do co najmniej 1,5kΩ). Poza tym piny portu COM muszą mieć odpowiednio dużą wydajność prądową. W komputerach stacjonarnych wydajność będzie zazwyczaj zadowalająca (15mA/Pin), ale w komputerach przenośnych najczęściej nie. Układ można także zasilać z testowanego urządzenia. Jest to w dodatku chyba najlepsze rozwiązanie. Masę badanego urządzenia łączymy bezpośrednio z masą analizatora. Natomiast punkt VCC najlepiej podłączyć do napięcia wyższego niż napięcie zasilające układy z interfejsem I^2. Połączenie może wyglądać na przykład tak jak na **rysunku 3**. Jeżeli napięcie przed stabilizatorem napięcia w badanym urządzeniu ma wartość mniejszą niż 7…8V, to niestety musimy zastosować osobny zasilacz do zasilania analizatora I^2.

Przy rezonatorze 16MHz układ jest w stanie analizować szyny I^2 o częstotliwościach sygnału zegarowego do 600kHz. Do 900kHz przy 24MHz, ale układ z takim rezonatorem będzie działał jedynie, gdy zastosujemy mikrokontroler ATmega88 (lub ATTINY2313). Dotyczy to wypełnienia sygnału zegarowego 33,3% (33,3%, bo przyjąłem, że czas trwania stanu niskiego na linii SCL będzie dwa razy dłuższy niż trwania stanu wysokiego – a to dlatego, że podczas trwania stanu niskiego może zmieniać się stan linii SDA. Na jeden cykl przypadają więc nie dwie, a trzy możliwe zmiany stanów, więc przyjąłem 33,3%. Widać to na **rysunku 4**). Dla innych współczynników wypełniania maksymalna częstotliwość analizowanych sygnałów może być mniejsza lub niekiedy nawet większa od wymienionych. Jednak nie ma to większego znaczenia, bo układ i tak spokojnie poradzi sobie z analizowaniem standardowych szyn o częstotliwości 100kHz, a nawet niezbyt popularnych szybkich szyn 400kHz. Przebiegi, których użyłem do określania maksymalnej prędkości, przedstawia **rysunek 4**. Wyjaśnia on także kilka innych ważnych rzeczy.

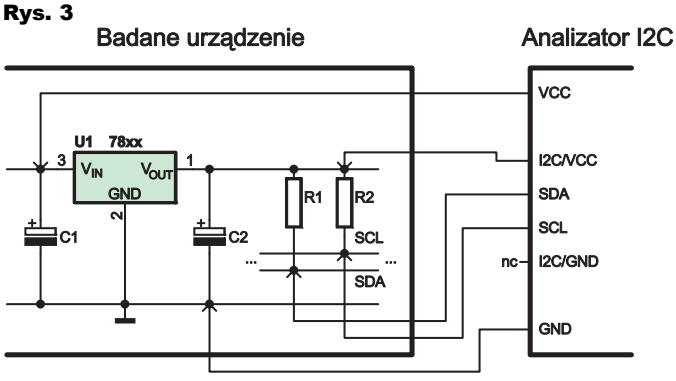
Układ może analizować szyny o poziomach napięć w bardzo szerokim zakresie, bo od kilkudziesięciu mV do około 30V! Wszystko to dzięki komparatorom umieszczonym na wejściu układu. Aż tak szeroki zakres w ogromnej większości przypadków nie będzie wykorzystywany. Zwykle szyny mają poziomy 5 lub 3,3V. Niekiedy także 2,7 lub nawet 1,8V. Ale zawsze lepiej mieć zapas.

Część programowa

Żeby nie zajmować artykułu długimi listingami oraz szybko i dokładnie wyjaśnić działanie programu, posłużę się pewnymi uproszczeniami. Na początek listing z takim uproszczonym (pseudo)programem (**listing 1**). Jak widać, programem sterują przede wszystkim obydwa przerwania INT0 i INT1. Jedno wykrywa zbocze opadające, a drugie narastające sygnału SDA. Program tuż po uruchomieniu czyści bufor (jest to konieczne, aby potem znaleźć miejsce, gdzie kończą się dane). Następnie zezwala na obsługę przerwań INT0 i INT1 i wchodzi w nieskończoną pętlę Do Loop. Jedynie przerwania INT0 i INT1 mogą go z tej pętli wyprowadzić (przerwanie od UART-a też, ale w inny sposób – o tym później).

W obsłudze przerwań znajduje się sprawdzenie sygnału SCL. Jak

Rys. 3




```
Scl Alias Pind.4
Sda Alias Pind.2
'-----------------------------------------
Config Int0 = Falling
Config Int1 = Rising
Enable Urxc
Label_Start:
Clear_ram
Loadadr Dane(1) , X
Clr_INT0_INT1_Flag
Enable Int0
Enable Int1
Label_wait_to_start:
Enable Interrupts
Do
Loop
 Label_start_receiving_data:
Enable Interrupts
      Odbierz_7      <----------------------
      Odbierz_6      <----------------------
Do
--->  Odbierz_5        <----------------------
--->  Odbierz_4        <----------------------
--->  Odbierz_3        <----------------------
--->  Odbierz_2        <----------------------
--->  Odbierz_1        <----------------------
--->  Odbierz_0        <----------------------
--->  Odbierz_Ack      <----------------------
        st X+, Wzkaźnik_Rodzaju_danych + Ack
        st X+, Data7…0
--->  Odbierz_7
        If Buffer = Full Then Goto Koniec
--->  Odbierz_6
Loop
Koniec:
Disable Int0
Disable Int1
Enable Interrupts
st X, Buffer_Full
!Label_Stop:
Enable Interrupts
Do
Loop
'-----------------------------------------
Int0_handler:
sbis Pind, 4
    reti
st X+, Start
pop R0
pop R0
If Buffer = Full Then Goto Koniec
GoTo Label_start_receiving_data
'-----------------------------------------
Int1_handler:
sbis Pind, 4
    reti
st X+, Stop
pop R0
pop R0
If Buffer = Full Then Goto Koniec
GoTo Label_wait_to_start
'-----------------------------------------
Uart_receive_int:
    In R16, UDR
    Loadadr Dane(1) , X
    pop R8
    pop R9
    Uart0_send Buffer
    Uart0_send R8
    Uart0_send R9
    Calculate CRC
    Uart0_send CRC
GoTo Label_Start
'-----------------------------------------
Uart0_send:
    !USART0_Transmit:
    sbis UCSR0A, UDRE0
    rjmp USART0_Transmit
    !Out UDR, R16
Return
'-----------------------------------------
Uart_printbin:
    Printbin Dane(1)
Return
'-----------------------------------------
```

wiadomo, linia SDA nie może się zmieniać w momentach, gdy SCL jest w stanie wysokim. Wyjątkiem są sygnały Startu i Stopu. Właśnie zmiana sygnału SDA podczas gdy SCL = 1 pozwala na wykrycie tych impulsów. Gdy SCL = 0, ma miejsce normalne przesyłanie danych. Rozkaz SBIS nie ominie więc instrukcji RETI i nastąpi normalne zakończenie obsługi przerwania. Jeżeli jednak SCL = 1, oznacza to, że nastąpił impuls Start lub Stop. Instrukcja SBIS wykonuje krótki skok, omijając instrukcję RETI. Po ominięciu tej instrukcji procesor zapisuje do bufora wskaźnik odebrania impulsu Start (gdy jest w INT0) lub Stop (gdy jest w INT1).

Odebranie impulsów Start lub Stop musi definitywnie zakończyć odbieranie danych. Odebranie impulsu Start musi zapoczątkować odbieranie danych od początku. Z kolei odebranie impulsu Stop musi spowodować przejście do fragmentu programu, w którym procesor oczekuje na pojawienie się impulsu Start. Dlatego też w tym momencie nie może nastąpić wyjście z przerwania za pomocą standardowej instrukcji powrotu RETI. Należy wykonać skok do odpowiedniego miejsca w programie. W przypadku odebrania impulsu Start jest to *Label_start_receiving_data*, a w przypadku impulsu Stop: *Label_wait_to_start*.

Wykonanie skoku – nic prostszego. Jednak należy pamiętać o dwóch sprawach. Po pierwsze, podczas wchodzenia do przerwania na stosie wylądował adres powrotu. Nie chodzi w zasadzie o adres, lecz po prostu o zajęcie dwóch bajtów na stosie. Przed wykonaniem skoku należy zatem zwolnić te dwa bajty. Do tego potrzebne są na przykład dwie instrukcje „pop R0".

Druga sprawa jest związana z bitem I w rejestrze SREG. Bit ten jest kasowany podczas wchodzenia do przerwania (sprzętowo przez procesor), aby w momencie wykonywania tego jednego przerwania nie mogło zostać wykonane inne przerwanie. Jest to takie zabezpieczenie przed przepełnieniem stosu w momencie, gdy każde przerwanie odkłada na stos dużo rejestrów. Dzięki kasowaniu tego bitu każde kolejne przerwanie jest obsługiwane dopiero po zakończeniu poprzedniego (a więc i zwolnieniu stosu). Instrukcja RETI automatycznie ustawia ten bit podczas jej wykonania. Jeżeli jednak wykonamy skok z przerwania i ręcznie nie ustawimy z powrotem bitu I, to program się zawiesi. Zawiesi się, bo żadne przerwanie od tego momentu nie będzie się mogło wykonać. Wykona się co najwyżej kilka rozkazów w programie głównym, ale program główny jest uzależniony od przerwań i bez nich po prostu w końcu się zatrzyma (tzn. wejdzie w nieskończoną pętlę). Dlatego też należy ustawić bit I. Wykonuje to rozkaz „Enable Interrupts" (rozkaz „SEI"). Nie umieściłem go jednak w przerwaniu, ale już za etykietami, do których skacze program po wykonaniu przerwań. Początkowo obawiałem się zawieszana programu, jeżeli nastąpi przerwanie tuż po ustawieniu bitu I, ale przed wykonaniem rozkazu skoku (a więc gdy zgłoszenie kolejnego przerwania nastąpi podczas wykonywania poprzedniego przerwania). Instrukcja RETI kasuje ten bit równolegle z wykonaniem skoku (powrotu) i problemu nie ma, jednak my ustawiany ten bit i skaczemy osobno. Wnikliwa analiza wykazała jednak, że takie przerwanie pierwszego przerwania innym przerwaniem nie jest niebezpieczne i nie spowoduje niczego złego. Jednak pozostawiłem rozkaz SEI poza przerwaniami.

Po odebraniu impulsu Start skok wykonuje się do etykiety *Label_start_receiving_data*. Po niej następuje wspomniane zezwolenie na obsługę przerwań. Następnie w pętli następuje programowe odczytywanie kolejnych przesyłanych magistralą I^2 bitów. Po odebraniu ostatniego bitu odbierany jest bit Ack i odebrane dane trafiają do bufora.

A dlaczego pętla odbierająca kolejne bity jest taka dziwna? Normalnie wystarczyłoby po zapisaniu w buforze danych (dwie instrukcje „ST X+, R1x") wykonać skok do etykiety *Label_start_receiving_data*. Jednak tak nie jest. Powód jest prosty: Zapisanie danych w buforze zajmuje trochę czasu, a wykonanie skoku oczywiście też. Tak więc po odebraniu impulsu ACK do odebrania kolejnego bitu procesor ma do wykonania więcej roboty niż zwykle. W tym przypadku maksymalną częstotliwość ogranicza najwolniejsza część programu. Tak więc ograniczona zostałaby maksymalna częstotliwość analizowanej transmisji. Lepiej czasochłonne fragmenty programu porozrzucać pomiędzy odbiór kolejnych bitów niż

Kodowana wartość	Bajt pierwszy									Bajt drugi							
	b7	b6	b5	b4	b3	b2	b1	b0		b7	b6	b5	b4	b3	b2	b1	b0
EMPTY	0	0	0	0	0	0	0	0									
I2C START	0	0	0	**1**	0	0	0	0									
I2C STOP	0	0	**1**	0	0	0	0	0									
I2C DATA	0	**1**	0	0	0	**ACK/NACK**	0	**D0**		**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**1**
BUFFER FULL	**1**	0	0	0	0	0	0	0									

Tabela 1

Tabela 2

I2C DATA = 0	0	**1**	0	0	0	**NACK**	0	**0**		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**

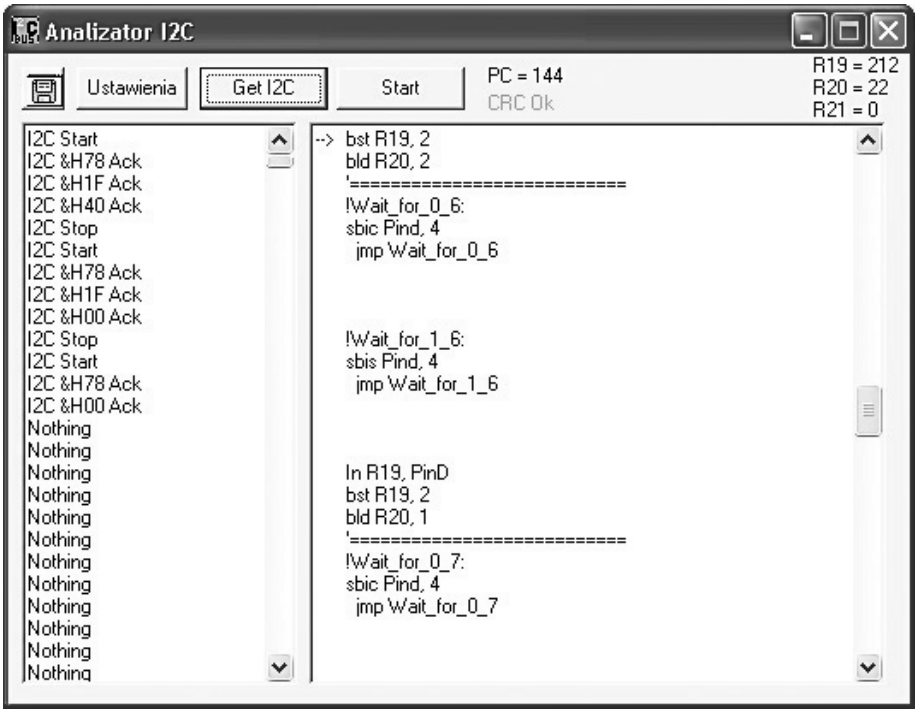
kumulować je w jednym miejscu. Właśnie taka dziwna pętla pozwala na szybszy zapis i szybszy skok. Zapis następuje w jednym momencie, po czym od razu następuje odbiór kolejnego bitu. Skok następuje dopiero dwa cykle odbioru później (wystarczyłby jeden cykl odbioru później, ale dzięki dwóm udało się tam jeszcze umieścić sprawdzanie stopnia napełnienia bufora). Skok następuje więc w momencie, gdy nie następuje zapis, a zapis, gdy nie następuje skok.

Skoro po zapisie danych ma nastąpić odebranie bitu nr 7 (bez wykonania skoku), to oczywiste jest, że kod odpowiadający za odbieranie tego bitu musi się znaleźć tuż za zapisaniem poprzedniego bajtu do bufora. Więc nie może to być ten sam kod, który odbierał siódmy bit podczas odbierania pierwszego bajtu. Stąd kolejne bajty będą już korzystały z innych fragmentów programu odbierających bity nr 7 i nr 6. Odbiór pierwszego bajtu danych jest zaznaczony na listingu 1 zielonymi strzałkami. Odbiór kolejnych bajtów – niebieskimi.

Wspomniane wcześniej sprawdzanie napełnienia bufora także działa w dość sprytny sposób. Sprawdzanie dwubajtowego adresu trwałoby raczej długo, więc zrealizowałem to trochę inaczej. Sprawdzana jest po prostu zawartość przedostatniego bajtu bufora. Normalnie bufor jest wyzerowany. Z kolei jakakolwiek inna wartość oznacza, że przedostatni bajt jest już zajęty. I jest to znak, że nie należy już dalej napełniać bufora i zatrzymać odbiór.

Sprawdzany jest przedostatni, a nie ostatni bajt, bo niektóre wartości zapisywane do bufora są dwubajtowe (chodzi o wartość I2CDATA przesłaną magistralą I^2. Pierwszy bajt oznacza, że kolejny bajt przechowuje przesłaną wartość). Gdyby ostatni bajt był wolny, program by uznał, że jeszcze można napełniać bufor, ale przecież taka dwubajtowa wartość by się tam nie zmieściła. Stąd sprawdzanie przedostatniego bajtu. Wtedy zawsze jest miejsce na dwubajtową wartość.

Ostatnia wartość w takim dwubajtowym słowie mogłaby także zawierać wartość 0, gdyby odebrany poprzez I^2bajt miał wartość zero. Zostałoby to omyłkowo zinterpretowane przez program, jako wolne miejsce w buforze. Dlatego odbierane z szyny I^2bajty są dzielone i do jednego bajtu bufora trafia jeden bit, a do drugiego pozostałe 7 bitów odebranego z magistrali bajtu. Ostatni wolny bit w drugim bajcie jest zawsze ustawiony, dzięki czemu



Rys. 5

wartość żadnego bajtu nigdy nie wynosi 0, nawet dla przechowywanej w nim wartości równej „I2CData = 0". Wszystkie dozwolone wartości mogące znajdować się w buforze są przedstawione w **tabeli 1**. **Tabela 2** przedstawia kodowanie wspomnianej wcześniej wartości I2CData = 0.

Jak już wspomniałem na samym początku, program zaczyna odbiór dopiero wtedy, gdy zostanie odebrany poprawny sygnał Start. Dzięki temu, gdy analizator zostanie włączony w momencie, gdy w magistrali I^2już są przesyłane jakieś dane, analizator grzecznie poczeka na zakończenie transmisji i zacznie podbierać dopiero po kolejnym impulsie Start (może to być także Repeat Start – wykorzystywany podczas odbioru danych, gdy po wysłaniu adresu urządzenia i adresu odczytu należy bez przerywania transmisji przełączyć się do odczytu). Program wprawdzie będzie obsługiwał przerwania INT1 i INT0, ale w obsłudze tych przerwań dojdzie do wniosku, że w momencie zmiany sygnału SDA, sygnał SCL ma wartość 0, czyli nie jest to sygnał Start/Stop, tylko normalny przesył danych.

Odebranie impulsu Stop definitywnie kończy odczyt danych. Program skacze do miejsca gdzie oczekuje na impuls Start (a więc do etykiety *Label_wait_to_start*).

Odebranie impulsu Start/Stop gdzieś w środku odbierania danych oznacza błąd transmisji. W takim momencie program także kończy odczyt i skacze do miejsca, gdzie oczekuje na sygnał Start (gdy odebrano Stop) lub zaczyna odbiór danych (gdy odebrano Start).

W przerwaniu od UART-a jest umieszczony kod odpowiadający za wysłanie całego bufora do komputera. Jak widać, wyjście z przerwania także realizowane jest przez skok. Jest to skok do (prawie) samego początku programu, gdzie czyszczony jest bufor i program oczekuje na całkowicie nową „sesję" danych.

Jak widać, każde z przerwań ostatecznie kończy odczyt danych i skacze do miejsca, gdzie oczekuje na nowe dane. Tak więc zawartość rejestrów nie jest dla nas istotna (nawet rejestru statusu SREG). Dlatego też podczas wchodzenia do obsługi przerwań nie jest zachowywany absolutnie żaden rejestr (nawet SREG!). Jedynym wyjątkiem są przerwania INT0 i INT1, które czasami normalnie kończą swoje działanie (tj. instrukcją RETI). Jednak wykonanie instrukcji SBIS i RETI nie zmienia zawartości żadnego rejestru (to znaczy zmienia bit I w

rejestrze SREG, ale on jest od tego, żeby się w przerwaniu zmienić). A więc w tym przypadku także nie musi zachowywać stanu żadnego rejestru.

Zdjęcie ze stosu adresu powrotu w przerwaniu od UART-a nie na końcu tuż przed wykonaniem skoku, ale już na początku obsługi przerwania ma pewien sens. Wysyłanie danych przez port szeregowy realizowane jest w podprogramie. Wejście do podprogramu powoduje zajęcie dwóch bajtów stosu. Dwa bajty adresu powrotu z przerwania + dwa adresu powrotu z podprogramu to 4 bajty zajętego stosu. Jeżeli jednak zdejmiemy adres powrotu przed wywołaniem procedur wysyłania danych, to zaoszczędzimy te dwa bajty (adres powrotu z podprogramu zajmie dokładnie te same dwa bajty, które wcześniej zajmował adres powrotu z przerwania). Zaoszczędzimy tylko 2 bajty, ale praktycznie żadnym kosztem (nie rozbudowujemy programu – zmieniamy tylko kolejność kilku linijek). A dzięki mniejszemu stosowi mamy więcej miejsca na bufor danych.

A teraz najlepsze: adres powrotu jest w pewien sposób wykorzystywany! Ale już nie w mikrokontrolerze. Jest on wysyłany do komputera! Sposób wykorzystania jest tak samo zaskakujący, jak przydatny. Adres powrotu jest miejscem, gdzie wróciłby procesor po normalnym wykonaniu przerwania. Jeżeli transmisja I^2 się zawiesi (na przykład zniknie sygnał zegarowy po odebraniu czwartego bitu danych), to program będzie czekał w nieskończoność (na sygnał zegarowy umożliwiający odbiór piątego bitu danych). Jeżeli zauważymy zawieszenie się transmisji, wystarczy odebrać bufor z analizatora do komputera. Odbierzemy także adres powrotu, a więc adres zawieszenia się programu. Pozwoli nam to dowiedzieć się, w którym momencie transmisja się urwała.

Przeliczenie wszystkich możliwych adresów powrotu na miejsca w kodzie programu jest proste, ale czasochłonne, więc zadałem sobie trochę trudu i robiłem to sam. Tak więc czytelnicy EdW dostają do ręki kompletne narzędzie wyświetlające na ekranie miejsce zawieszenia się programu! (**rysunek 5**).

Odbierane są także rejestry R19, R20 i R21. Pozwoli to wyznaczyć wartości jakie miały odebrane bity niekompletnego bajtu.

Dla dociekliwych i zaawansowanych

Program przysporzył niesłychanych problemów podczas uruchamiania. Opiszę je, bo naprawdę czegoś takiego się nie spodziewałem. Może dzięki temu ktoś zaoszczędzi sobie wielu nieprzespanych nocy.

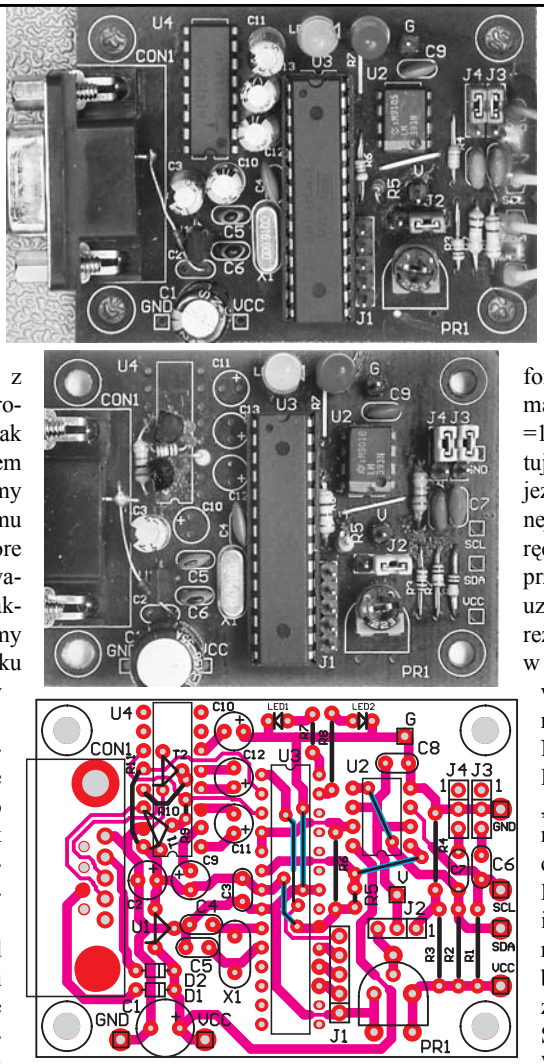
Pierwsza rzecz (na którą przygotowałem się już kilka lat temu, pisząc dekompilator) to błędy kompilacji. Początkowo układ projektowałem dla procesora ATmega88, gdyż nie spodziewałem się aż tak szybkiego wykonywania się programu i chciałem po prostu zastosować szybszy procesor, aby podołał zadaniu. Nic nie działało, ale już po kilku minutach doszedłem, co jest nie tak. Zdekompilowałem program i szybko załapałem, że BASCOM ma źle przypisane adresy do niektórych rejestrów specjalnych oraz że stosuje nieodpowiednie rejestry do konfiguracji niektórych bloków mikrokontrolera. Kiedyś naprawiałem takie błędy, grzebiąc w plikach konfiguracyjnych BASCOM-a, ale już tego nie robię. Gdyby ktoś (nie mając takiego „naprawionego" BASCOM-a) skompilował mój program, to on by mu po prostu nie działał. Dlatego wolę po prostu ręcznie wpisać w asemblerze nazwę rejestru albo jego adres i po sprawie.

W tym konkretnym przypadku BASCOM inicjalizując przerwania (INT0 = Rising…) wartości wpisywał do rejestru MCUCR zamiast EICRA. MCUCR występuje zarówno w ATmega8, jak i w ATmega88. Ale w procesorze ATmega88 ten rejestr służy już do czegoś innego. Natomiast do ustalania rodzaju przerwań używa się rejestru EICRA. MCUCR to po prostu pozostałość po procesorze ATmega8. W ATmega8 ten rejestr służy między innymi do ustalania sygnałów, które będą wyzwalały przerwania INT0 i INT1. Chyba wszystkie rejestry, które procesor ATmega88 odziedziczył po ATmega8, jeżeli tylko zmieniły swoje przeznaczenie, zmieniły swoją nazwę. Ale ten jeden rejestr nie i prawdopodobnie projektanci bibliotek kompilatora BASCOM po prostu zapomnieli o nim, co zaowocowało tym błędem.

A po co linia: „Printbin Dane(1)"? Ten fragment, mimo że nigdy nie zostanie wykonany, umożliwia działanie całego programu. Na początku programu jest informacja dla kompilatora, która mówi, jaka ma być prędkość UART-a. Chodzi o: „$baud =19200". Jednak kompilator nie zaimplementuje inicjalizacji sprzętowego układu UART, jeżeli w kodzie programu nie znajdzie żadnej instrukcji Print lub Input. Żeby uniknąć ręcznego konfigurowania układu UART oraz przeliczania wartości jakie należy wpisać dla uzyskania różnych prędkości i to przy różnych rezonatorach kwarcowych, wystarczy wpisać w kodzie np. „Printbin Dane(1)", a BASCOM, widząc to, sam wszystko policzy i skonfiguruje!

Niespodzianką jest także fragment „LDS R24, {Dane+1017}" dla ATmega88 oraz „LDS R24, 1113" dla ATmega8 (tych fragmentów nie ma na listingu 1, tzn. są uproszczone do postaci „If Buffer = Full Then"). BASCOM-owi oczywiście coś nie pasowało i dla procesora ATmega8 nie chciał poprawnie przeliczyć adresu zmiennej. Dlatego byłem zmuszony samemu wyliczyć adres zmiennej Dane(1016) i ten adres podać BASCOM-owi na tacy, jako liczba 1113.

Wszystkie wymienione błędy dotyczą jednej konkretnej wersji BASCOM-a. Inne mogą działać inaczej. Ostatnio BASCOM jest mocno pogardzany przez wielu programistów. Jednak inne środowiska także mają błędy podobne do wymienionych.



Rys. 6

Montaż i uruchomienie

Układ można zmontować na płytce drukowanej pokazanej na **rysunku 6**. Montaż jest klasyczny. Najlepiej zacząć od wlutowania pięciu zwór. Ze względu na małe wymagania co do procesora (ATmega8 zastosowałem tylko dlatego, aby mieć długi bufor danych) można zastosować w zasadzie każdy inny procesor AVR, oczywiście taki z pamięcią RAM (więc ATtiny11, ATtiny12 i ATtiny15 odpadają). Oczywiście w przypadku wykorzystania innego procesora niż ATmega8/88 należy zastosować inną płytkę drukowaną, albo zamontować procesor na przewodach łączących jego nóżki z odpowiednimi punktami lutowniczymi.

W wersji podstawowej będzie montowany układ U4 wraz ze współpracującymi kondensatorami C9–C12. Diody D1 i D2 nie będą montowane. Zamiast MAX232 i kondensatorów można także zastosować elementy T1, T2, R10, R11, R10 (Uwaga! Montować w wymienionej kolejności! W przeciwnym razie mogą być problemy ze zmieszczeniem rezystorów). Wtedy będzie

można zaryzykować zasilanie układu z portu COM.

Do punktów I2C/VCC, SCL, SDA i I2C/GND można przylutować przewody zakończone krokodylkami lub innymi elementami umożliwiającymi łatwe podłączenie do badanego układu. Z praktyki wiem, że przylutowane do płytki przewody dość łatwo ulegają uszkodzeniu w miejscu, gdzie były robione nacięcia na izolacji przed jej ściągnięciem (a więc będzie to miejsce tuż nad płytką). Aby się przed tym zabezpieczyć, można zalać przewody klejem lub jeszcze lepiej tworzywem mięknącym pod wpływem temperatury. Ja zrobiłem to, używając kleju termicznego (kleju do pistoletów klejowych) rozgrzanego stacją Hot Air (ustawioną na 160°).

Sygnały do wtyku DB9F są podłączone w taki sposób, aby możliwe było podłączenie układu bezpośrednio do złącza w komputerze. Może nie jest to zbyt wygodne, ale ma pewną zaletę. Mianowicie pozwoli podłączyć układ bezpośrednio do konwertera USB-RS232. Jeżeli chcemy podłączyć układ do gniazda znajdującego się z tyłu obudowy komputera, to możemy to zrobić, ale wtedy musimy wykręcić śrubki z gniazda DB9F. Takie połączenie nie będzie jednak wygodne. Lepiej użyć kabla, z tym że musi to być kabel bez przeplotu.

Zworki J2, J3 i J4 należy ustawić w pozycjach „1”. Podłączamy zasilanie, układ do komputera, a przewody pomiarowe do jakiegoś układu korzystającego z magistrali I^2. Najlepiej gdyby to był już działający układ. Uruchamiamy program sterujący na komputerze. Po naciśnięciu przycisku Ustawienia mamy możliwość wybrania numeru portu COM. Powracamy do okna głównego i klikamy przycisk Start. Naciśnięcie tego przycisku powoduje zapoczątkowanie nowej sesji pomiarowej. Po napełnieniu się bufora do pełna nastąpi zaświecenie diody LED1. Zapełnienie bufora jest raczej normalnym objawem, więc jest to dioda żółta. Z kolei wejście w odbiór (po sygnale I2CSTART) i brak sygnału I2C-STOP zaowocuje ciągłym świeceniem diody LED2. To nie jest już normalny objaw, oznacza on zawieszenie się transmisji I^2. Dlatego dioda LED2 jest czerwona.

Oczywiście bufor można odebrać także przed całkowitym jego zapełnieniem. Do odbierania danych służy przycisk Get I2C. Przesyłane dane są zabezpieczone za pomocą prostej sumy kontrolnej. Błąd sumy kontrolnej zaowocuje wyświetleniem się odpowiedniego komunikatu ostrzegawczego. W takim przypadku wystarczy ponownie nacisnąć przycisk Get I2C. Odbieranie danych (poprzez naciskanie przycisku Get I2C) nie powoduje wyczyszczenia bufora. Czyszczenie następuje dopiero po zainicjowaniu nowej sesji pomiarowej (a więc po naciśnięciu przycisku Start). Widok okna programu widać na rysunku 5. Jeżeli próby wypadną pomyślnie, to układ jest gotowy do pracy.

Andrzej Jabłoński
atom1477@wp.pl

Wykaz elementów

Półprzewodniki

U1	78L05
U2	LM393
U3	ATmega8(88) - 28 DIP
U4	MAX232
LED1	LED 5mm żółta
LED2	LED 5mm czerwona
T1	BC557 (tylko w wersji 2)
T2	BC547 (tylko w wersji 2)
D1,D2	1N4148 (tylko w wersji 2)

Rezystory

R1,R2	100kΩ
R3	10kΩ
R4	12kΩ
R5,R6	4,7kΩ
R7,R8	470Ω (1,5kΩ dla wersji 2)
R9-R11	10kΩ (tylko w wersji 2)
PR1	22kΩ PR

Kondensatory

C1	220µF/16V (220µF/35V)
C2	10µF/10V
C3,C8	100nF ceramiczny
C4,C5	20p (18…22p)
C6,C7	470p
C9-C12	10µF/16V

Inne

X1	kwarc 16MHz
J1	listwa goldpin 1x5 pin
J2-J4	listwa goldpin 1x3 pin
V,G	goldpin 1 pin
CON1	DB9F do druku
Podstawka DIP28	
Jumper x 3	

Komplet podzespołów z płytką jest dostępny w sieci handlowej AVT jako kit szkolny AVT-2899.