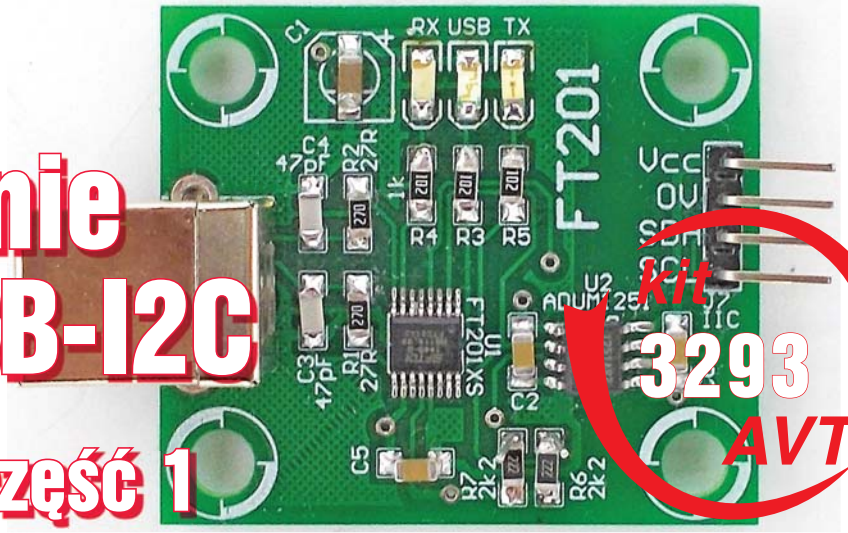
Izolowany galwanicznie mostek USB-I2C

część 1



Prezentowany w artykule konwerter zapewnia izolację galwaniczną między interfejsem USB a mikrokontrolerem. Komunikacja z konwerterem odbywa się nie przez UART, tylko z wykorzystaniem I2C. Interfejs I2C w porównaniu z UART ma wiele zalet.

Opisywane urządzenie służy do galwanicznej izolacji interfejsu USB od mikrokontrolera. Istnieją gotowe izolatory USB, na przykład ADuM1460, ale do tanich nie należą. Zdecydowanie taniej można zbudować izolator z wykorzystaniem układów FT201 i ADuM1250. Układ FT201 jest mostkiem USB-I2C i w stosunku do mostka z interfejsem UART ma szereg zalet, zwłaszcza przy współpracy z AVR, PIC czy zapomnianym już 8051:

– AVR mają mało UART. Jest to szczególnie odczuwalne w przypadku ArduinoUNO. Niektóre AVR mają 2 UART, nieliczne Mega po cztery, ale są one montowane w obudowach 100pin.
– Zegar AVRmega/tiny musi być stabilizowany rezonatorem kwarcowym. Wbudowany RC ma zbyt małą stabilność, aby wykorzystać go z UART.
– Wbudowane w mostki USB-UART FIFO nie spełnia swojej roli. Mostki mają FIFO, ale jeżeli znaki, które przyszły z USB, są wysyłane do UART, czy uC tego chce, czy nie. Ten problem najbardziej odczuwalny jest w Arduino z AVRmega/tiny, w którym podczas komunikacji z WS2812 czy 1-Wire najczęściej zawieszane są przerwania.
– Kontrola przepływu wymaga dodatkowych GPIO uC oraz implementacji jej po stronie HOST-a, co nie zawsze jest możliwe, przykładowo gdy nie mamy kodów źródłowych HOST-a.
– Używając UART, nie można stwierdzić, czy USB HOST jest przyłączony, czy nie. Do tego trzeba zaangażować kolejne linie mikrokontrolera.
– Konfigurację mostka, czyli VID, PID, desktyptor, funkcje GPIO, itd. można przeprowadzić (o ile w ogóle można, bo w np. CP2101 nie) tylko z poziomu komputera odpowiednią aplikacją. Dla układów FTDI jest to FT_PROG. Nie można tego zrobić z poziomu uC.

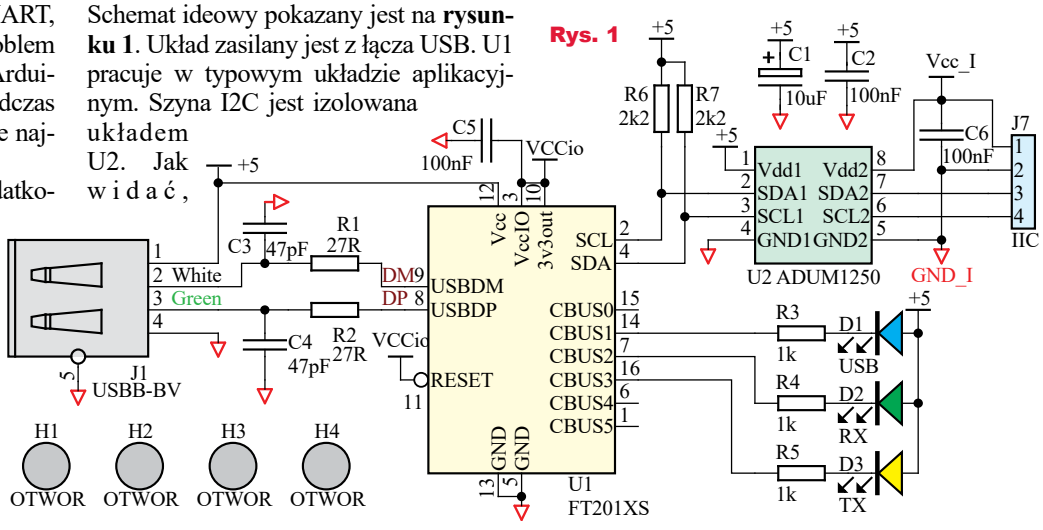
Wszystkich wyżej wymienionych wad pozbawiony jest układ FT201:

– Komunikacja interfejsem I2C do 3,4MHz (w projekcie konwertera prędkość ogranicza ADuM1250 do 1MHz).
– I2C akceptuje logikę 5V.
– Prędkość komunikacji USB 1Mb/s.
– Dwa bufory FIFO 512 bajtów.
– Wszystkie opcje konfiguracji programem FT_PROG dostępne z poziomu interfejsu I2C.
– Ponad 1kB EEPROM do dyspozycji użytkownika.
– 5 konfigurowanych linii GPIO, które między innymi mogą sygnalizować nieodebrane znaki w FIFO, wolne miejsce w FIFO nadawczym, sterowanie LED-ami sygnalizacyjnymi.

Opis układu

Schemat ideowy pokazany jest na **rysunku 1**. Układ zasilany jest z łącza USB. U1 pracuje w typowym układzie aplikacyjnym. Szyna I2C jest izolowana układem U2. Jak widać, budowa sprzętowa jest banalna, cała moc leży w oprogramowaniu. Zanim przystąpię do opisu oprogramowania, wyjaśnię funkcjonowanie FIFO w mostkach USB. W przypadku mostków z UART, dane przychodzące po USB są zapisywane w FIFO, skąd są wysyłane przez UART. Nie znam sposobu, aby zatrzymać wysyłanie danych z FTDI. Zmiana stanu linii CTS czy DTR powoduje tylko zmianę stanu tych wirtualnych linii dostępnych w HOST przez API. HOST może reagować na stan CTS/RTS, ale to, co już jest w buforze układu FTDI, musi zostać wysłane. Z tego powodu reakcja na zmianę CTS/RTS nie jest natychmiastowa i w przypadku układów FTDI w najgorszym przypadku mikrokontroler może jeszcze otrzymać 512 znaków od czasu zmiany stanu linii CTS/RTS. Ponadto, nie zawsze program będzie reagował na stan owych linii. Jeśli nie mamy kodów źródłowych, nic z tym się nie da zrobić. Arduino nader często blokuje przerwania. W przypadku transmisji do LED WS2812 taka blokada może trwać kilkadziesiąt milisekund, a przy prędkości 115200, blo-

Rys. 1



kada na dłużej niż ok. 173µs spowoduje gubienie znaków. W przypadku 921600 wystarczy ok. 22µs.

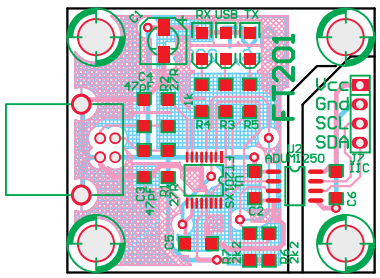
Z FT201 jest inaczej. Dane przychodzące po USB są tak jak i w przypadku mostków z UART zapisywane w FIFO, ale FIFO będzie odczytane dopiero wtedy, gdy zrobi to mikrokontroler, bo FT201 jest układem slave i sam z siebie nie wyśle danych na I2C. Pozwala to na zdecydowanie dłuższy czas blokować przerwania mikrokontrolera, a nawet obsłużyć komunikację I2C bez użycia przerwań.

Montaż i uruchomienie

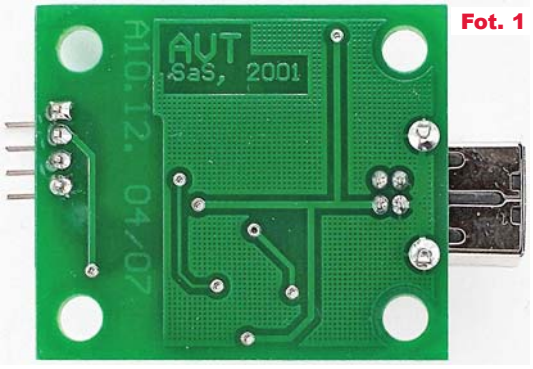
Układ można zmontować na płytce drukowanej, której projekt pokazany jest na **rysunku 2**. Układ montujemy standardowo, zaczynając od elementów najmniejszych, a kończąc na największych.

Fotografia wstępna oraz **fotografia 1** pokazują model. Układ zmontowany prawidłowo ze sprawnych elementów powinien od razu pracować.

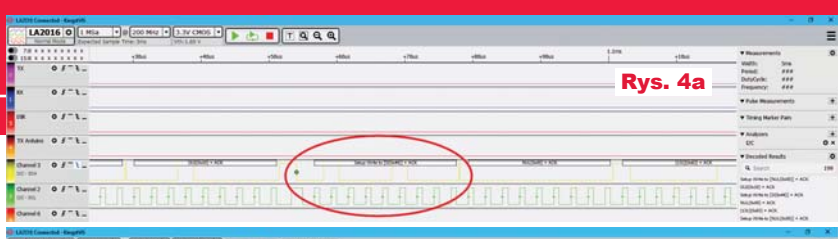
Obsługa programowa. Podstawowa funkcjonalność związana z transmisją danych jest osiągalna w zadziwiająco prosty sposób. Aby wysłać dane po USB, wystarczy zaadresować układ slave o adresie 0x22 (0x44) do zapisu. Adres 0x22 jest domyślnym adresem układu FT201, można go zmienić programem FT_PROG (**rysunek 3**) lub modyfikując obszar pamięci MTP przez mikrokontroler, o czym później. ***Uwaga! Wszystkie rysunki – zrzuty z artykułu o dużej rozdzielczości są dostępne w Elportalu wśród materiałów dodatkowych do tego numeru EdW.*** Adres w programie FT_PROG wpisujemy w postaci liczby szesnastkowej 7-bit. Trzeba o tym pamiętać, bo łatwo o pomyłkę, ponieważ adres 0x22 w analizatorze czy na oscyloskopie jest reprezentowany w postaci liczby 0x44 przy zapisie (**rysunek 4a**) i 0x45 przy odczycie (**rysunek 4b**). Programy operują na różnych adresach, przykładowo Arduino używa adresowania 7-bitowego, co oznacza, że domyślnym adresem FT201 jest 0x22, natomiast HAL STM32 posługuje się adresem 8-bitowym, więc

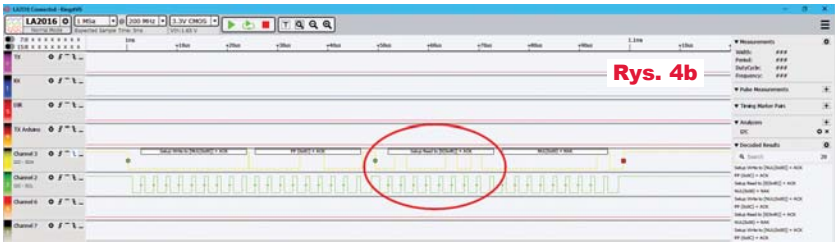


Rys. 2



Fot. 1

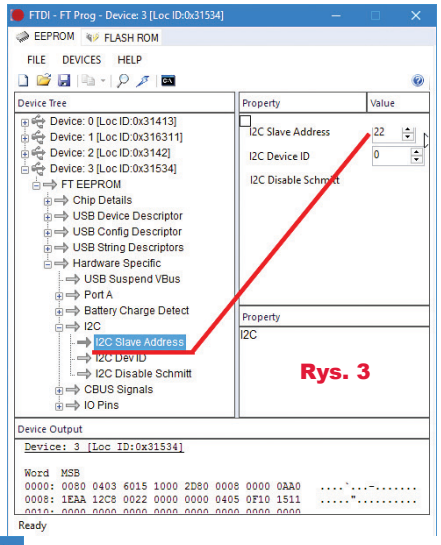

Rys. 4a


Rys. 4b

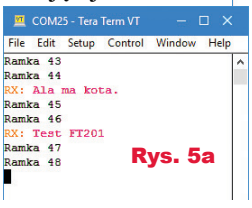
co widać w kodach źródłowych, należy posługiwać się adresem 0x44. Po zaadresowaniu slave brak potwierdzenia ACK świadczy o zapełnionym buforze FIFO.

Odbiór jest równie prosty. Po zaadresowaniu slave do odczytu należy sprawdzić sygnał ACK. Jeśli go nie ma, to w FIFO nie ma znaku do odczytu, jeśli jest, w FIFO znajduje się co najmniej jeden znak. Na **listingu 1a** znajduje się fragment programu wysyłania i odbioru danych do/z FT201 napisany dla STM32L412, natomiast **listing 1b** przedstawia program dla Arduino. ***Uwaga! Wszystkie listingi do tego artykułu są dostępne w Elportalu wśród materiałów dodatkowych do tego numeru EdW.***

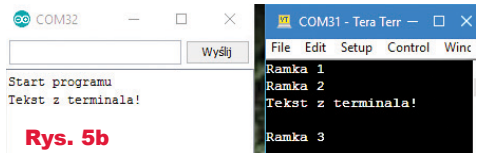
Program wysyła co dwie sekundy napis „Ramka” oraz jej numer i odsyła przychodzące dane, poprzedzając je tekstem „RX:” – **rysunek 5a**. Program dla Arduino nie odsyła tekstów na terminal, tylko wyświetla w monitorze portu szeregowego – **rysunek 5b**. Programy w pełnej wersji dostępne są też w materiałach dodatkowych na Elportalu.



Rys. 3



Rys. 5a

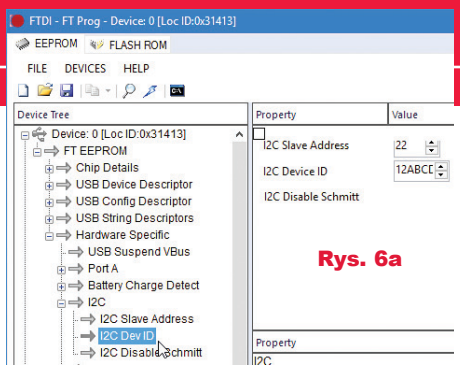


Rys. 5b

Czas podnieść poprzeczkę i wykorzystać zaawansowane możliwości układu. Jak odczytać status układu czy liczbę bajtów zgromadzonych w FIFO? Adresowanie i odczyt układu nie mają sensu, bo zwraca on zawartość FIFO. Aby nie rezerwować kolejnego adresu, konstruktorzy zdecydowali się na wykorzystanie adresu broadcastowego. Ze względu na to, że na adres ten reagują wszystkie układy na magistrali, adres broadcastowy nie służy do odczytu/zapisu rejestrów układu FT201, tylko do wysyłania komend. Aby odczytać, ile bajtów zgromadzono w FIFO, należy:

– Wygenerować START.
– Zaadresować slave 0 (broadcast) do zapisu.
– Wysłać komendę, w przypadku odczytu liczby bajtów w FIFO 0x0C.
– Wygenerować ponowny START. **Nie może to być STOP-START, musi być ponowny start!**
– Zaadresować FT201 do odczytu.
– Odczytać bajt bez ACK. W bajcie zawarta będzie informacja o liczbie bajtów w FIFO.
– Wygenerować STOP.

Rodzi się pytanie, jak jednym bajtem odczytać liczbę 9-bit? FIFO ma 512 bajtów, aby więc poinformować o liczbie bajtów, potrzebne są dwa bajty. Przyznam, że nie wiem, w nocie katalogowej też nic na ten temat nie znalazłem. Pisząc artykuł, opierałem się na przykładzie z noty („AN_255_USB to I2C Example using the FT232H and FT201X devices”, strona 25), a wynikiem jest **listing 2**.



Rys. 6a

Wszystko na to wskazuje, że gdy danych będzie więcej niż 255, FT201 zwróci wartość 255. Po odczytaniu danych z FIFO, kolejny odczyt liczby bajtów w FIFO zwróci resztę, która została. Na **listingu 3a** pokazana jest funkcja odczytująca liczbę bajtów w FIFO dla STM32F411 i zmodyfikowana funkcja odsyłająca dane.

Zdziwić może, dlaczego nie użyłem HAL-a? Czyżby nie istniała funkcja, która wysyła ponowny START? Istnieje (**`HAL_I2C_Mem_Read`**), ale niestety, twórcy HAL nie przewidzieli, że adres do zapisu może być inny niż do odczytu. Ze względu na to, że zrozumienie HAL STM32 jest zadaniem trudnym, prościej było napisać ten fragment, posługując się rejestrami. Ta sama funkcja dla Arduino jest na **listingu 3b**.

Tu też skorzystałem z rejestrów, bo biblioteki Arduino są niedopracowane. Standardowe wcale nie pozwalają na wygenerowanie ponownego startu (HAL STM32 tylko wtedy, gdy adresy slave się różnią). To bardzo dziwne postępowanie, zwłaszcza że wiele układów korzysta z tej funkcjonalności. Ponadto, bufor nadawczy jest ograniczony do 32 bajtów. Bez modyfikacji bibliotek zwiększenie bufora nadawczego zwiększy także bufor odbiorczy. W konsekwencji, gdyby chcieć wykorzystać maksymalny dostępny rozmiar dla FT201 o wielkości 512 bajtów, zwiększenie bufora dla I2C spowodowałoby zużycie połowy RAM dostępnej w ArduinoUNO!

Trochę odejdę od tematu głównego i opiszę problem, z którym już kiedyś się spotkałem i dał mi się we znaki podczas pisania oprogramowania dla FT201. W magistrali I2C, w specyficznych warunkach może dojść do sytuacji, że slave zablokuje magistralę. Może tak się stać wtedy, gdy mikrokontroler zostanie zresetowany w momencie, gdy slave wystawia zero na magistrali danych. W takiej sytuacji, po resecie, nie można poprawnie przeprowadzić komunikacji po i2C, ponieważ master stwierdza, iż magistrala jest zajęta przez inny master. Szanse na zaistnienie takiego zdarzenia są tym większe, im częściej odczytuje się dane ze slave. W przypadku FT201, gdy nie używamy przerwań od FIFO, odpytywanie jest bardzo częste (dobry powód, aby jednak korzystać z przerwań). Jak wyjść z takiej patowej sytuacji, gdy slave blokuje magistralę? Najprościej zresetować slave, ale nie każdy układ ma taką możliwość, na przykład FCP8574. Co wtedy? Wystarczy wygenerować dziewięć impulsów na linii SCK, po czym warunek stopu. Niestety nie da się zrobić tego z układu I2C master wbudowanego w mikrokontroler. Należy go wyłączyć i prostym programem wygenerować impulsy oraz STOP. Funkcję taką można także wywołać, gdy nagle tracimy komunikacje z układami slave. Czasem pomaga. Kod takiej funkcji dla STM32 i miejsce jej umieszczenia pokazuje **listing 4a**. To samo dla AVR pokazuje **listing 4b**.

Gdy już mamy pewną obsługę I2C odczytamy status FT201. Operacja przebiega tak samo jak w przypadku odczytu liczby bajtów w FIFO, aby więc nie tworzyć kolejnych funkcji, stworzymy uniwersalną, której argumentem będzie komenda dla układu FT201 jak na **listingu 5a**.

Status jest liczbą z zakresu 0...3 a oznacza:

0x00 Suspended
0x01 Default
0x02 Addressed
0x03 Configured

Jeśli układ został skonfigurowany przez HOST, status zwraca 3. Odczyt statusu dla Arduino na **listingu 5b**.

Kolejną funkcją jest odczyt ID układu. ID można ustawić programem FT_PROG – **rysunek 6a**. ID zawiera trzy bajty, w FT_PROG wprowadza się je w formie liczby szesnastkowej. Aby odczytać ID przez I2C, nie jest używany jak poprzednio adres broadcastowy, tylko używa się dodatkowego adresu układu FT201: 0xF8 (0x7C). Niestety, operacja nie jest standardowa, tak jak i poprzednia z adresem broadcastowym. Aby odczytać ID, należy:

– Wygenerować START.
– Zaadresować slave 0xF8 (0x7C) do zapisu.
– Wysłać adres układu FT201 do odczytu.
– Wygenerować ponowny START (nie może być stop-start musi być ponowny start).
– Zaadresować slave 0xF9 (0x7C) do odczytu.
– Odczytać bajt bez ACK. W bajcie zawarta będzie informacja o liczbie bajtów w FIFO.
- Wygenerować STOP.

W tym celu napisałem funkcję pokazaną na listingu 6a. Odczyt ID na Arduino na **listingu 6b** (tylko w Elportalu).

Te karkołomne sztuczki z adresem broadcastowym i F8 pozwalają na używanie wielu układów na jednej magistrali, a jednocześnie nie trzeba rezerwować osobnych adresów dla danych i komend lub wydłużać transmisji o dodatkowy bajt informujący, czy chcemy operować na danych USB, czy na rejestrach układu FT201.

Na koniec pozostawiłem najciekawszą możliwość FT201: konfigurowanie układu z poziomu mikrokontrolera. **Wszystko, co można zrobić programem FT_PROG, można zrobić także z poziomu mikrokontrolera!** Daje to duże możliwości.

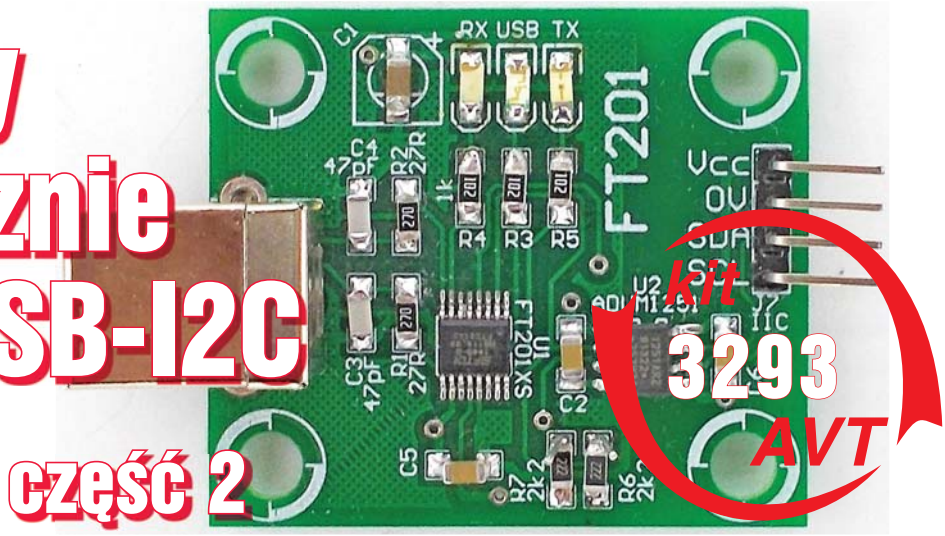
– Pierwsza to fakt, że nie trzeba wgrywać programu do urządzenia dwa razy, raz programu dla mikrokontrolera, za drugim razem konfiguracji dla FTDI.
– Kolejna zaleta to ewentualny upgrade programu. Przykładowo mikrokontroler może pobrać najnowszą wersję programu z Internetu i zaprogramować siebie, ale co z mostkiem USB? Gdy jest to mostek UART, a istnieje konieczność zmiany konfiguracji układu, to mamy sytuację patową. W przypadku FT201 problemu nie ma.
– Kolejny przykład to wymiana uszkodzonego mostka USB. W przypadku standardowych układów trzeba jeszcze wgrać konfigurację, a w przypadku FT201 konfigurację może ustawić mikrokontroler.
– Mikrokontroler może sprawdzić konfigurację, pozwalając zabezpieczyć się przed „grzebaniem" w niej przez osoby postronne.
– MTP i powiązana z nią pamięć EEPROM pozwala na umieszczanie w niej danych, które nie ulegną zniszczeniu po wymianie mikrokontrolera. Pozwala to tworzyć licznik czasu pracy, zabezpieczenia czy konfigurację w postaci dodatkowej kopii przydatnej, gdy zawartość pamięci EEPROM w mikrokontrolerze ulegnie uszkodzeniu.
– Aplikacja może odczytać informacje o urządzeniu, korzystając z EEPROM, przy czym nie ma tu ograniczenia liczby danych do 32, jak w przypadku deskryptora USB.

Ciąg dalszy w następnym numerze.

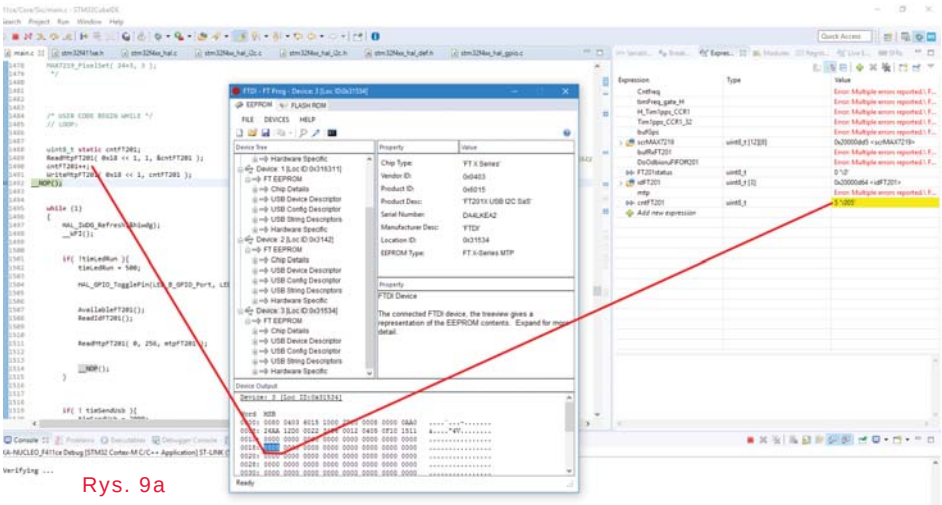
SaS
sas@elportal.pl

Izolowany galwanicznie mostek USB-I2C

część 2



Kontynuujemy opis układu FT201. Zanim opiszę sposób konfigurowania pamięci MTP, zacznę od prostszego zagadnienia, pamięci EEPROM. Cała pamięć FT201 jest podzielona na kilka obszarów **rysunek 7**. Zielone obszary są do dowolnego wykorzystania przez użytkownika, przy czym obszar 0x24...0x7F (słowa 0x12...0x3F) jest widoczny w oknie programu FT_PROG – **rysunek 8**. Pozostałe obszary kontrolowane są 16-bitową sumą kontrolną. Jeśli będzie błędna, układ przyjmie standardową konfigurację. Aby odczytać zawartość pamięci EEPROM, należy ją najpierw zaadresować. Jak łatwo się domyślić, jest używany do tego mechanizm dostępu przez adres broadcastowy. Po zaadresowaniu można odczytać bajt. Nie będę szczegółowo opisywał tych funkcji, zainteresowanych odsyłam do kodów źródłowych. Dla większości użytkowników wystarczy wiedza, że funkcja „`uint16_t ReadMtpFT201( uint16_t adres, uint16_t len, uint8_t *buf )`” odczytuje bajt/bajty spod adresu „adres” do bufora „buf”. Liczba bajtów zawiera argument „len”. Odczytanie całej pamięci MTP może wyglądać tak, jak na **listingu 7**.



Rys. 9a

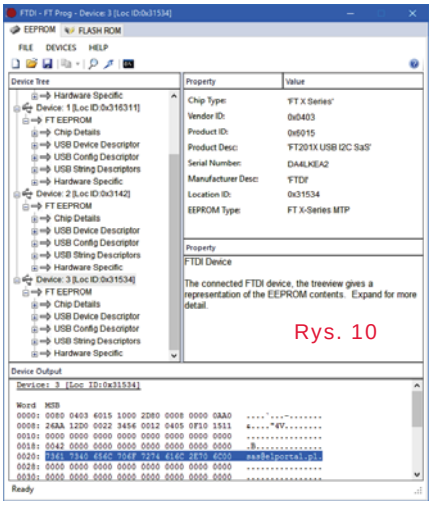
Zapis bajtu jest równie prosty z punktu widzenia funkcji „`uint16_t WriteMtpFT201( uint16_t adres, uint8_t data )`”. Na listingu prosty program zwiększający komórkę 0x30 (młodszy półbajt słowa 0x18), jak pokazuje **listing 8**.

Memory Area Description	Word Address	Byte Address
User Area 2 Accessible via USB and I²C	0x3FF - 0x80	0x7FF – 0x100
Checksum	0x7F	0xFF – 0xFE
String Descriptor Area Accessible via USB and I²C	0x7D - 0x50	0xFB – 0xA0
FTDI Configuration Area Cannot be written	0x4F - 0x40	0x9F – 0x80
User Area 1 Accessible via USB and I²C	0x3F - 0x12	0x7F – 0x24
Chip Configuration Area Accessible via USB and I²C	0x11 - 0x00	0x23 - 0x00

Rys. 8

Figure 3.1: Simplified memory map for the FT-X

W przykładowym programie funkcja była wywołana przed pętlą główną, więc zliczała liczbę resetów mikrokontrolera. Efekt działania funkcji widać w debuggerze i programie FT_PROG **rysunek 9a**. Wersja Arduino na **rysunku 9b** (tylko w Elportalu). Licznik na jednym bajcie nie ma praktycznego zastosowania, ale nie ma problemu, aby rozszerzyć go do 4 bajtów. Korzystając z tego, że obszar do 0xFF (do słowa 0x80), z czego dla użytkownika 0x12...0x3F (słowa 0x24...0x7F), jest widoczny w okienku FT_PROG, można w nim przechowywać informacje o wersji programu i na przykład adres e-mail autora programu, **rysunek 10**. Ze względu na to, że FT_PROG wyświetla słowa, a nie bajty, nie można wprost wpisać tekstu, należy zamienić bajty parzyste z nieparzystymi. Zrealizowałem to prostym programem, jak pokazuje **listing 9**.

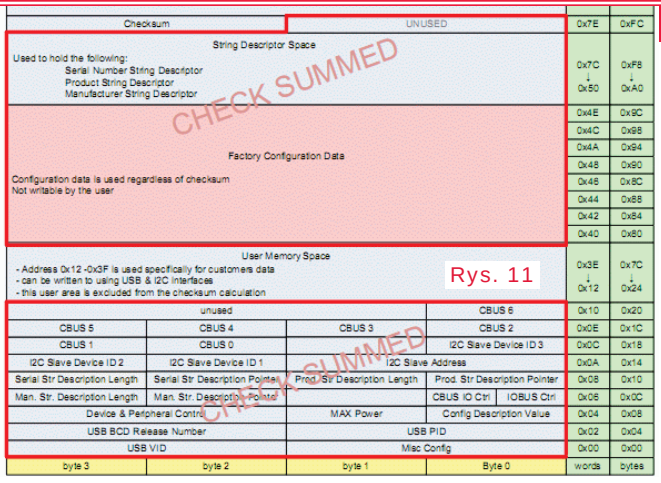


Rys. 10

Tekst ten jawnie nie musi występować w kodzie programu, można go zaszyfrować

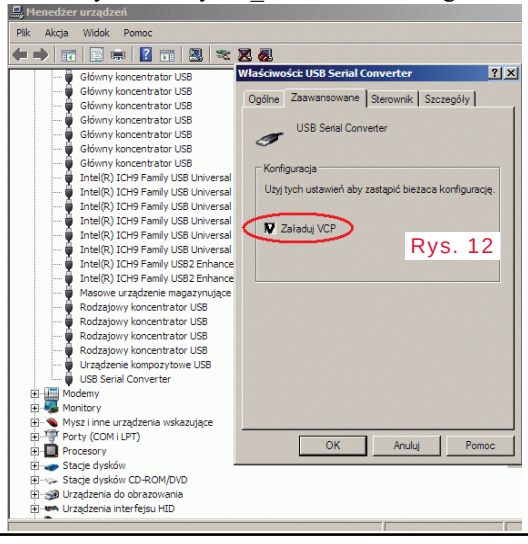
```
Device Output
0040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0048: 0000 0000 0000 030A 0046 0054 0044 0049  .........F.T.D.I
0050: 0322 0055 0053 0042 0020 003C 002D 003E  .".U.S.B. .<.->
0058: 0020 0032 0078 0052 0053 0032 0033 0032  . .2.x.R.S.2.3.2
0060: 0043 0312 0032 0078 0052 0053 0032 0033  .C...2.x.R.S.2.3
0068: 0032 0043 0000 0000 0000 0000 0000 0000  .2.C............
0070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0078: 0000 0000 0000 0000 0000 0000 0000 37C5  ..............7.
Ready
```

Rys. 7



kluczem o długości samego tekstu. Tekst jest widoczny w oknie FT_PROG (w tym przypadku specjalnie dałem go w obszarze 0x24...0x7F), ale można go umieść tam, gdzie FT_PROG go nie pokaże, czyli w obszarze x0100...0x7FF (0x80...0x3FF). Sygnaturę można odczytać na komputerze przez D2XX lub przez uC. Całkiem dobry sposób na zabezpieczenie swoich praw. MCP2221 czy mostki z UART nie dają takiej możliwości.

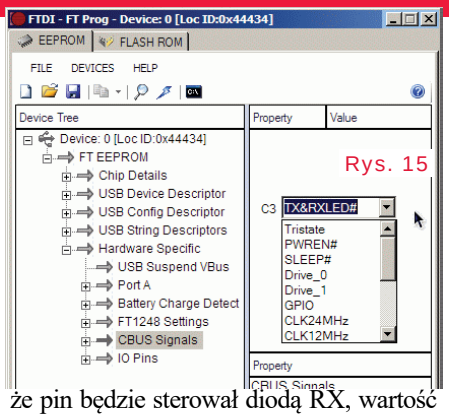
Obszar EEPROM i samej pamięci MTP może być odczytywany zarówno z poziomu mikrokontrolera, jak i programu w komputerze. Daje to możliwość odczytania przez HOST dużej liczny informacji o urządzeniu, przy czym nie ma tu ograniczenia liczby danych do 32 jak w przypadku deskryptora USB. Co ważne, sam mikrokontroler nie uczestniczy w tej operacji. Przykładowo licznik czasu pracy w mikrokontrolere jest nie do odczytania, a w FT201 nie ma z tym problemu. Można oczywiście taki licznik przechowywać w zewnętrznej pamięci EEPROM, ale jej odczyt wymaga dodatkowych zabiegów, w przypadku FT201 wystarczy do tego komputer. Nawet nie trzeba pisać specjalnej aplikacji, wystarczy darmowy FT_PROG i wiedza, gdzie



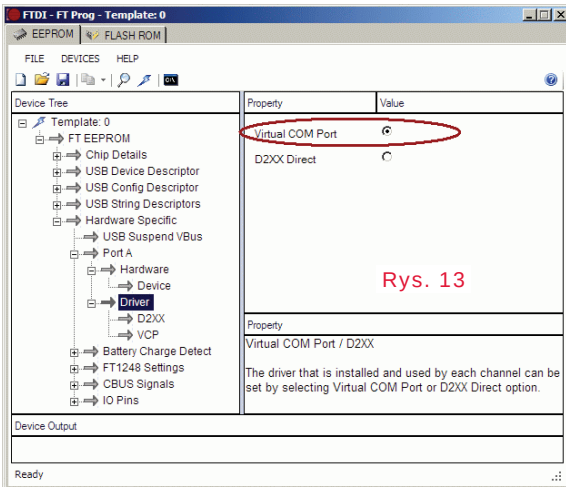
i jak są zapisane informacje.

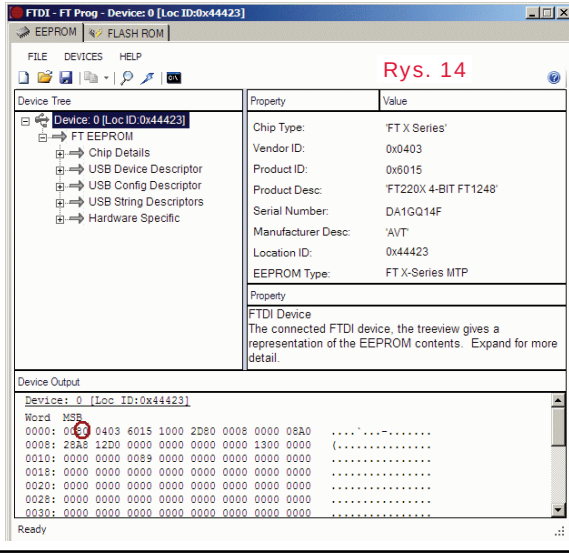
Teraz najtrudniejsze zagadnienie, obszar MTP. Na początek warto zapoznać się z mapą pamięci MTP – **rysunek 11**. Mapę można znaleźć na 8 stronie pliku *AN_201_FT-X MTP Memory Configuration*. W obszarze tym można zapisywać i odczytywać informacje o VID, PID, funkcjach CBUS, poborze prądu, itp. Oczywiście konfiguracja nieistniejącego CBUS nie odniesie skutku. O CBUS później napiszę coś więcej, teraz skupmy się na najbardziej potrzebnych opcjach. Obszar od 0xA0 do 0xF8 (w słowach 0x50..0x7C) przechowuje informacje o nazwie interfejsu, producencie, numerze seryjnym. Wskaźniki do tego obszaru zawierają bajty 0x0E..0x13. Jak później pokażę, wskaźników tych w wielu przypadkach nie trzeba liczyć. Dość istotne znaczenie ma pierwszy bajt. W nim jest zawarta informacja między innymi o tym, czy ładować biblioteki VCP, czy nie. W przeciwieństwie do układów z rodziny FT232, FT23x, w mostkach IIC/SPI domyślnym ustawieniem jest nieładowanie VPC (ładowanie D2XX). Jest to o tyle istotne, że taki układ nie będzie widziany jako wirtualny COM i komunikacja z nim będzie możliwa tylko przez biblioteki D2XX. Można to zmienić z poziomu menedżera urządzeń, **rysunek 12**, zaznaczając opcję „załaduj VCP" lub lepiej z poziomu FT_PROG – **rysunek 13.** Dlaczego sugeruję robić to z poziomu FT_PROG? Otóż jak zrobimy to w menedżerze urządzeń, to po zmianie deskryptora i enumeracji, konieczne będzie powtórzenie operacji w „menedżerze urządzeń", a gdy zrobimy to w FT_PROG, to ominie nas ta wątpliwa przyjemność. Oczywiście lepszym rozwiązaniem będzie zmiana bitu odpowiedzialnego za te funkcje z poziomu mikrokontrolera. Za ładowanie VCP odpowiedzialny jest siódmy bit (licząc od zera) pierwszego bajtu obszaru MTP – **rysunek 14**. Aby go ustawić, można skorzystać z kodu z **listingu 10**.

Konfiguracja CBUS. Bajty od 0x1A (0x34) do 0x20 (0x40) odpowiadają za konfigurację linii CBUS od 0 do 6. Wpisanie wartości 1 spowoduje,



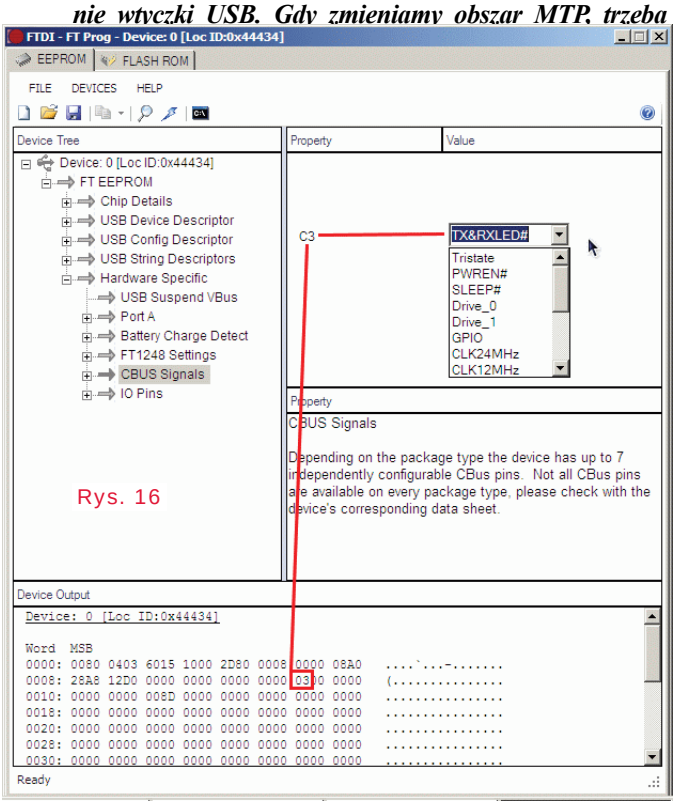
że pin będzie sterował diodą RX, wartość 2 TX, 3 RX+TX. W tablicy 7.18 na stronach 16 i 17 dokumentu „*AN_201_FT-X MTP Memory Configuration* opisano wszystkie możliwe ustawiania. Podczas prób zmiany ustawień CBUS napotkałem ciekawe zachowanie układu, a właściwie sterownika dla Windows. Z poziomu FT_PROG nie można uzyskać pewnych opcji, na przykład „LedTX&RX" dla FT22x/200x – **rysunek 15**. Można to jednak zrobić, modyfikując MTP z poziomu mikrokontrolera – **rysunek 16**. Mimo że opcji „LedTX&RX" nie ma na liście rozwijanej, pojawiła się w okienku!





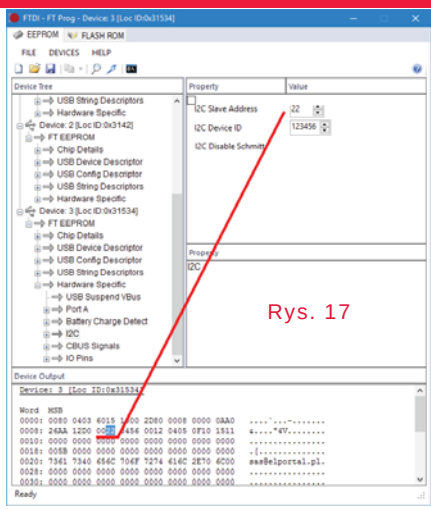
Trzeba jednak być ostrożnym z takimi operacjami, bo układ FT201 po tej zmianie został rozpoznany jako FT232H. Komunikacja działała, ale nie można zagwarantować, że inne opcje będą funkcjonowały bez problemu, choć teoretycznie nie powinno być z tym kłopotów, ponieważ układy FT20x/22x są okrojoną wersją FT232H.

Uwaga! MTP jest wczytywana raz po resecie FT201. Jeśli więc zostanie zmieniona jej zawartość (nie mylić obszaru MTP z obszarem EEPROM użytkownika), trzeba wymusić enumerację np. przez chwilowe odłączenie wtyczki USB. Gdy zmieniamy obszar MTP, trzeba zapisać poprawną sumę kontrolną, którą można wyliczyć funkcją `CalculateMtpCrc`. Modyfikując pozostały obszar EEPROM nie zapisujemy CRC ponieważ jest on wyłączony z kontroli.

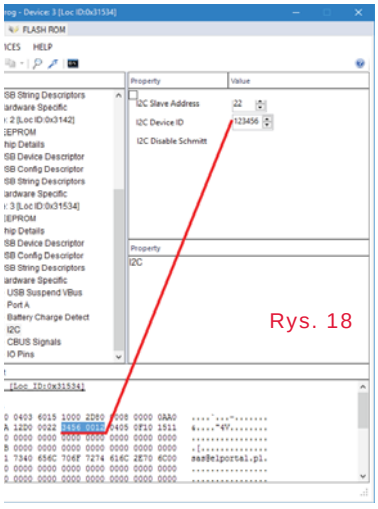


Rys. 16

Na zakończenie opiszę kilka innych obszarów pamięci MTP. W bajcie 0x14 i 0x15 (słowo 0x0C) zapisany jest adres slave I2C. Domyślny adres to 0x22. Sprawdźmy, czy się zgadza w ustawianiach FT_PROG – **rysunek 17**. Pora sprawdzić identyfikator. Powinien być w bajtach 0x16...0x18 (słowo 0x0B i młodszy bajt słowa 0x0C) – **rysunek 18**. Próba odczytu analizatorem (**rysunek 19**) i w programie przy użyciu debuggera (**rysunek 20**) potwierdzają poprawność konfiguracji.



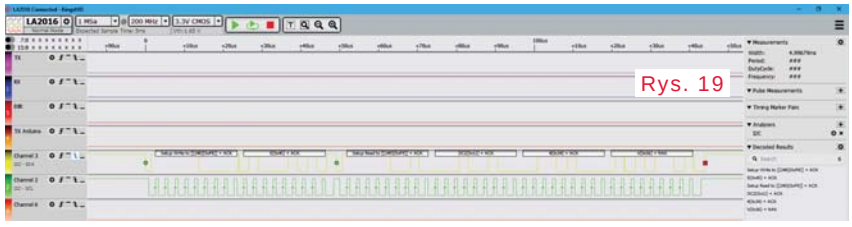
Rys. 17



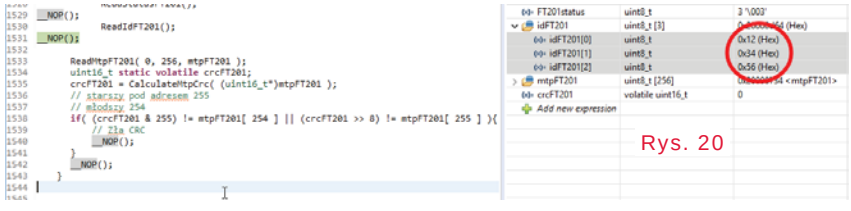
Rys. 18

Debugger to potężne narzędzie w rękach programisty. Niestety Arduino jest pozbawione debugera. Jest jednak alternatywa, w przypadku AVR AtmelStudio, które importuje projekty z Arduino i pozwala na debugowanie. Jeśli Czytelnicy są zainteresowani artykułem na ten temat, proszę pisać do redakcji.

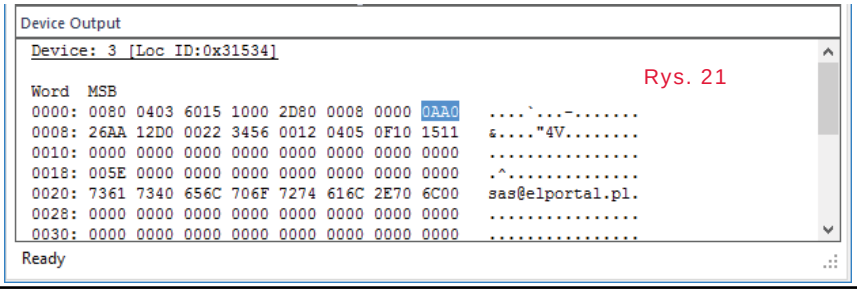
Bajty 0x0E i 0x0F (słowo 0x07) określają nazwę producenta – **rysunek 21**. Zawierają wskaźnik 0xA0, długość 0x0A. Bajt 0xA0 / 2 = słowo 0x50. Długość 0x0A = 10. 10 / 2 = 5 słów. Podglądamy zawartość pamięci – **rysunek 22**. Coś nie do końca się zgadza. Tekst ma długość czterech, a nie pięciu znaków i zaczyna się od jakichś tajemniczych 0x030A. Sprawdzamy więc nazwę produktu. Bajty 0x10 i 0x11 (słowo 0x08) zawierają adres nazwy



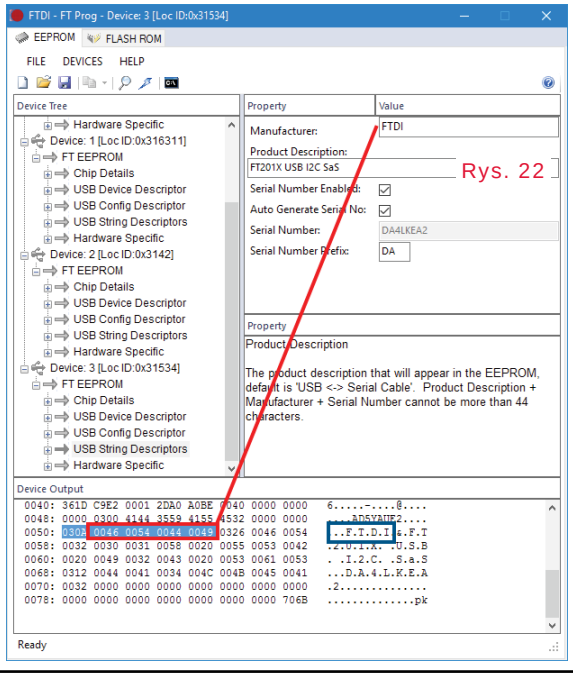
Rys. 19



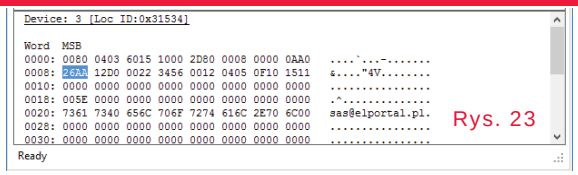
Rys. 20



Rys. 21



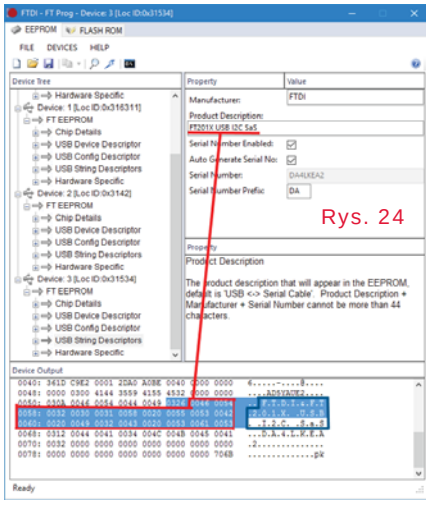
Rys. 22



Rys. 23

deskryptora i jego długość. Na co wskazują bajty? **Rysunek 23** pokazuje wskaźnik 0xAA, długość 0x26, 0xAA / 2 = 0x55. 0x26 = 38, 38 / 2 = 19 słów. Sprawdzamy **rysunek 24**. Znów niezgodność długości o jeden znak i pierwsze słowo o wartości 0x0326, która zdaje się coś sugerować. Sprawdzimy więc jeszcze numer seryjny. Bajty 0x12 i 0x13 zawierają wskaźnik 0xD0 i długość 0x12 – **rysunek 25**. 0xD0 / 2 = 0x68, 0x12 = 18 / 2 = 9 słów. Numer seryjny zawiera 8 znaków i tajemniczy ciąg 0x0312. Łatwo wywnioskować, że pierwsze słowo zawiera w starszym bajcie 0x03, a w młodszym długość ciągu znaków w bajtach. Wszystko wskazuje na to, że można pisać „między wierszami", bo w słowie zawierającym znaki wykorzystany jest tylko jeden, młodszy bajt. Starszy jest do wykorzystania, ale nie sprawdzałem takiej możliwości.

Dwa bajty na znak do zakodowania kodów ASCII to rozrzutność. Gdyby było to kodowanie UITF-16, wszystko byłoby zrozumiałe, ale w deskryptorze są tylko kody ASCII. Dlaczego więc na znak zużyto aż dwa bajty? Prawdopodobnie wynika to z faktu, że w układach FTDI użyto mikrokontrolerów 16-bitowych. Na niektóre układy FTDI można pisać własne programy, a producent udostępnia nieodpłatnie IDE i dokumentację. Takimi układami są na przykład VNC-2, które mają dwa USB, które mogą pracować w trybie HOST/DEVICE. Nie znam tańszego układu niż VNC, który miałby dwa HOST-y USB. Jeśli Czytelnicy wykażą zainteresowanie programowaniem układów VNC, na łamach EdW pojawi się stosowny materiał.



Rys. 24

Opisywanie wszystkich bajtów konfiguracji nie ma większego sensu, zainteresowanych odsyłam do dokumentu *AN_201_FT-X MTP Memory Configuration*. Teraz tylko krótka porada: Jeśli układ FTDI ma być w całości skonfigurowany przez mikrokontroler, to nie ma potrzeby robić tego bit po bicie, bajt po bajcie. Wystarczy zrobić to programem FT_PROG, a następnie taką konfigurację przenieść do kodu źródłowego. Taką operację można przeprowadzić bibliotekami D2XX lub odczytać MTP mikrokontrolerem i wysłać w postaci kodu C/C++ na terminal. Taki kod wklejamy do kodu źródłowego. Teraz wystarczy, że po resecie mikrokontroler sprawdzi, czy CRC obszaru MTP jest zgodne z tym w kodzie źródłowym i w razie niezgodności zapisze MTP w FTDI. Trzeba pamiętać, aby odczyt MTP przeprowadzać, gdy układ jest skonfigurowany (funkcja „ReadStFT201()" zwróci 3). Według noty katalogowej, dostęp do MTP nie zawsze jest możliwy, dlatego operacje na tej pamięci trzeba weryfikować zarówno przy odczycie (na przykład dwa odczyty), jak i przy zapisie. ***Zmiany w obszarze MTP są widoczne zaraz po odczytaniu danych przez FT_PROG. Jednak VID, PID i inne parametry związane z deskryptorem system widzi po enumeracji urządzeń USB. Enumeracje można wywołać z poziomu systemu, przez wyjęcie i włożenie wtyku USB lub reset układu FTDI.***

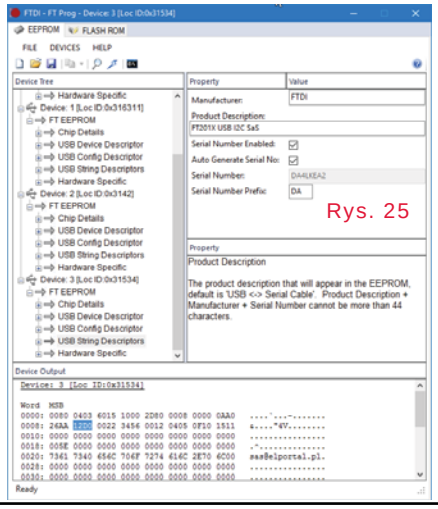
Wykaz elementów

R1,R2	27Ω 1206
R3,R4,R5	1kΩ 1206
R6,R7	2,2kΩ 1206
C1	10uF
C3,C4	47pF 1206
C2,C5,C6	100nF 1206
U1	FT201XS
U2	ADUM1250ARZ
D1	LED Niebieska 1206
D2	LED Zielona 1206
D3	LED Żółta 1206
J1	Gniazdo kątowe USB
J7	NS25-W4P

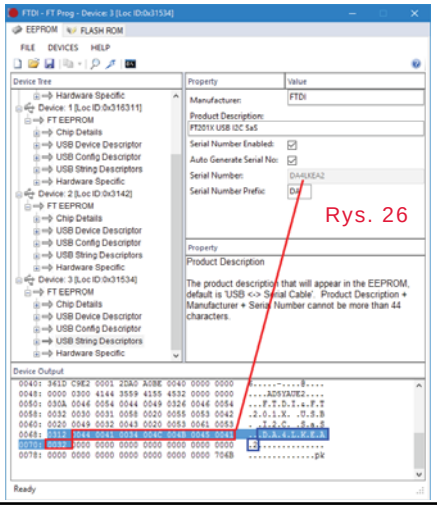
Płytka drukowana jest dostępna w Sklepie AVT jako AVT3293

Zamieszczone w Elportalu programy są demonstracjami przedstawiającymi, jak używać układu FT201. Nie mają zabezpieczeń, nie zrealizowano timeoutu, więc w razie problemów z I2C program zresetuje się (zadziała watchdog).

W przygotowaniu jest też artykuł opisujący obsługę układów FT22x. Jest on bliźniaczo podobny do FT201, tyle że komunikuje się z wykorzystaniem interfejsu SPI. W przypadku tego układu można bez problemu wykorzystać standardowe biblioteki Arduino, co paradoksalnie wynika z gorszych bibliotek dla SPI niż I2C. Nie używają one buforów, tylko on-line wysyłają/odbierają dane z SPI, co nie powoduje rezerwacji RAM na potrzeby buforów. Nie ma problemu z ponownym startem jak w I2C. Niestety, SPI to aż 5 przewodów, o jeden mniej niż dla alfanumerycznego LCD w jednokierunkowym trybie 4-bit. Mam nadzieję, że nikt nie wpadnie na to, aby użyć PCF8474 jak w LCD, aby zmniejszyć liczbę pinów potrzebną do sterowania układem FT22x. Mimo wady w postaci dużej liczby portów mikrokontrolera, FT22x mają dwie główne zalety: dużą prędkość komunikacji i możliwość zapisu/odczytu stanu linii modemowych. Dzięki drugiej zalecie można uniknąć latającego jak wariat wskaźnika myszy i klikania, gdzie popadnie, gdy uC wysyła dane po USB, którego VCOM nie jest otwarty. Pewnie z tego powodu sterowniki mają opcje VCOM albo DD2X. Nie ma VCOM, nie ma problemu z myszą. Czytelników zainteresowanych kostkami FT22x zachęcam do przysyłania e-maili do redakcji EdW, co przyspieszy publikację.



Rys. 25



Rys. 26

SaS
sas@elportal.pl