***** trzy gwiazdki – SMD, znajomość C/C++**

Izolowany galwanicznie mostek USB-SPI
Przykłady wykorzystania w Arduino, AVR i STM32

Prezentowany w artykule konwerter zapewnia izolację galwaniczną interfejsu USB od mikrokontrolera. Ponadto komunikacja z konwerterem odbywa się nie, jak to jest najczęściej przy pomocy UART ale z wykorzystaniem SPI. Interfejs ten, w stosunku do UART, w przypadku mostków USB ma wiele zalet.

Do czego to służy?
Opisywane urządzenie służy do galwanicznej izolacji interfejsu USB od mikrokontrolera. Istnieją izolatory USB, na przykład ADuM1460 ale do tanich nie należą. Zdecydowanie korzystniej jest zbudować izolator z wykorzystaniem układów FT220/221 i ADuM1402. Układy FT22x są mostkami USB-SPI i w stosunku do mostka z interfejsem UART mają szereg zalet, zwłaszcza przy współpracy z Arduino, AVR, PIC:

- AVR mają mało UART. Jest to szczególnie odczuwalne w przypadku ArduinoUNO. Niektóre AVR mają 2 UART, nieliczne Mega po cztery ale gdy są montowane w obudowach co najmniej 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 co z tego jak znaki, które przyszły z USB są wysyłane do UART czy mikrokotroler tego chce czy nie (wyjaśnienie w dalszej części artykułu). 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 mikrokotrolera oraz implementacji jej po stronie HOST-a, co nie zawsze jest możliwe, przykładowo nie mamy kodów źródłowych HOST-a.
- Przez UART nie można stwierdzić czy USB HOST jest przyłączony czy nie. Tak samo jak odczytać stanu linii sterujących przepływem (DTR, DSR) czy je ustawić (RTS, CTS, CDC, RI). Aby to zrobić 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 mikrokotrolea.

Wszystkich z wyżej wymienionych wad pozbawione są układy FT22x. Jakie mają możliwości?

- Komunikacja interfejsem SPI zegarem do 12MHz. SPI może pracować w trybach 1, 2, 4 a w przypadku FT221 także 8-bit.
- Automatyczne rozpoznawanie szerokości szyny SPI.
- Automatyczne rozpoznawania trybu pracy SPI pomiędzy MODE 1 a MODE 3.
- Prędkość komunikacji USB 1Mb/s (500kb/s dla FT220).
- 1kB FIFO, 512 bajtów bufora nadawczego, 512 odbiorczego.
- Wszystkie opcje konfiguracji programem FT_PROG dostępne z poziomu interfejsu SPI.

- Ponad 1kB EEPROM do dyspozycji użytkownika.
- Jedna konfigurowana linia GPIO.

Jak to działa?
Schemat ideowy pokazany jest na **rysunku 1**. Układ zasilany jest z USB. U1 pracuje w typowym układzie aplikacyjnym. Szyna SPI jest izolowana układem U4. Ze względu na to, że układ U4 nie może zmienić kierunku konwersji w czasie pracy oraz nie posiada pojedynczo sterowanych wyjść Open Drain, aby komunikować się z U1 zastosowano U2. Bramki 3-stanowe U2 pozwalają na zamianię interfejsu SPI 4-wire na 3-wire. Wyjście Open Drain (Open Colector) jest symulowane bramką U2C. Dzięki podłączeniu wejścia sterującego /OE U2C (pin 10) do linii danych MOSI oraz zwarciu wejścia U2C (pin) do masy, stan niski MOSI wymusza stan niski na wyjściu U2C (pin 8) natomiast stan wysoki na wejściu wymusza stan trzeci na wyjściu co jest tożsame (o ile nie jest przekroczone na wyjściu napięcie zasilania + ok 600mV) z pracą wyjścia Open Drain. Metoda ta jest powszechnie stosowana w układach FPGA, CPLD czy dawniej w GAL, które nie posiadają wyjść Open Drain. Linię sygnalizującą obecność znaku w odbiorczym FIFO izoluje transoptor U5. Rozwiązanie to ma pewną wadę związaną z szybkością pracy transoptora. Jak sobie poradzić z tym problemem opisane będzie w dalszej części artykułu. Linia CBUS3 U1 nie jest izolowana, jeśli istnieje taka potrzeba należy to zrobić poza płytką interfejsu.
Jak widać budowa sprzętowa nie jest przesadnie skomplikowana, cała moc interfejsu 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 najgorszym przypadku mikrokontroler może jeszcze otrzymać, w przypadku układów FTDI, 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, blokada na dłużej niż ok 173µs spowoduje gubienie znaków. W przypadku 921600 wystarczy ok 22µs.
Z FT22x jest inaczej. Dane przychodzące po USB są tak jak i w przypadku mostków z UART zapisywane w FIFO ale FIFO będzie odczytane dopiero gdy zrobi to mikrokontroler bo FT22x jest układem Slave i sam z siebie nie wyśle danych na SPI. Pozwala to na zdecydowanie dłuższy czas blokować przerwania mikrokontrolera a nawet obsłużyć komunikację SPI 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 (chyba zbędna)** pokazują model. Więcej fotografi znajduje się w materiałach dodatkowwych. Układ zmontowany prawidłowo ze sprawnych elementów powinien od razu pracować.

Obsługa programowa.
Zanim opiszę komendy układów FT22x przypomnę, że SPI układu FT22x pracuje w trybie MODE1 lub 3. Domyślne SPI jest ustawione w trybie MODE 0 dlatego należy to zmienić **rysunek 3**. Druga ważna informacja to konieczność zaznaczenia opcji „Virtual port COM” **rysunek 4** w FTDI.

Po zapisaniu konfiguracji konieczne jest przeprowadzenie ponownej enumeracji. Najprościej to zrealizować odłączając na chwile wtyczkę USB. Bez tego wirtualny port COM nie pojawi się w systemie i komunikacja z układem będzie możliwa tylko przez API bibliotek D2XX. Schemat podłączenia modułu z płytką NUCLEO-L412RB-P znajduje się na **rysunku 5**, ArduinoUNO na **rysunku 6**. Podstawowa funkcjonalność związana z transmisja danych jest osiągalna w zadziwiająco prosty sposób. Aby wysłać daną po USB, wystarczy ustawić wejście /SS w stan niski, wysłać komendę 0x00 a po niej bajt, który ma być wysłany przez USB. Po komendzie można wysłać więcej bajtów byleby nie przepełnić FIFO. Na **rysunku 7** widać początek wysyłania tekstu „Ramka 2" zakończonego znakami LF+CR. Dlaczego komenda „0" wysyłana jest w postaci bajtu o wartości 0x01 będzie wyjaśnione w dalszej części artykułu. Dane zarejestrowane analizatorem LA2016 są dostępne w materiałach dodatkowych.

<ramka>

W materiałach dodatkowych, poza przebiegami opisanymi w artykule znajdują się także inne, dotyczące odbioru danych, sprawdzania statusu modemu, komunikacji z pamięcią MTP. Ponadto można znaleźć oscylogramy pokazujące komunikację i z FT22x oraz fotografie pokazujące sposób podłączenia do NUCLEO L412 i ArduinoUNO. Przebiegi zarejestrowane analizatorem można obejrzeć bezpłatnym programem udostępnionym na stronie producenta http://www.qdkingst.com/en.

</ramka>

W stanie nieaktywnym wejścia /SS, na linii MIOSIO układ wystawia stan flagi TXE# informującej o wolnym miejscu w buforze nadawczym.

<ramka>

Ze względu na to, że układ w stanie nieaktywnym blokuje magistralę SPI, nie można w prosty sposób podłączyć innych układów do magistrali. Istnieje możliwość zablokowania wystawiania flagi TXE# w konfiguracji układu **rysunek 8.**

</ramka>

Jeśli TXE# jest wystawiane tylko gdy układ jest nieaktywny to jak stwierdzić podczas wysłania wielu bajtów, że FIFO jest zapełnione? Wątpliwości wyjaśnia **rysunek 9.** W czasie ósmego impulsu zegarowego na wejściu SCK, na wyjściu MISO (na schemacie z rysunku 1 linia SO) układ FT22x wystawia stan flagi TXE# natomiast dziewiątego impulsu potwierdza komendę. Odbiór bajtu z FIFO polega na uaktywnieniu wejścia /SS i wysłania komendy „1" (nie bajtu 0x01 tylko komendy „1"). Jeśli w FIFO znajduje się znak linia SO przyjmie poziom niski co widać na **rysunku 10** (odbiór kodu ASCII litery „a"), gdy znaku nie ma, linia SO będzie w stanie wysokim **rysunek 11**. Wyjaśnienia wymaga dlaczego na analizatorze dwa razy występuje linia SO. Kanał zero (fioletowy, etykieta „SO U4") reprezentuje sygnał na wyprowadzeniu układu FT22x, kanał 1 (niebieski, etykieta „SO OPTO") na wyjściu transoptora U5. Widać wyraźne przesunięcie sygnału, dlatego powolny transoptor nie pozwala na uzyskanie maksymalnych prędkości transmisji. Dokładnie zmierzone czasy opóźnienia zarówno oscyloskopem jak i analizatorem przedstawia **rysunek 12**. Tak się jednak składa, że korzystając ze standardowych bibliotek HAL dla STM32, nie trzeba martwić się opóźnieniami, ponieważ same biblioteki mają duży narzut czasowy przy rozpoczynaniu i kończeniu transmisji co widać po odstępach pomiędzy transmitowanymi bajtami. W przypadku AVR jest podobnie ale tam odstępy są spowodowane innymi czynnikami (głównie prędkością a raczej powolnością mikrokontrolera). Gdyby istniała konieczność szybkiej transmisji danych z wykorzystaniem DMA należy zastosować szybszy transoptor lub któryś z izolatorów ADuM na przykład ADuM1200. Wyjaśnienia wymaga fakt, że analizator w czasie wysyłania komendy „0" zarejestrował bajt 0x01, natomiast komendy „1" zarejestrował 0x03. Wynika to ze sposobu kodowania bitów komendy w bajcie **rysunek 13.** Dlaczego bity komendy są rozłożone w tak dziwny sposób w bajcie? Dzięki temu układ FT22x może automatycznie określić czy transmisja jest jedni, dwu, trzy, czy w przypadku układu FT221 8-bajtowa. Korzystając z faktu, że w bajcie komendy bit 0 jest nieistotny ustawiono go na jeden:

<listing>

```
#define cmdFT22x_WrChar         0b00000001      // Write request command (CMD=0x00)
```

```
#define cmdFT22x_GetChar        0b00000011      // Read request command (CMD=0x01 0x0001)
#define cmdFT22x_StModem        0b00000101      // Read modem status (CMD=0x02 0b0010)
#define cmdFT22x_WrModem        0b00000111      // write modem status (CMD=0x03 0x0011)
#define cmdFT22x_Flush          0b00010001      // write buffer flush (CMD=0x04 0x0100)
#define cmdFT22x_AdrMtp         0b00010011      // Adres MTP memory (CMD=0x05 0b0101)
#define cmdFT22x_WriteMTP       0b00010101      // write eeprom (CMD=0x06 0b0110)
#define cmdFT22x_ReadMTP        0b00010111      // READ MTP memory (CMD=0x07 0b0111)
#define cmdFT22x_RDstatus       0b10000001      // 0x08
```

<listing>

Dzięki temu, w transmisji 1-bit, taki stan przyjmuje linia MOSI po zakończeniu transmisji komendy (LSB transmitowany jest jako ostatni) co wpływa na to, że bramka U2C nie blokuje układu FT22x podczas wysyłania ewentualnej odpowiedzi przez niego. **Listing 1a** zawiera istotne fragmenty kodu programu dla STM32L412

<ramka>

Program był testowany na płytce NUCLEO-L412RB-P2. STM32L412 jest doskonałym zamiennikiem popularnego, niewiele tańszego STM32F103 ale o dużo mniejszych możliwościach niż L412 i mniejszej pamięci RAM (20kB vs 80kB w L412).

</ramka>

odpowiedzialne za wysyłanie i odbieranie danych po magistrali USB oraz praktyczny przykład wysyłania i odbierania danych:

<listing 1a>

```
//-----------------------------------------------------------------//
void SPI_FT22x_SSl(){
        HAL_GPIO_WritePin( SS_FT220_GPIO_Port, SS_FT220_Pin, GPIO_PIN_RESET );
}


//-----------------------------------------------------------------//
void SPI_FT22x_SSh(){
        HAL_GPIO_WritePin( SS_FT220_GPIO_Port, SS_FT220_Pin, GPIO_PIN_SET );
}


//-----------------------------------------------------------------//
uint8_t txSendFT20x, rxSendFT20x, statSendFT20x;
uint32_t statSendFT20xerr;
//-----------------------------------------------------------------//
uint8_t FT22xout( uint8_t TXdana ){
        HAL_SPI_Transmit( &hspi1, &TXdana, 1, 1 );

        return statSendFT20x;
}


//-----------------------------------------------------------------//
uint8_t FT22xin(){
        uint8_t static RXdana;


        uint8_t TXdana = oxff;
        HAL_SPI_TransmitReceive( &hspi1, &TXdana, &RXdana, 1, 1 );


        return RXdana;
}


//-----------------------------------------------------------------//
uint8_t RD_SO_FT22x(){
        return HAL_GPIO_ReadPin( SO_FT22x_GPIO_Port, SO_FT22x_Pin );
}


//-----------------------------------------------------------------//
uint8_t StFT22x;
//-----------------------------------------------------------------//
```

```
uint8_t PutCharFT22x( char znak )
{
	uint8_t st;


	SPI_FT22x_SSl();
	FT22xout( cmdFT22x_WrChar );		// Write request command (CMD=0x00)
	st = RD_SO_FT22x();
	FT22xout( znak );
	SPI_FT22x_SSh();


	return( st );
}


//-----------------------------------------------------------------//
uint8_t PrintStringFT22x( char *text )
{
	char st;


	SPI_FT22x_SSl();
	FT22xout( cmdFT22x_WrChar );// Write request command (CMD=0x00)
	st = RD_SO_FT22x();
	if ( st ){ SPI_FT22x_SSh(); return( false ); }
	while (*text){
		char znak = (*text);
		FT22xout( znak );
		text++;
	}
	SPI_FT22x_SSh();


	return( true );
}


//-------------------------------------------------------------------------//
uint16_t GetFT22x()
{
	uint8_t znak, st;


	SPI_FT22x_SSl();
	FT22xout( cmdFT22x_GetChar );		// Read request command (CMD=0x01 0x0001)
	if ( (st=RD_SO_FT22x()) )	{
		SPI_FT22x_SSh();
		return( UINT16_MAX );
	}
	znak = FT22xin();
	SPI_FT22x_SSh();


	return( znak );
}


//====================== Pętla główna ==================================
	while (1)
	{
		HAL_IWDG_Refresh(&hiwdg);
		__WFI();


		if( ! timSendUsb ){
			timSendUsb = 2000;

			char static txt[100];
			uint16_t static cnt;
			sprintf( txt, "Ramka %d\n\r", ++cnt );
			PrintStringFT22x( txt );
```

```
}


{
        uint16_t znak;
        while( (znak=GetFT22x()) != UINT16_MAX ){
                PutCharFT22x( znak );
        }

}
```

</listing 1a>

Efekt działania programu można zaobserwować w terminalu **rysunek 14a**. **Listing 1b** przedstawia program dla Arduino.

<listing 1b>

```
#define UINT16_MAX  0xFFFF
#include <SPI.h>


#define SS_FT22X  10
#define SO_FT22X  9

//=========================================================================//
#define cmdFT22x_WrChar   0b00000001    // Write request command (CMD=0x00)
#define cmdFT22x_GetChar  0b00000011    // Read request command (CMD=0x01 0x0001)
#define cmdFT22x_StModem  0b00000101    // Read modem status (CMD=0x02 0b0010)
#define cmdFT22x_WrModem  0b00000111    // write modem status (CMD=0x03 0x0011)
#define cmdFT22x_Flush    0b00010001    // write buffer flush (CMD=0x04 0x0100)
#define cmdFT22x_AdrMtp   0b00010011    // Adres MTP memory (CMD=0x05 0b0101)
#define cmdFT22x_WriteMTP 0b00010101    // write eeprom (CMD=0x06 0b0110)
#define cmdFT22x_ReadMTP  0b00010111    // READ MTP memory (CMD=0x07 0b0111)
#define cmdFT22x_RDstatus 0b10000001    // 0x08


//-------------------------------------------------------------------------//
void SPI_FT22x_SSl() {
  digitalWrite( SS_FT22X, LOW );
}


//-------------------------------------------------------------------------//
void SPI_FT22x_SSh() {
  digitalWrite( SS_FT22X, HIGH );
}


//-------------------------------------------------------------------------//
void FT22xout( uint8_t TXdana ) {
  SPI.transfer( TXdana );
}


//-------------------------------------------------------------------------//
uint8_t FT22xin() {
  return SPI.transfer( 0xFF );
}


//-------------------------------------------------------------------------//
uint8_t RD_SO_FT22x() {
  return digitalRead( SO_FT22X );
}


//-------------------------------------------------------------------------//
uint8_t PutCharFT22x( char znak )
{
  uint8_t st;


  SPI_FT22x_SSl();
  FT22xout( cmdFT22x_WrChar );    // Write request command (CMD=0x00)
  delayMicroseconds( 6 );
```

```
  st = RD_SO_FT22x();
  FT22xout( znak );
  SPI_FT22x_SSh();


  return ( st );
}


//-------------------------------------------------------------------------//
uint8_t PrintStringFT22x( char *text )
{
  char st;


  SPI_FT22x_SSl();
  //todo: po uaktywnieniu linii SS na linii MIOSIO pojawia sie informacja o przepełnieniu bufora TX
  FT22xout( cmdFT22x_WrChar );// Write request command (CMD=0x00)
  st = RD_SO_FT22x();
  if ( st ) {
    SPI_FT22x_SSh();
    return ( false );
  }
  while (*text) {         // Wyswietl tekst z RAMu
    char znak = (*text);
    FT22xout( znak );
    text++;
  }
  SPI_FT22x_SSh();


  return ( true );
}


//-------------------------------------------------------------------------//
uint16_t GetFT22x()
{
  uint8_t znak, st;


  SPI_FT22x_SSl();
  FT22xout( cmdFT22x_GetChar );   // Read request command (CMD=0x01 0x0001)
  delayMicroseconds( 6 );
  if ( (st = RD_SO_FT22x()) ) {
    SPI_FT22x_SSh();
    return ( UINT16_MAX );
  }
  znak = FT22xin();
  SPI_FT22x_SSh();


  return ( znak );
}


//=========================================================================//
void setup() {
  pinMode( SS_FT22X, OUTPUT);
  pinMode( SO_FT22X, INPUT_PULLUP );

  // initialize SPI:
  SPI.begin();
  SPI.setDataMode( SPI_MODE1 );
  SPI.setClockDivider( SPI_CLOCK_DIV16 );


  Serial.begin( 115200 );
}


//=========================================================================//
void loop() {
  char txt[100];
```

```
  uint32_t static tim;
  if ( millis() > tim ) {
    tim = millis() + 2000;

    char static txt[100];
    uint16_t static cnt;
    sprintf( txt, "Ramka %d\n\r", ++cnt );
    PrintStringFT22x( txt );
  }


  uint16_t znak;
  while ( (znak = GetFT22x()) != UINT16_MAX ) {
    PutCharFT22x( znak );
      Serial.print( (char)znak );
  }

}
```

</listing 1b>

W przypadku Arduino teksty przysyłane terminalem są wyświetlane także w serial monitorze **Rysunek 14b**. Pełne listingi znajdują się w materiałach dodatkowych na Elportalu podobnie jak i zrzuty ekranowe, które w czasopiśmie, ze względu na małe wymiary, nie zawsze są czytelne.
Z układu FT22x można odczytać status status układu.

<listing 2>

```
uint8_t ReadStFT22x() {
        SPI_FT22x_SSl();
        FT22xout( cmdFT22x_RDstatus );          // Read USB Status (CMD=0x08 0b1000)
        StFT22x = FT22xin();
        SPI_FT22x_SSh();
        return( StFT22x );
}
```

</listing 2>

Ze względu na to, że funkcje „niskopoziomowe" dla STM32 i Arduino zostały już pokazane w listingach 1 a i b, od teraz funkcje są wspólne dla obu platform. O takich dogodnościach mogą tylko pomarzyć programiści piszący w Basic na przykład Bascom

<ramka>

Bascom istnieje tylko na zapomniany już 8051, AVR. Twórca Bascom nie zauważył, że „świat„ (Windows 10 jest „cudowny". Nie wiem czemu ale dużej litery „ś" nie mogę uzyskać) poszedł naprzód i istnieją dziesiątki czy setki razy lepsze mikrokontrolery niż AVR przy okazji tańsze a bardzo często dużo tańsze. Brak wsparcia Bascom dla ARM oznacza rychłą „śmierć" tego języka o ile już praktycznie nie „umarł". Nie ma co ronić łez, Basic jest słusznie uznawany za najgorszy popularny język programowania. Swoją drogą ciekawe, że twórcą/współtwórcą najgorszego ale i o dziwo najpopularniejszego języka, DOS-a i systemu okienkowego jest ta sama osoba.

<ramka>

To są zalety języka C/C++. Czasem, ze względu na różnicę w działaniu kompilatorów, uruchomienie programu na innej platformie, najczęściej przeniesionego z „lepszej" na „gorszą", wymaga niewielkich zmian w kodzie, często związanych z rozmiarem zmiennej. Gdy przyjrzeć się funkcjom obsługi FT22x różnice dotyczą tylko funkcji:

```
SPI_FT22x_SSl();
SPI_FT22x_SSh();
FT22xout();
FT22xin();
RD_SO_FT22x();
```

Po wykazaniu wyższości C na Basic przejdźmy do obsługi FT22x. Funkcja `ReadStFT22x() zwraca` liczbę z zakresu 0...3 a oznacza ona stan:
0x00 Suspended
0x01 Default
0x02 Addressed

0x03 Configured
Jeśli układ został skonfigurowany przez HOST, i możliwa jest komunikacja przez USB, FT22x zwraca 3. Praktyczne zastosowanie funkcji dla STM32:
<listing 3>
```
char static txt[100];
uint16_t static cnt;
sprintf( txt, "Ramka %d status %d\n\r", ++cnt, ReadStFT22x() );
PrintStringFT22x( txt );
```
</listing 3>
Efekt działania programu widać na **rysunku 15**. FT22x potrafi ustawić stan linii modemowych CTS, DSR, CDC i RI.
<listing 4>
```
//---------------------------------------------------------------------------//
//      Bit=1 OFF, =0-ON
//      bit     7       CD
//      bit 6   RI
//      bit 5   DSR
//      bit 4   CTS
uint8_t SetModemFT22x( uint8_t modem )
{
        uint8_t st;


        SPI_FT22x_SSl();
        FT22xout( cmdFT22x_WrModem );
        st = RD_SO_FT22x();
        FT22xout( modem );
        SPI_FT22x_SSh();


        return( st );
}
```
</listing 4>
Jeżeli potrafi ustawić to i potrafi odczytać ustawienie linii RTS, DTR:
<listing 5>
```
uint8_t ReadModemFT22x(){
        uint8_t st=-1;


        SPI_FT22x_SSl();
        FT22xout( cmdFT22x_StModem );    // Read modem status (CMD=0x02 0b0010)
        st = FT22xin();                  // bit 0 - DTR, bit 1-RTS
        SPI_FT22x_SSh();


        return( st );
}
```
</listing 5>
Praktyczne zastosowanie:
<listing 6>
```
uint8_t static modem, prevMod;
modem = ReadModemFT22x();
SetModemFT22x( modem << 4 );

if( prevMod != modem ){
        prevMod = modem;

        char static txt[100];
        sprintf( txt, CRLF"Zmiana linii lodemu. DTR=%x RTS=%x"CRLF, (modem & 1), ((modem & 2) >> 1) );
        PrintStringFT22x( txt );
```
</listing 6>
W efekcie zmianie stanu linii RTS towarzyszy zmiana DRS i wyświetlenie komunikatu w oknie terminala **rysunek 16**, podobnie zmianie DTR towarzyszy zmiana CTS i komunikat **rysunek 17**. Zapraszam do analizy danych z LA2016 w materiałach dodatkowych.

Na koniec pozostawiłem najciekawszą możliwość układów FT22x, konfigurowanie ustawień 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 wygrywać programu do urządzenia dwa razy, raz programu dla mikrokontrolera za drugim razem konfiguracji dla FTDI.
- Kolejna zaleta to ewentualne 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 FT22x problemu nie ma.
- Kolejny przykład, wymiana uszkodzonego mostka USB. W przypadku standardowych układów trzeba jeszcze wgrać konfigurację, w przypadku FT22x konfigurację może ustawić mikrokontroler.
- To, że mikrokontroler może sprawdzić konfigurację pozwalają zabezpieczyć się przed „grzebaniem" w niej przez osoby postronne.
- MTP i powiązaną 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 mikrokontrolereze 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.

Zanim opiszę sposób konfigurowania pamięci MTP zacznę od prostszego zagadnienia, pamięci EEPROM. Cała pamięć FT22x jest podzielana na kilka obszarów **rysunek 18**. Zielone obszary są do dowolnego wykorzystania przez użytkownika przy czym obszar 0x24...0x7F (słowa 0x12...0x3F) jest widoczny w oknie programu FT_PROG **rysunek 19**. Pozostałe obszary kontrolowane są 16-bitową sumą kontrolną. Jeśli będzie błędna układ przyjmie standardową konfigurację. Aby odczytać zawartość pamięci EEPROM należy ją najpierw zaadresować. 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 ReadMtpFT22x( 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:

<listing 7a>

```
uint8_t mtpFT22x[2048];
ReadMtpFT22x( 0, 2048, mtpFT22x );
```

</listing 7a>

W przypadku AVRmega328 zamontowanego w ArduinoUNO nie da się wczytać 2kB EEPROM z układu FT22x do mikrokontrolera bo posiada on 2048bajtów ram a gdzieś muszą zmieścić się stos, zmienne, bufory UART, dlatego w tym przypadku wczytane zostanie piewsze 256bajtów czyli cała pamięć MTP.

<listing 7b>

```
uint8_t mtpFT22x[256];
ReadMtpFT22x( 0, 256, mtpFT22x );
```

</listing 7b>

Użytkownicy ArduinoMega mogą bez problemu wczytać cały EEPROM bo Mega2560 posiada 8kB RAM.

Zapis bajtu jest równie prosty, z punktu widzenia funkcji „`uint16_t WriteMtpFT22x( uint16_t adres, uint8_t data )`". Na listingu prosty program licznika resetów:

<listing 8>

```
uint16_t static cntFT22x;
uint8_t const AdrCntFT22x = 0x18 << 1;
ReadMtpFT22x( AdrCntFT22x, 2, ((uint8_t*)(&cntFT22x)) );
cntFT22x++;
WriteMtpFT22x( AdrCntFT22x, cntFT22x );
WriteMtpFT22x( AdrCntFT22x+1, cntFT22x >> 8 );

char static txt[100];
sprintf( txt, "Licznik resetow %d"CRLF, cntFT22x );
PrintStringFT22x( txt );
```
</listing 8>

Efekt działania programu widać na **rysunku 20**.

<ramka>
Debuger to potężne narzędzie w rękach programisty. Niestety Arduino jest pozbawione debugera. Rozwiązaniem, w przypadku AVR jest AtmelStudio, które importuje projekty z Arduino i pozwala na debugowanie. Jeśli Czytelnicy są zainteresowani artykułem na ten temat proszę pisać do redakcji.
</ramka>

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 21**. 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:

<listing 9>
```
char txt[] = "sas@elportal.pl dla EdW";
uint8_t adr = 0x20 << 1;
for(uint16_t x=0; x<strlen(txt); x+=2){
	WriteMtpFT22x( adr++, txt[ x+1 ] );
	WriteMtpFT22x( adr++, txt[ x+0 ] );
}
```
</listing 9>

Tekst ten jawnie nie musi występować w kodzie programu, można go zaszyfrować kluczem o długości samego tekstu. Tekst jest widoczny w oknie FT_PROG (w tym przypadku specjalnie dałem go w obszarze 0x40...0x56) ale można go umieść tam, gdzie FT_PROG go nie pokaże czyli w obszarze x0100...0x7FF (słowa 0x80...0x3FF). Sygnaturę można odczytać na komputerze przez D2XX lub przez uC. W ten sposób można stworzyć całkiem dobre zabezpieczenie swoich praw autorskich (kto by się spodziewał danych w mostku USB?), MCP2221 czy mostki z UART nie dają takiej możliwości.

<ramka>
Obszar EEPROM jak i samej pamięci MTP może być odczytywany zarówno z poziomu mikrokontrolera jak i programu w komputerze. Daje do 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 więc nawet gdy jest uszkodzony ale kiedykolwiek skonfigurował FT22x dane można odczytać z niego. Przykładowo licznik czasu pracy w mikrokontrolere jest nie do odczytania a w FT22x nie ma z tym problemu. Można oczywiście taki licznik przechowywać z zewnętrznej pamięci EEPROM ale jej odczyt wymaga dodatkowych zabiegów, w przypadku FT22x wystarczy do tego komputer. Nawet nie trzeba pisać specjalnej aplikacji, wystarczy darmowy FT_PROG i wiedza gdzie i jak są zapisane informacje.
</ramka>

Teraz najtrudniejsze zagadnienie, obszar MTP.

Na początek warto zapoznać się z mapą pamięci MTP **rysunek 22**, w której układy FTDI przecho-

wują dane o swojej konfiguracji. 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 napiszę dalej, 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 SPI/I2C domyślnym ustawieniem jest nieładowanie sterowników VCP tylko bibliotek 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 menadżera urządzeń **rysunek 23** Zaznaczając opcję „załaduj VCP” lub lepiej z poziomu FT_PROG rysunek 4. Dlaczego sugeruję rozbić to z poziomu FT_PROG? Otóż jak zrobimy to w menadżerze urządzeń to po zmianie deskryptora i enumeracji, konieczne będzie powtórzenie operacji w „Menadżerze urządzeń”, jak zrobimy to w FT_PROG, to ta wątpliwa przyjemność ominie nas. Oczywiście lepszym rozwiązaniem będzie zmiana bitu odpowiedzialnego za ta funkcje z poziomu mikrokontrolera. Za ładowanie VCP odpowiedzialny jest siódmy bit (licząc od zera) pierwszego bajtu obszaru MTP **rysunek 24a** i **24b**. Należy pamiętać, że aby zobaczyć zmiany w dolnym oknie pokazującym zawartość pamięci MTP należy ją najpierw zapisać po czym odczytać, natomiast aby zmiany odniosły skutek konieczna jest enumeracja.

<ramka>
Efektem ubocznym zmian konfiguracji jest zmiana numeru wirtualnego portu COM w systemie Windows. Tego problemu nie ma w systemie Linux i Unix.
</ramka>

Aby ustawić bit odpowiedzialny za ładowanie sterowników VCOM można skorzystać z poniższego kodu:

</listing 10>
```
uint8_t mtp[256];
ReadMtpFT22x( 0, 256, mtp );
crc = CalculateMtpCrc( (word*)mtp ); crcL=crc;
if ( (crcL==mtp[0xFE]) && (crc>>8==mtp[0xFF]) )
       {
       //jeśli CRC ok to zapis
       WriteMtpFT22x( 0x24, data );
       //
       mtp[0] |= 0x80;           // Ustaw ładowanie VCP
       crc = CalculateMtpCrc( (word*)mtp );
       mtp[0xFE] = crc; mtp[0xFF] = crc >> 8;
       WriteMtpFT22x( 0, mtp[0] );
       WriteMtpFT22x( 0xFE, mtp[0xFE] );
       WriteMtpFT22x( 0xFF, mtp[0xFF] );
       }
```
</listing 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 na ciekawe zachowanie układu a właściwie sterownika dla Windows. Na czym to „ciekawe” zachowanie polega? Z poziomu FT_PROG nie można uzyskać pewnych opcji, na przykład „LedTX&RX” dla FT22x/200x **rysunek 25**. Może to jednak zrobić modyfikują MTP z poziomu mikrokontrolera **rysunek 26,** mimo

że opcji „LedTX&RX” nie ma na liście rozwijanej pojawiła się w okienku! Trzeba jednak być ostrożnym z takimi operacjami bo układ FT22x, 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.

<ramka>

MTP jest wczytywana do FT22x raz po resecie. Jeśli więc zostanie zmieniona jej zawartość (nie mylić obszaru MTP z obszarem EEPROM użytkownika) aby HOST widział zmiany, trzeba wymusić enumerację np. przez chwilowe odłączenie wtyczki USB. Gdy zmieniamy obszar MTP trzeba zapisać poprawną sumę kontrolą, która można wyliczyć funkcją `CalculateMtpCrc`. Modyfikując pozostały obszar EEPROM nie zapisujemy CRC ponieważ jest on wyłączony z kontroli przez CRC.

</ramka>

Bajty 0x0E i 0x0F (słowo 0x07) określają nazwę producenta **rysunek 27**. Zawierają wskaźnik 0xA0, długość 0x0A. Bajt 0xA0 / 2 = słowo 0x50. Długość 0x0A = 10. 10 / 2 = 5 słów. Podglądamy zawartość pamięci **rysunek 28**. Coś nie do końca się zgadza. Tekst ma długość czterech a nie pięciu znaków i zaczyna się od jakiś „tajemniczych” 0x030A. Sprawdzamy więc nazwę produktu. Bajty 0x10 i 0x11 (słowo 0x08) zawierają adres nazwy deskryptora i jego długość. Na co wskazują bajty? **Rysunek 29** pokazuje wskaźnik 0xAA, długość 0x28, 0xAA / 2 = 0x55. 0X28 = 40. 40 / 2 = 20 słów. Sprawdzamy. **Rysunek 30**. Znów niezgodność długości o jeden znak i pierwsze słowo o wartości 0x0328, która zdaje się coś sugerować. Sprawdzimy więc jeszcze numer seryjny. Bajty 0x12 i 0x13 zawierają wskaźnik 0xD2 i długość 0x12 **rysunek 31**. 0xD2 / 2 = 0x69, 0x12 = 18 / 2 = 9 słów. Numer seryjny zawiera 8 znaków i „tajemniczy” ciąg 0x0312 **rysunek 32**. Łatwo wywnioskować, że pierwsze słowo zawiera w starszym bajcie 0x03 w młodszym długość ciągu znaków w bajtach. Wszystko wskazuje na to, że można pisać „między wierszami”, bo w słowie zawierającym znaki wykorzystane jest tylko jeden, młodszy bajt. Starszy jest to wykorzystania ale nie sprawdzałem takiej możliwości.

<ramka>

Dwa bajty na znak dla 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 tego, że w układach FTDI użyto mikrokontrolerów 16-bit. ***Na niektóre układy FTDI można pisać własne programy a producent udostępnia nieodpłatnie IDE i dokumentację.*** Takimi układami są na przykład VNC-2, które posiadają dwa USB, które mogą pracować w trybie HOST/DEVICE. Nie znam tańszego układu niż VNC, który posiadałby dwa HOST-y USB. Jeśli Czytelnicy wykażą zainteresowanie programowaniem układów VNC na łamach EdW pojawi się stosowny kurs.

</ramka>

Suma kontrolna MTP znajduje się w bajtach 0xFE i 0xFF (słowo 0x7F) **rysunek 33**. Sumę kontrolną oblicza funkcja:

<listing 11>

```
uint16_t CalculateMtpCrc( uint16_t *buf )
{
uint16_t Checksum = 0xAAAA;              // Variable for checksum value
uint8_t fl15, AddressCounter = 0x00;     // Variable for address counter
uint16_t TempChecksum = 0x0000;          // Used whilst calculating checksum
uint8_t CheckSumLocation = 0x7F;         // Address at which checksum stored in FT-X


        // Starting at Word address 0x00
        AddressCounter = 0x00;

        // Calculation uses addresses from 0x00 up to 0x7E (checksum itself is located at 0x7F)
        while(AddressCounter < CheckSumLocation)
                {
        // EXOR the data with the current checksum and then rotate one bit to the left
                TempChecksum = *(buf+AddressCounter) ^ Checksum;
                //Checksum = (word)((TempChecksum << 1) | (TempChecksum >> 15));
```

```
                if (TempChecksum&0x8000) fl15=true; else fl15=false;
                TempChecksum <<= 1; if (fl15) TempChecksum |= 1;
                Checksum = TempChecksum;

        // Go to next word address.
        // If we have reached word address 0x12, then skip forward to address 0x40
        AddressCounter ++;
        if(AddressCounter == 0x12) AddressCounter = 0x40;
        }


        return Checksum;
}
```

</listing 11>

Sposób jej użycia można znaleźć na listingu 10. Aby zaobserwować zmiany sumy można zmodyfikować na przykład deskryptor.

Opisywanie wszystkich bajtów konfiguracji nie ma większego sensu, zainteresowanych odsyłam do dokumentu „AN_201_FT-X MTP Memory Configuration”.

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 następnie taką konfiguracje 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 zapisać MTP w FTDI. Trzeba pamiętać aby odczyt MTP przeprowadzać gdy układ jest skonfigurowany (funkcja „ReadStFT22x()” zwróci 3).

<ramka>

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 zapisie.

</ramka>

Programy przedstawione w artykule są demonstracjami pokazującymi jak komunikować się z układami FT22x. Nie posiadają zabezpieczeń, nie zrealizowano timeout więc w razie problemów program zresetuje się (zadziała watchdog).

Firma FTDI produkuje układ FT221. Z punktu widzenia oprogramowania w trybach SPI 1...4 bit nie ma między nimi różnic. Zaletą FT221 jest większa prędkość komunikacji USB wynosząca do 1Mb/s w stosunku do 500kb/s dla FT220 i możliwość pracy w trybie SPI 8-bit w stosunku do 4-bit dla FT220. Wadą FT221 jest większa liczba wyprowadzeń a co za tym idzie obudowa. Jeśli Czytelnicy będą zainteresowani płytką z układem FT221 powstanie taki projekt. W tej sprawie proszę pisać do redakcji EdW.

SaS
sas@elportal.pl

Wykaz elementów:

Rezystory 1206

R1 R2	27R
R3 R4 R5	1k

Kondensatory

C1	10uF	Kondensator Elektrolityczny
C2 C5 C6 C7	100nF	Ceramiczny 1206
C3 C4	47pF	Ceramiczny 1206

Półprzewodniki

U1	FT220XS
U2	74LVC125AD
U4	ADUM1401ARW
U5	LTV357T

Inne

J2	IDC10MLP