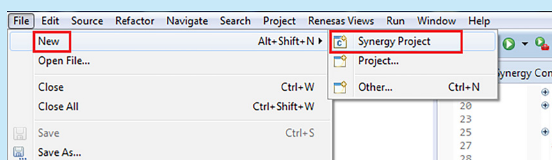
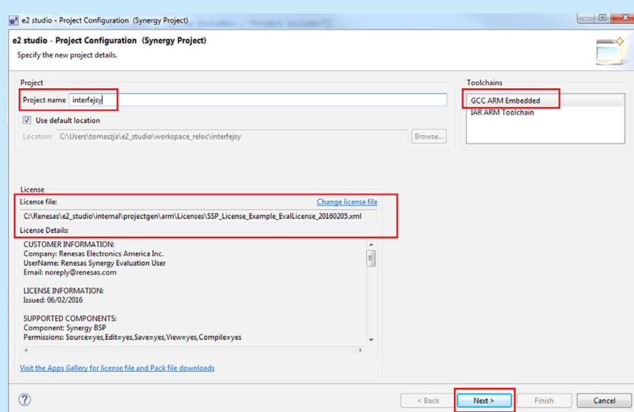
Renesas Synergy – interfejsy szeregowe (1)

Interfejsy szeregowe są bardzo ważnym elementem budowania systemów mikroprocesorowych. Wiele elementów zewnętrznych, takich jak wyświetlacze, czujniki, moduły komunikacyjne i inne mają wbudowane szeregowe interfejsy komunikacyjne z jednej strony, a mikrokontrolery układy peryferyjne obsługujące transmisję szeregową z drugiej strony. Moduły komunikacyjne mikrokontrolerów są często bardzo rozbudowane. Przychodzące i wysyłane dane mogą być przesyłane kanałami DMA lub są buforowane w FIFO. Bardziej rozbudowane interfejsy, na przykład I²C, mogą pracować jako master w magistrali z wieloma masterami. Konfigurowanie tego typu peryferii jest prawdziwą udręką dla programistów. Konieczność zapisania wielu rejestrów konfiguracyjnych i wzajemne czasami skomplikowane zależności pomiędzy bitami konfiguracyjnymi powodują, że łatwo się pomylić i bardzo trudno znaleźć przyczynę pomyłki. Żeby ułatwić i przyspieszyć konfigurację, stosuje się dwa wzajemnie się uzupełniające elementy. Pierwszy z nich to najczęściej graficzny konfigurator, a drugi to gotowe biblioteki warstwy HAL.

Warstwa HAL – Hardware Abstraction Layer, to warstwa biblioteki zawierająca funkcje, które są niezależne od rozwiązań sprzętowych. Użytkownik używający tych funkcji nie musi znać żadnych rejestrów konfiguracyjnych i szczegółów sprzętowych interfejsu. Funkcje warstwy HAL wykorzystują funkcje warstwy niższej zależnej od sprzętu i różnych dla różnych typów mikrokontrolerów. Żeby takie rozwiązanie mogło poprawnie działać, trzeba bibliotekę skonfigurować. Ręczna konfiguracja wymagałaby znajomości sprzętu, a to zniweczyłoby korzyści ze stosowania warstwy HAL. Dlatego stosuje się różnego rodzaju graficzne konfiguratory. Takie oprogramowanie na podstawie znajomości wybranego typu mikrokontrolera i danych wprowadzonych przez użytkownika generuje struktury konfiguracyjne, z których korzystają funkcje biblioteki. Renesas oferuje środowisko projektowe e2studio zawierające graficzny konfigurator projektu. Jednym z możliwości tego konfiguratora jest szybka i bezproblemowa konfiguracja układów peryferyjnych. W wyniku pracy konfiguratora dostajemy projekt używający gotowych firmowych bibliotek SSP, w tym funkcji używających warstwy HAL.



Rysunek 1. Nowy projekt



Rysunek 2. Konfiguracja projektu

W tym artykule pokażę, jak:

- Utworzyć nowy projekt w środowisku e2studio. Projekt będzie przeznaczony dla mikrokontrolera serii S1 R7F-S124773A01CFM zamontowanego w module Arrow Aris EDGE (kompatybilnym sprzętowo z systemem ARDUINO).
- Skonfigurować driver interfejsu SCI do pracy w trybie SPI, wykorzystując konfigurator Synergy Configurator wbudowany w e2studio.
- Obsłużyć mały wyświetlacz OLED ze sterownikiem SSD1306 wykorzystując funkcje drivera r_sci_spi warstwy HAL biblioteki SSP.
- Skonfigurować dwa drivery interfeju I²C, wykorzystując konfigurator Synergy Configurator wbudowany w e2studio.
- Obsłużyć dwa czujniki: temperatury i wilgotności, wykorzystując funkcje drivera r_riic warstwy HAL biblioteki SSP.
- Zdefiniować i użyć funkcji callback przeznaczonych do powiadamiania między innymi o zakończeniu transmisji w interfejsach szeregowych.

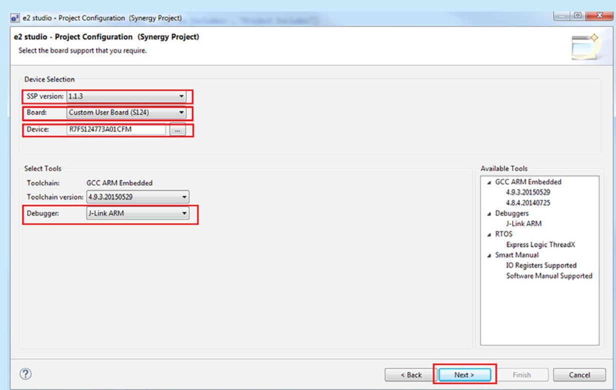
Projekt

Użycie modułu Arrow Aris EDGE z mikrokontrolerem serii S1 R7F-S124773A01CFM skutkuje brakiem wsparcia BSP (Board Support Package). Takie wsparcie mają tylko firmowe moduły ewaluacyjne. Jednak nie jest to żaden problem, bo odpowiednie konfiguracje wykonamy samodzielnie. Tworzenie projektu zaczynamy z menu File→New→Synergy Project (**rysunek 1**). W kolejnym kroku nadajemy naszemu projektowi w oknie Project name nazwę „interfejsy", wybieramy bezpłatny kompilator GCCARM Embedded i podajemy ścieżkę dostępu do pliku licencji kompilatora, jeżeli nie była do tej pory podana. Pokazano to na **rysunku 2**. Ostatnie okno konfiguracji projektu zawiera okna wyboru:

- Wersji biblioteki SSP.
- Płytki ewaluacyjnej.
- Typu mikrokontrolera.
- Typu debuggera/programatora.

Wybieramy najnowszą dostępną w czasie pisania artykułu wersję SSP o numerze 1.1.3. Ponieważ płytka nie jest wspierana (przynajmniej w momencie pisania tego tekstu), w oknie Board wybieramy **Custom User Board (S124)**. Typ mikrokontrolera płytki ewaluacyjnej wybieramy w oknie Device (**rysunek 3**).

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 92822, pass: 37euo8qf



Rysunek 3. Końcowy etap konfiguracji projektu

Po przejściu wszystkich etapów wstępnej konfiguracji wykonywanej w trakcie tworzenia nowego projektu konfigurator generuje szkielet projektu gotowy do skompilowania. Oczywiście taki projekt niczego nie robi i trzeba go uzupełnić o funkcje obsługi interfejsów komunikacyjnych warstwy HAL i ewentualnie o funkcje warstw wyższych.

Interfejs SPI

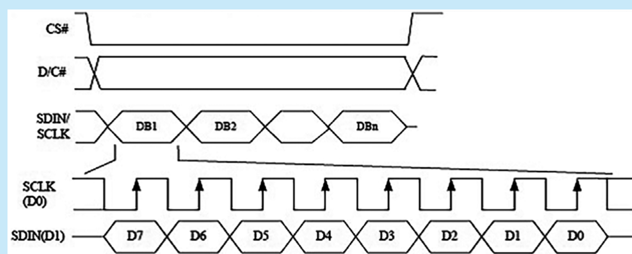
Konfigurowanie interfejsu dla samej konfiguracji nie ma większego sensu. Dlatego pokażę jak skonfigurować interfejs SPI sterujący pracą wyświetlacza OLED z wbudowanym sterownikiem SSD1306.

SSD1306 ma możliwość komunikacji za pomocą magistrali równoległej w standardzie Intel8080 lub Motorola 6800 oraz za pomocą magistrali szeregowej w standardzie SPI lub I²C. Dla aplikacji, w których jest ważna prędkość przesyłanych danych są wybierane magistrale równoległe. W pozostałych wypadkach – magistrale szeregowe. Dla typowych aplikacji z interfejsem użytkowania wystarczająca jest prędkość dostępna za pomocą interfejsu szeregowego. Podstawowe właściwości sterownika umieszczono w **tabeli 1**.

W tym typie wyświetlacza na etapie produkcji wybrano 4-przewodowy interfejs SPI. Ponieważ nie przewidziano odczytywania jakichkolwiek danych ze sterownika, to SPI ma tylko linię danych DIN (MOSI z punktu widzenia hosta) oraz linię zegarową CLK. Oprócz tych linii do przesyłania danych są używane linie CS i D/C. Sygnał CS jest standardowym sygnałem SPI uaktywniającym układ, do którego są wysyłane dane. Używając CS

Tabela 1. Podstawowe właściwości sterownika SSD1306

Rozdzielczość	Matryca 128×64 piksele
Napięcie zasilania	Układy logiczne od +1,65 V do +3,3 V
	Drivery panelu matrycy od +7 V do +15 V
Maksymalny prąd segmentu	100 μA
Maksymalny całkowity prąd matrycy	15 mA
Kontrast matrycy	256 poziomów (programowany)
Wbudowana pamięć obrazu	SRAM 128×64 bity
Interfejsy komunikacyjne	Równoległy 8-bitowy 8080/6800
	Szeregowy SPI 3-, lub 4-liniowy
	Szeregowy I²C
Funkcje akceleratora graficznego	Ciągłe skrolowanie w poziomie
	Ciągłe skrolowanie w pionie
Remapowanie	Kolumn i wierszy
Oscylator	Wbudowany w układ
Zakres temperatury pracy	-40...+85°C



Rysunek 4. Przebiegi transferu danych na magistrali SPI sterownika SSD1306

można do jednej magistrali dołączyć kilka wyświetlaczy – każdy ze swoją linią CS. Linia D/C jest wykorzystywana do kierowania danych albo do rejestru komend (D/C=0), albo do pamięci obrazu (D/C=1). Na **rysunku 4** pokazano przebiegi czasowe na magistrali SPI w trakcie transferu danych.

Organizacja pamięci obrazu, komendy sterujące, inicjalizacja sterownika itp. jest dokładnie opisana w dokumentacji sterownika i wyświetlacza. Sam moduł wyświetlacza był również opisywany na łamach Elektroniki Praktycznej i nie ma powodów by je tutaj powielać. Zajmiemy się głównym tematem naszych rozważań, czyli interfejsem SPI i jego konfiguracją.

Mikrokontrolery Synergy mają wbudowane dwa typy interfejsów szeregowych. Pierwszy z nich to interfejs SCI – Serial Communications Interface. Jest to uproszczony uniwersalny interfejs, który można zaprogramować do pracy w trybie SPI, I²C lub UART Drugi typ to natywne interfejsy SPI, I²C i UART. Pracują tylko ze swoją magistralą, ale mają większe możliwości konfiguracji. Z punktu widzenia sterowania wyświetlacza nie ma większego znaczenia, który interfejs wybierzemy, bo SSP obsługuje oba typy. Wyznaję zasadę, że najlepiej jest zaczynać od rzeczy prostych i dlatego na początek wybierzemy interfejs SCI pracujący jako SPI.

Mikrokontroler ma wbudowane dwa moduły SCI0 i SCI1. Schemat blokowy SCI pokazano na **rysunku 5**. Wbudowane rejestry bufora FIFO pozwalają na pracę w trybie full-duplex i bezprzerwowe wysyłanie danych. Ważnym elementem jest układ BRG – Baud Rate Generator ustalający prędkość wysyłania i odbierania danych niezależnie dla każdego z modułów SCI.

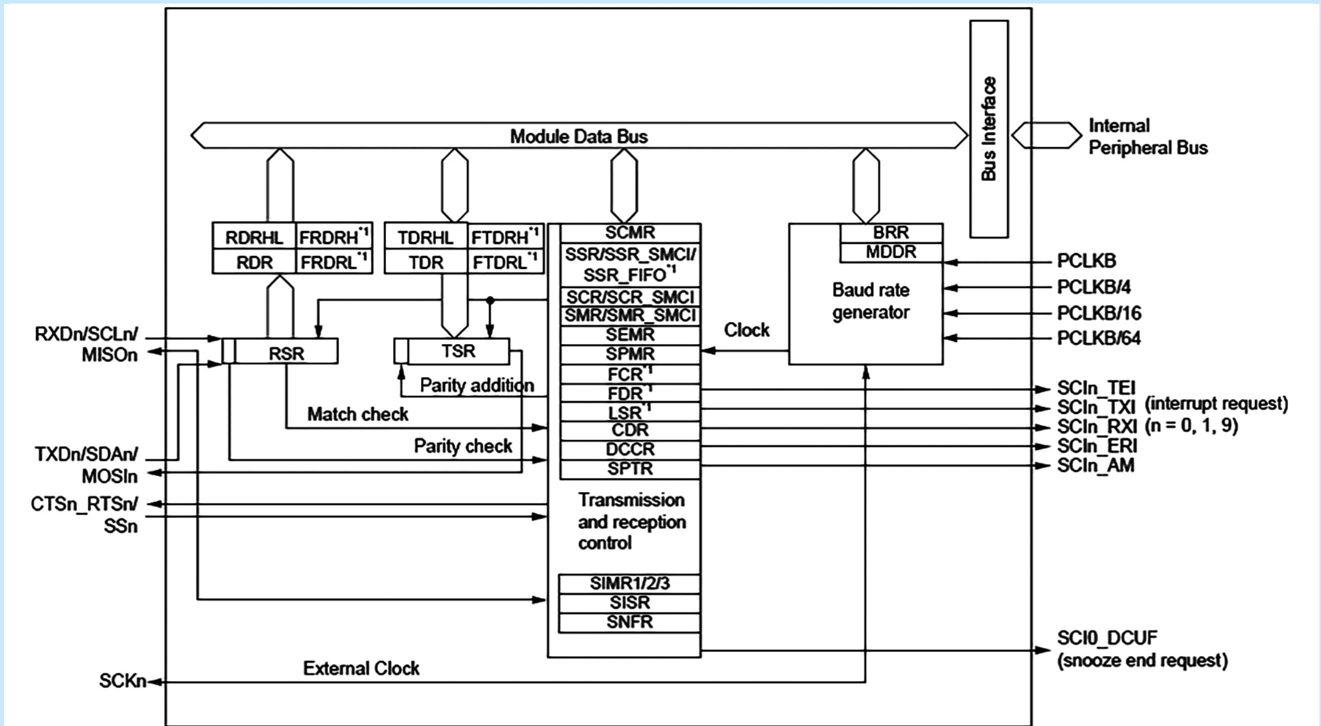
Konfiguracja interfejsu SCI w trybie SPI

Konfiguracja interfejsu z poziomu konfiguratora środowiska projektowego e2studio pozwala na ustawienie parametrów pracy interfejsu: wyboru kanału (numeru) interfejsu, trybu SPI, prędkości transmisji, fazy i polaryzacji zegara taktującego. Dodatkowo konfiguruje się układ przerwań niezbędny do działania drivera SPI. Konfigurator po zatwierdzeniu zmian dodaje do szkieletu projektu konieczne pliki źródłowe ze strukturami konfiguracji.

Konfigurowanie interfejsu zaczynamy od zakładki Clocks – **rysunek 6**. Przyjąłem, że mikrokontroler będzie taktowany maksymalna częstotliwością 24 MHz. Jeżeli z jakichś powodów, na przykład przy optymalizowaniu pobory mocy trzeba będzie zmniejszyć częstotliwość taktowania, to najwygodniej jest to zrobić w zakładce Clocks.

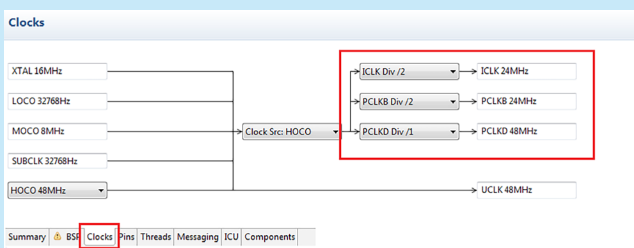
Po ustawieniu taktowania przechodzimy do zasadniczej fazy naszego projektu: wyboru interfejsu SCI i drivera z warstwy HAL. W zakładce Threads wybieramy HAL/Common i dodajemy driver klikając na ikonkę „plusa" w prawym górnym rogu okna HAL/Common Stacks. Z rozwijanego menu wybieramy: Driver → Connectivity → SPI Driver on r_sci_spi, jak pokazano na **rysunku 7**. Jak już wiemy, SCI może pracować w trybie SPI, UART lub I²C. Sposób wyboru interfejsu pokazano na **rysunku 8**. Dodany driver wymaga kolejnego konfigurowania wykonywanego w oknie Properties, jak pokazano na **rysunku 9**. Zależnie od potrzeb ustawiamy:

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 92822, pass: 37eu08qf

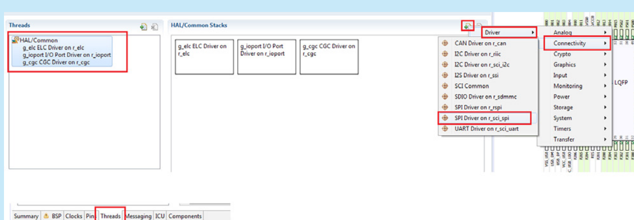


Rysunek 5. Schemat blokowy interfejsu SCI

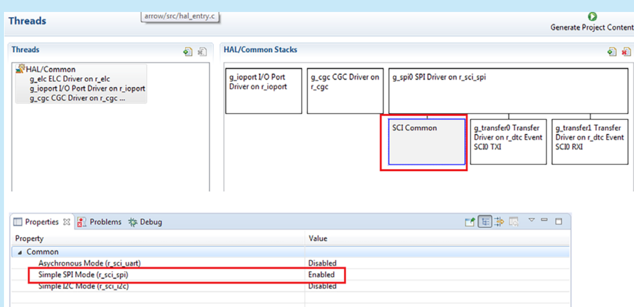
- Włączenie i priorytety obsługi przerwań od zdarzeń zgłaszanych przez driver. To ustawienie jest konieczne dla prawidłowego działania drivera.
- Nazwę modułu i numer kanału SCI – wybieramy kanał zerowy.
- Tryb pracy modułu Master/Slave – wybieramy tryb Master.
- Polaryzację i fazę zegara taktującego transmisję – te ustawienia muszą się pokrywać z trybami pracy układu slave, w naszym wypadku – z trybem pracy magistrali SPI sterownika wyświetlacza.
- Kolejność wysyłania bitów w słowie – pierwszy najstarszy.
- Prędkość wysyłania bitów 100 kb/s.
- Nazwę funkcji callback – zostanie dokładnie opisana dalej.



Rysunek 6. Zakładka ustawień zegara taktującego



Rysunek 7. Dodanie drivera SPI interfejsu SCI



Rysunek 8. Wybór magistrali SPI interfejsu SCI

Na tym etapie kończymy konfigurowanie drivera SPI warstwy HAL i przechodzimy do warstwy niższej związanej ze sprzętem. Musimy zaprogramować odpowiednie linie portu przez przypisanie ich do linii interfejsu SCI. Wykorzystujemy w tym celu zakładkę Pins. W oknie Pins Selection wybieramy Pripherials SCI0_2_4_6_8. W oknie Pin Configuration przypisujemy liniom interfejsu linie portu P1 – rysunek 10. Skonfigurowane linie P100, P101, P102 i P103 można zobaczyć na rysunku obudowy mikrokontrolera umieszczonego w oknie Package. Każda z przypisanych linii musi być jeszcze skonfigurowana przez określnie głównie kierunku przepływu danych: wejście lub wyjście, jak na **rysunku 11**. Po wykonaniu wszystkich czynności konfiguracyjnych klikamy na przycisk **Generate Project Content** i konfigurator umieści w projekcie wszystkie pliki źródłowe i konfiguracyjne. Teraz musimy napisać sobie procedury warstwy aplikacji.

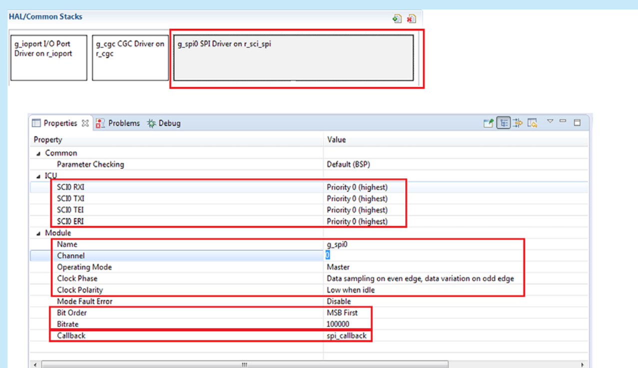
Obsługa interfejsu SPI jest wykonywana przez funkcje drivera SCI SPI R_SCI_SPI. Za ich pomocą użytkownik może zainicjować driver i wykonywać operacje transferu danych poprzez fizyczną magistralę SPI. Wszystkie parametry konfiguracji ustawiane na rysu. 10 są umieszczane w strukturze **spi_cfg_t** generowanej automatycznie przez konfigurator. Druga strukturą używana przed driver jest **spi_ctr_t**. Zawiera ona informacje o tym czy kanał komunikacyjny został otwarty, numer używanego kanału itp.

Jako pierwszą opiszemy funkcję otwarcia kanału drivera R-SCI_SSPI_Open.

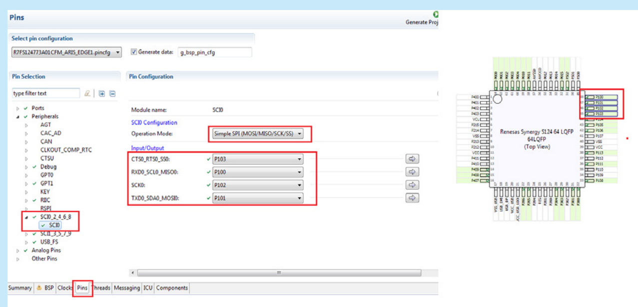
```
ssp_err_t R_SCI_SSPI_Open (spi_ctrl_t *p_ctrl, spi_cfg_t const
          *const p_cfg)
```

Jej argumentami są wspomniane struktury spi_ctrl_t i spi_cfg_t wygenerowane przez konfigurator. Funkcja sprawdza poprawność parametrów i ewentualnie generuje informacje o błędach, włącza zasilanie kanału SPI, blokuje przerwania i na podstawie wprowadzonych ustawień inicjalizuje rejestry konfiguracyjne SCI. Jeśli używamy więcej niż jednego drivera SPI pracującego

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 92822, pass: 37euo8qf



Rysunek 9. Konfiguracja drivera SPI interfejsu SCI



Rysunek 10. Przypisanie linii portu do interfejsu SCI

z tą sama magistralą funkcja *Open* w połączeniu z funkcja *Close* pozwala na sterowanie dostępem driverów do magistrali. Wywołanie funkcji otwarcia kanału drivera będzie wyglądało następująco: *err=R_SCI_SPI_Open(g_spi0.p_ctrl, g_spi0.p_cfg);*. Wszystkie funkcje drivera zwracają informacje o statusie wykonania funkcji *ssp_err_t*. Status *SSP_SUCCESS* oznacza prawidłowe wykonanie funkcji. Każda inna wartość oznacza jakiś błąd.

Z funkcją otwarcia kanału jest skojarzona funkcja zamknięcia kanału

```
ssp_err_t     R_SCI_SSPI_Close (spi_ctrl_t *const p_ctrl)
```

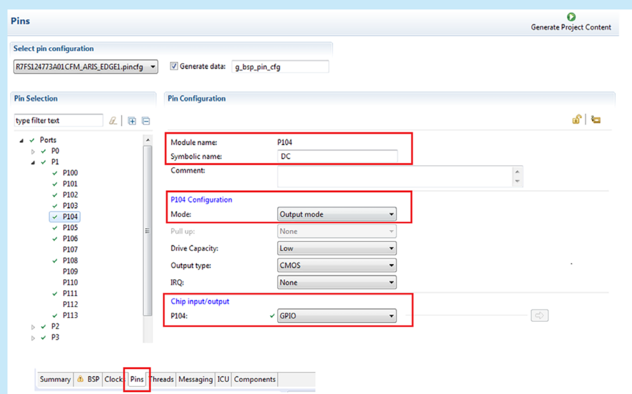
Funkcja zamyka otwarty kanał, wyłącza jego zasilanie, blokuje przerwania skojarzone z obsługą drivera i odświeża status kanału. Jeżeli używamy jednego kanału SPI o stałych parametrach transmisji i nie używamy systemu RTOS, to nie ma potrzeby zamykania kanału SPI funkcją Close. W przypadku używania RTOS może się okazać, że różne watki będą rywalizować o jedna magistrale SPI. Obie opisywane funkcje pozwalają na zarządzanie zasobem, którym jest dostęp do magistrali SPI.

Do sterowania transferem danych na magistrali są przeznaczone 3 funkcje:

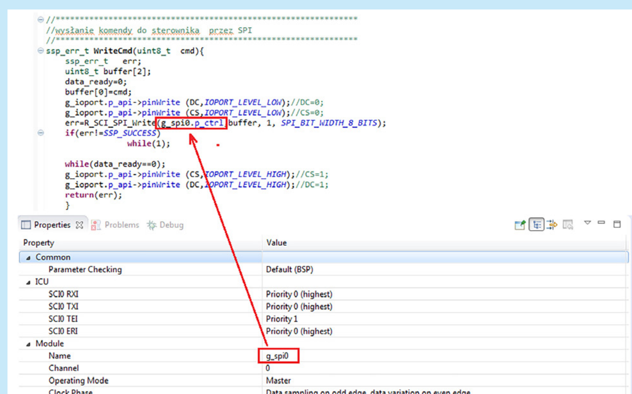
- **R_SCI_SPI_Write** – zapisywanie danych przez interfejs SPI.
- **R_SCI_SPI_Read** – odczytywanie danych za pomocą SPI.
- **R_SCI_SPI_WriteRead** – jednoczesne zapisywanie i odbieranie danych.

Dala nas jest najważniejsza funkcja zapisu danych na magistralę. Umożliwia ona zapisanie określonej liczby danych z bufora pamięci RAM. Jej działanie jest wykonywane w kilku krokach:

- Sprawdzenie poprawności parametrów i wygenerowanie ewentualnych informacji o błędach.
- Zablokowanie przerwań.
- Ustawienie długości przesyłanego słowa w bitach.
- Odblokowanie nadajnika danych.
- Odblokowanie przerwań.
- Rozpoczęcie transmisji danych po wykryciu przerwania zgłaszanego przez pusty bufor danych.
- Przekopiowanie danych z bufora źródłowego do rejestru danych modułu SPI.



Rysunek 11. Konfiguracja linii portów



Rysunek 12. Nazwa modułu określa nazwę struktur konfiguracyjnych i sterujących

- Odebranie danych po zgłoszeniu przerwania od zapełnionego bufora odbiornika.
- Ignorowanie odebranych danych.
- Zablokowanie nadajnika danych.
- Zablokowanie odbiornika danych.
- Zablokowanie przerwań.

Funkcja R_SCI_SSPI_Write

```
ssp_err_t    R_SCI_SSPI_Write (spi_ctrl_t *const p_ctrl, void
             const *p_src, uint32_t const length, spi_bit_
             width_t const bit_width
```

ma następujące argumenty

- **p_ctrl** – wskaźnik na strukturę konfiguracyjną,
- **p_src** – wskaźnik na bufor z danymi do wysłania,
- **length** – liczba danych do wysłania,
- **bit_width** – długość słowa danych w bitach.

Podobnie jak funkcje *Open* i *Close*, jest zwracany status (kod błędu) wykonania funkcji *spp_err_t*. Zwrócenie *SSP_SUCCESS* oznacza, że jest możliwe wysłanie danych i moduł je rozpoczął. Każda inna zwrócona wartość sygnalizuje niepowodzenie transmisji i program użytkownika musi jakoś na to zareagować. Pozostałe funkcje: *R_SCI_SPI_Read* i *R_SCI_SPI_WriteRead* nie będą używane w procedurach obsługi wyświetlacza i nie będziemy ich dokładniej opisywać.

Programowa obsługa wyświetlacza

4-przewodowy interfejs SPI, oprócz linii MOSI, MISO i SCK, wymaga standardowej linii interfejsu SPI CS (P105) i linii DC (P104) określającej miejsce docelowe przesyłanych danych (rejestry sterujące, lub pamięć obrazu). Poza tym sterownik SSD1306 musi zostać sprzętowo wyzerowany przez wymuszenie stanu niskiego na linii RESET (P106). Wszystkie te linie musza być ustawione jako wyjściowe GPIO w zakładce Pins konfiguratora e2studio jak pokazano na **rysunku 12**. Wyprowadzenia płytki wyświetlacza nie zawierają linii wyjścia danych ze sterownika, a to oznacza jak już

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 92822, pass: 37euo8qf

wiemy, że żadnych danych nie będziemy mogli odczytywać. Do sterowania wyświetlaczem będą nam potrzebne dwie procedury: zapisania rejestrów stepujących, czyli kodu komendy i ewentualnie jej argumentów oraz zapisania danej do pamięci obrazu wyświetlacza (**listing 1** i **listing 2**).

Wysłanie komendy rozpoczyna się od wyzerowania linii DC (komendy) i CS (wybranie układu). Manipulacje poziomami linii wykonywane są za pomocą funkcji *pinWrite* warstwy HAL. Dane na magistralę są wysyłane przez funkcję *R_SCI_SPI_Write*.

Nazwa struktury sterującej *p_ctrl* dla funkcji *R_SCI_SPI_Write* jest określana automatycznie przez konfigurator projektu, jak pokazano na **rysunku 13**. Konfigurator deklaruje struktury konfiguracyjną i sterowniczą oraz zapisuje je wartościami określonymi w oknie *Properties* zwalniając programistę z wykonania tych czynności.

Jeżeli funkcja *R_SCI_SPI* zwróci kod błędu, to program dla celów testowych wchodzi w pętlę nieskończoną. W praktycznych zastosowaniach takie rozwiązanie jest nie do przyjęcia, bo może zablokować cały program, kiedy coś pójdzie nie tak. Użytkownik powinien napisać obsługę błędu tak by program mógł na niego efektywnie zareagować. Kolejna pętla nieskończona czeka na zakończenie wysyłania danych. Tu również należałoby napisać zabezpieczenie przed zablokowaniem programu. Mechanizm wykrywania końca transmisji zostanie opisany za chwilę przy okazji omawiania mechanizmu *callback*.

Wysyłanie danych jest zorganizowane identycznie, tylko przed wysłaniem danych na linia DC jest ustawiana (**listing 2**). Jak już wcześniej wspomniałem funkcja *R_SPI_SCI_Write* inicjalizuje wysyłanie danych i nie sprawdza czy transmisja została zakończona. Dobrze napisane funkcje biblioteczne nie mogą czekać w pętli na wystąpienie jakiegoś zdarzenia, w tym przypadku na zakończenie transmisji danych po magistrali SPI. Jednak wykrycie zakończenia transmisji jest konieczne do tego by móc wysłać kolejna daną i nie spowodować konfliktu na magistrali. W systemach wykorzystujących RTOS można zatrzymać wątek i na semaforze i czekać aż wysyłanie bajtu zostanie zakończone. My nie używamy RTOS i trzeba sobie poradzić inaczej. Zakończenie transmisji jest powiązane ze zgłoszeniem przerwania. Użytkownik może sobie napisać procedurę obsługi i tam testować wystąpienie przerwania sygnalizującego zakończenie transmisji danej. Jednak trochę koliduje to z ogólną ideą użycia bibliotek HAL. Dlatego wprowadzono mechanizm nazwany **callback**. Jeżeli w zakładce *Callback* okna *Properties* drivera SPI zamiast opcji NULL wpiszemy jakąś swoją nazwę, to konfigurator zdefiniuje prototyp funkcji o tej nazwie wywoływanej w momencie zgłoszenia przerwania. Argumentem tej funkcji jest wskaźnik na strukturę zawierającą między innymi informacje o zdarzeniach dotyczących transmisji przez SPI. Zostało to pokazane na **rysunku 13**. Użytkownikowi pozostaje tylko napisać tą swoja funkcję, umieścić ją w programie źródłowym i testować tam na przykład zdarzenie zakończenia transmisji. Ta funkcja musi być formalnie zdefiniowana zgodnie z wymaganiami biblioteki SSP – **listing 3**.

Zdefiniowałem zmienną globalną **data_ready**, która jest zerowana przed każdym wywołaniem funkcji transmisji *R_SCI_SPI_Write*. Po zakończeniu działania tej funkcji testuję jej wartość. Jeżeli w funkcji *spi_callback* zostanie wykryte zdarzenie *SPI_EVENT_TRANSFER_COMPLETE* oznaczające zakończenie wysyłania danej po magistrali SPI, to do *data_ready* jest wpisywana jedynka i procedury wysyłania danych mogą przejść do transferu kolejnej danej. Jak widać nie potrzebujemy pisać własnych procedur obsługi przerwania, ani testować bitów rejestrów konfiguracyjnych, czy rejestrów powiązanych z przerwaniami. Ten mechanizm działa z każdym mikrokontrolerem Synergy wspieranym przez aktualnie dostępna bibliotekę SSP i za każdym razem jest taki sam niezależnie od typu mikrokontrolera.

Listing 1. Wysłanie komendy sterującej do sterownika

```
//wysłanie komendy do sterownika przez SPI
ssp_err_t WriteCmd(uint8_t  cmd){
    ssp_err_t   err;
    uint8_t buffer[2];
    data_ready=0;
    buffer[0]=cmd;
    g_ioport.p_api->pinWrite (DC,IOPORT_LEVEL_LOW);//DC=0;
    g_ioport.p_api->pinWrite (CS,IOPORT_LEVEL_LOW);//CS=0;
    err=R_SCI_SPI_Write(g_spi0.p_ctrl,buffer, 1, SPI_BIT_WIDTH_8_
BITS);
    if(err!=SSP_SUCCESS)
                 while(1);
    while(data_ready==0);
    g_ioport.p_api->pinWrite (CS,IOPORT_LEVEL_HIGH);//CS=1;
    g_ioport.p_api->pinWrite (DC,IOPORT_LEVEL_HIGH);//DC=1;
    return(err);
}
```

Listing 2. Wysłanie danej do sterownika przez SPI

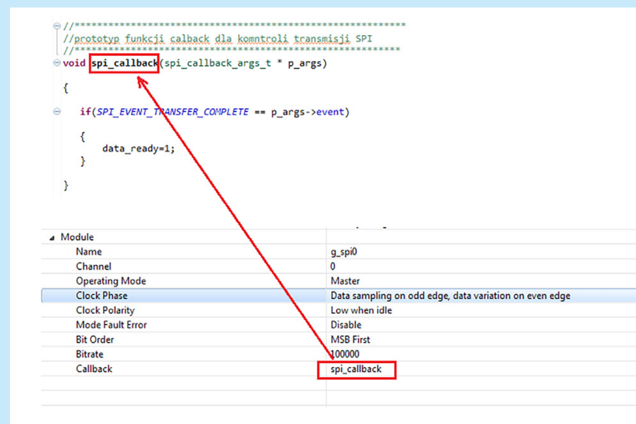
```
//wysłanie danej do sterownika przez SPI
ssp_err_t WriteData(uint8_t data){
    ssp_err_t   err;
    uint8_t buffer[2];
    data_ready=0;
    buffer[0]=data;
    g_ioport.p_api->pinWrite (DC,IOPORT_LEVEL_HIGH);//DC=1;
    g_ioport.p_api->pinWrite (CS,IOPORT_LEVEL_LOW);//CS=0;
    err=R_SCI_SPI_Write(g_spi0.p_ctrl,buffer, 1, SPI_BIT_WIDTH_8_
BITS);
    if(err!=SSP_SUCCESS)
                 while(1);
    while(data_ready==0);
    g_ioport.p_api->pinWrite (CS,IOPORT_LEVEL_HIGH);//CS=1;
       g_ioport.p_api->pinWrite (DC,IOPORT_LEVEL_HIGH);//DC=1;
    return(err);
}
```

Listing 3. Przykład funkcji callback

```
//prototyp funkcji calback dla kontroli transmisji SPI
void spi_callback(spi_callback_args_t * p_args)

{

   if(SPI_EVENT_TRANSFER_COMPLETE == p_args->event)

   {
       data_ready=1;
   }

}
```

Listing 4. Inicjalizacja sterownika SSD1306

```
//inicjalizacja sterownika wyświetlacza
ssp_err_t InitOled(void)
{
   ssp_err_t   err;
    uint16_t i;
    err=R_SCI_SPI_Open(g_spi0.p_ctrl, g_spi0.p_cfg);
    g_ioport.p_api->pinWrite (RES,IOPORT_LEVEL_LOW);//RES=0;
   for(i=0;i<0xffff;i++)
       i=i;
    g_ioport.p_api->pinWrite (RES,IOPORT_LEVEL_HIGH);//RES=1;

    for(i=0;i<28;i++)
        {err=WriteCmd(Buffer_Init[i]);
         if(err!=SSP_SUCCESS)
             while(1);
        }

    DisplayCls(0);
    return(err);
}
```



Rysunek 13. Definiowanie funkcji callback

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 92822, pass: 37euo8qf

Mamy już funkcje wysyłające do sterownika wyświetlacza komendy i dane pamięci obrazu. Możemy zacząć obsługiwać wyświetlacz od procedury inicjalizacyjnej. SSD1306 tak jak większość sterowników po włączeniu zasilania ustawia w rejestrach sterowniczych domyślne wartości nie zawsze właściwe dla zastosowanej matrycy i zazwyczaj program użytkownika musi je zmienić w procesie inicjalizacji. Inicjalizacja polega na wysłaniu szeregu komend do sterownika i realizuje ją procedura **InitOled** pokazana na **listingu 4**. Inicjalizacja rozpoczyna się od wywołania funkcji otwarcia kanału SPI za pomocą funkcji Open i wykonaniu sekwencji zerowania sterownika przez chwilowe wyzerowanie linii RES. Potem jest wysyłanych 28 komend inicjalizujących sterownik. Po włączeniu zasilania pamięć RAM obrazu jest zapisana przypadkowymi wartościami. Dlatego na zakończenie inicjalizacji ta pamięć jest zapisywana wartością zerową (zgaszenie wszystkich pikseli matrycy). Procedura zerująca **DisplayCls** została pokazana na **listingu 5**.

W pamięci RAM mikrokontrolera jest zdefiniowana dwuwymiarowa tablica **static unsigned char DispBuff[128][8];** odpowiadająca pamięci obrazu sterownika. Organizacja bufora odpowiada organizacji pamięci RAM sterownika. DisplayCls najpierw zeruje bufor, a potem wywołuje procedurę RefreshRAM, która zapisuje jego zawartość do pamięci sterownika wyświetlacza (**listing 6**). W docelowej aplikacji będziemy wyświetlać tylko informacje tekstowe – wyświetlacz graficzny doskonale się nadaje do tego celu. W typowych wyświetlaczach alfanumerycznych czcionki mają jednakową wielkość, zależną od wielkości wyświetlacza i mogą być wyświetlane w ustalonych wierszach. W wyświetlaczu graficznym można definiować czcionki o różnych wymiarach zależnie od potrzeb i umieszczać je w dowolnym miejscu wyświetlacza – o ile się tam zmieszczą.

Przy adresowaniu Page Addressing Mode jest bardzo łatwo zdefiniować znaki o wysokości 8 pikseli lub o wielokrotności tej wartości: 16, 24, 32 itd. Najprościej definiować znaki np. 8×6 pikseli umieszczając w pamięci generatora znaków 6 kolejnych bajtów. Niestety, w tym wyświetlaczu znaki o wysokości 8 pikseli są praktycznie nieczytelne. Wynika to po prostu z małych wymiarów (przekątnej) matrycy. Wartością graniczną jest wysokość 12 pikseli, a najlepiej gdyby znaki miały 16 i więcej pikseli.

Do rysowania znaków o dowolnej wielkości potrzebna będzie funkcja **DrawPoint** (**listing 7**), która umożliwia zaświecenie lub zgaszenie pojedynczego piksela o dowolnych współrzędnych x, y, niezależnych od trybu adresowania.

Jak łatwo zauważyć funkcja *void DrawPoint()* z list. 7 tylko modyfikuje zawartość dwuwymiarowego bufora *DispBuff* i nie zapisuje modyfikacji do sterownika SSD1306. Jak zobaczymy dalej wszystkie procedury wyświetlania modyfikują tylko ten bufor i aby zobaczyć efekt tych modyfikacji trzeba przepisać całą zawartość *DispBuff* do pamięci sterownika wywołując funkcję **void RefreshRAM(void)**. Mając do dyspozycji procedurę zaświecającą/gaszącą piksel o konkretnej współrzędnej możemy rysować proste, figury, okręgi, ale też znaki alfanumeryczne o dowolnej wielkości. Żeby to robić, potrzebne są wzorce znaków umieszczone w tablicy zwanej też generatorem znaków. Tablice są tak zbudowane, że kod ASCII znaku adresuje grupę bajtów definiujących ten znak. To znacznie upraszcza procedury wyświetlające łańcuchy znaków (napisy). Na przykład, tablica dla znaków o wysokości 12 i szerokości 6 pikseli jest dwuwymiarową tablicą **const char c_chFont1206[95][12]** i zawiera 95 wzorców znaków . Każdy element wzorca znaków ma 12 bajtów i definiuje znak o szerokości 6 pikseli, ale wykorzystuje przestrzeń 12×8 pikseli. Dla znaków 16×8 pikseli jest zdefiniowana druga tablica **const uint8_t c_chFont1608[95][16]**. Tablica generatora znaków i procedura *DrawPoint* ustawiająca/zerująca bit w buforze pamięci mikrokontrolera, odpowiadający zawartości pamięci RAM sterownika, a tym samym odpowiadający pikselowi na matrycy OLED, pozwala napisać procedurę „rysującą" znak w pamięci RAM mikrokontrolera. Pamiętamy, że aby ten znak

Listing 5. Zerowanie wyświetlacza

```
//zerowanie wyświetlacza

void DisplayCls(uint8_t fill)
{
    uint8_t  i, j;

    for (i = 0; i < 8; i ++) {

        for (j = 0; j < 128; j ++) {
            DispBuff[j][i] = fill;
        }
    }

    RefreshRAM();//zawartosc bufora do RAM obrazu

}
```

Listing 6. Zapisanie zawartości bufora DispBuff do pamięci wyświetlacza

```
void RefreshRAM(void)
{
    uint8_t i, j ;

    for (i = 0; i < 8; i ++) {
        WriteCmd(0xb0+i);
        SetColStart();
        for (j = 0; j < 128; j ++) {
            WriteData(DispBuff[j][i]);
        }
    }
}
```

Listing 7. „Rysowanie" punktu w pamięci obrazu wyświetlacza

```
//"rysowanie" punktu w buforze RAM mikrokontrolera Hosta
void DrawPoint(uint8_t x ,uint8_t y, uint8_t p)
{
    uint8_t chPos, chBx, chTemp = 0;

    if (x > 127 || y > 63) {
        return;
    }
    chPos = 7 - y / 8;
    chBx = y % 8;
    chTemp = 1 << (7 - chBx);

    if (p) {
        DispBuff[x][chPos] |= chTemp;

    } else {
        DispBuff[x][chPos] &= ~chTemp;
    }
}
```

się wyświetlił trzeba przepisać zawartość *DispBuff* do pamięci RAM wywołując funkcję *void RefreshRAM()*.

Na **listingu 8** pokazano procedurę **void DisplayChar()** z argumentami:

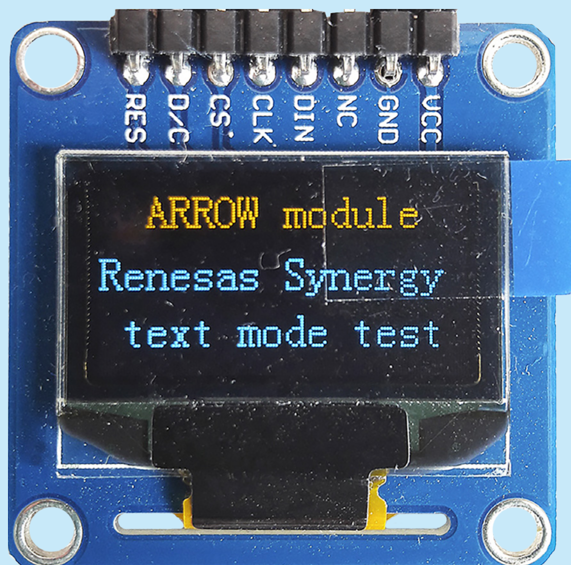
x i y – współrzędne początku znaku na ekranie,

Chr – kod ASCII wyświetlanego znaku,

size – wielkość znaku 12 lub 16 pikseli,

mode=1 wyświetlanie normalne, **mode=0** wyświetlanie w negatywie.

Zależnie od wartości argumentu *size*, bajty wzorca są pobierane z tablicy *c_chFont1206[95][12]* lub z tablicy *c_chFont1608[95][16]*. Jeśli



Rysunek 14. Testowanie trybu alfanumerycznego

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **92822**, pass: **37euo8qf**

```
Listing 8. Wyświetlenie jednego znaku
//wyświetlenie jednego znaku
//argumenty:
//x,y - współrzędne na ekranie
//Chr - kod ASCII znaku
//size - rozmiar 12, lub 16
//mode=1 znak wyświetlany normalnie, mode=0 znak wyświetlany
w negatywie

void DisplayChar(uint8_t x, uint8_t y, uint8_t Chr, uint8_t
size, uint8_t mode)
{
    uint8_t i, j;
    uint8_t chTemp, chYpos0 = y;

    Chr =Chr - ' ';
    for (i = 0; i < size; i ++) {
        if (size == 12) {
            if (mode) {
                chTemp = c_chFont1206[Chr][i];
            } else {
                chTemp = ~c_chFont1206[Chr][i];
            }
        } else {
            if (mode) {
                chTemp = c_chFont1608[Chr][i];
            } else {
                chTemp = ~c_chFont1608[Chr][i];
            }
        }
        for (j = 0; j < 8; j ++) {
            if (chTemp & 0x80) {
                DrawPoint(x, y, 1);
            } else {
                DrawPoint(x, y, 0);
            }
            chTemp <<= 1;
            y ++;

            if ((y - chYpos0) == size) {
                y = chYpos0;
                x ++;
                break;
            }
        }
    }

}
```

argument *mode* jest wyzerowany, to dodatkowo wartość pobranego bajtu jest negowana. Potem jest analizowany każdy bit bajtu wzorca i zależnie od jego wartości *DrawPoint* () zapisuje do bufora *DispBuff* odpowiednią wartość.

Na **listingu 9** pokazano procedurę **void DispTxt()** wyświetlającą łańcuch znaków od określonej pozycji z argumentami:

x i y – współrzędne początku znaku na ekranie,

***txt** – wskaźnik na początek bufora z łańcuchem znaków,

size – wielkość znaku 12 lub 16 pikseli,

mode=1 wyświetlanie normalne, **mode=0** wyświetlanie w negatywie.

Testowanie trybu wyświetlania znaków alfanumerycznych wykonamy umieszczając trzy wywołania funkcji *DispTxt* w funkcji

Rysunek 15. Wyświetlanie bitmapy

```
Listing 9. Wyświetlenie łańcucha znaków
//wyświetlenie łańcucha znaków - napisu

//argumenty
//x,y - współrzędne na ekranie
//*txt - wskaźnik na początek bufora zwierającego łańcuch znaków
ASCII do wyświetlenia
//size - wysokość znaków 12, lub 16 pikseli
//mode=1 znaki wyświetlane normalnie, mode=0 znaki wyświetlane
w negatywie

void DispTxt(uint8_t x, uint8_t y, const uint8_t *txt, uint8_t
size, uint8_t mode)
{
    while (*txt != ,\0') {
        if (x > (SSD1306_WIDTH - size / 2)) {
   wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww-
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww         x
= 0;
            y += size;
            if (y > (SSD1306_HEIGHT - size)) {
                y = x = 0;
                DisplayCls(0x00);
            }

        }

        DisplayChar(x, y, *txt, size, mode);
        x += size / 2;
        txt ++;
    }
}
```

```
Listing 10. Testowanie trybu alfanumerycznego
void hal_entry(void)
{
    InitOled();
    DispTxt(0, 0, „  ARROW module", 16,1);
    DispTxt(0, 20, „Renesas Synergy", 16,1);
    DispTxt(0, 40, „ text mode test „, 16,1);
    RefreshRAM();
    while(1);
}
```

hal_entry, jak pokazano na **listingu 10**. Efekt działania tego testu pokazano na **rysunku 14**.

Każda czcionka wyświetlana w trybie tekstowym jest bitmapą rysowaną na ekranie. Te bitmapy – wzorce znaków umieszczane są w tablicy generatora znaków i mają jednakowe wymiary. Jednak często zachodzi konieczność wyświetlania bitmap o różnych wymiarach. Będziemy potrzebowali programu konwertującego bitmapy monochromatyczne na tablicę w języku C w formacie zgodnym z organizacją pamięci RAM sterownika wyświetlacza. Ja używam prostego programu **bmp2.exe** służącego do konwersji monochromatycznej mapy o niewielkiej rozdzielczości na tablicę w języku C. Kiedy już mamy skonwertowaną bitmapę, to procedura jej wyświetlania jest stosunkowo prosta – pokazano ją na **listingu 11**. Wywołanie funkcji rysującej pełnowymiarową bitmapę pokazano na **listingu 12**. Tablica bmparray jest generowana automatycznie przez program bmp2.exe. Po zapisaniu bufora pamięci DispBuff procedura RefreshRAM przepisuje jego zawartość do pamięci sterownika wyświetlacza. Efekt działania programu wyświetlającego bitmapę jest pokazany na **rysunku 15**.

Tomasz Jabłoński, EP

```
Listing 11. Rysowanie bitmapy
//******************************
//rysuj bitmapę
//argumenty
//x,y współrzędne początku bitmapy
//*bmp wskaźnik na tablicę z przekonwertowaną  bitmapą
//width  - szerokość bitmapy
//height - wysokość bitmapy
//******************************

void DrawBmp(uint8_t x, uint8_t y, const uint8_t *bmp, uint8_t
width, uint8_t height)
{
    uint16_t i, j;
    uint16_t byteWidth = (width + 7) / 8;

    for(j = 0; j < height; j ++){
        for(i = 0; i < width; i ++ ) {
            if(*(bmp + j * byteWidth + i / 8) & (128 >> (i &
7))) {
                DrawPoint(x + i, y + j, 1);
            }
        }
    }
}
```

```
Listing 12. Rysowanie bitmapy o rozdzielczości 128×64 piksele
DrawBmp(0,0, bmparray, 128, 64);
RefreshRAM();
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP:
ftp://ep.com.pl, user: 92822, pass: 37euo8qf