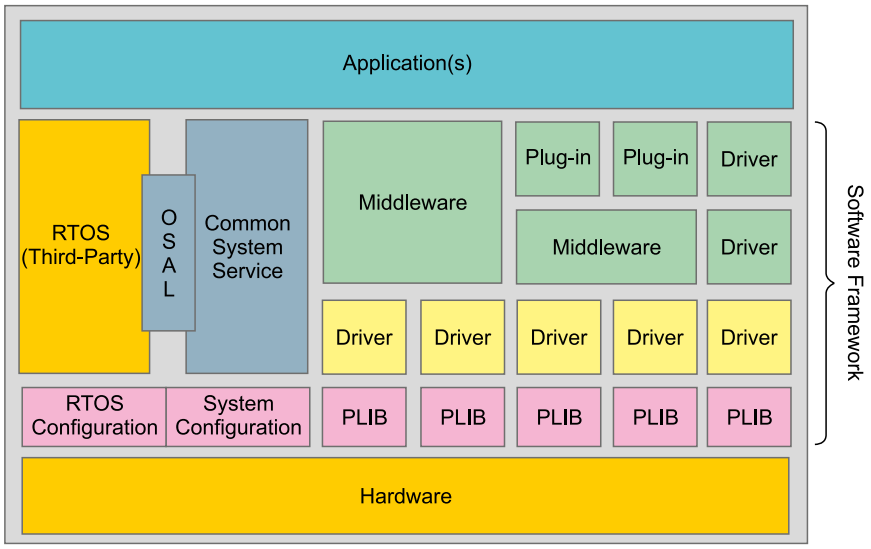
MPLAB Harmony, czyli harmonia według Microchipa

Konstruktorzy i programiści systemów embedded, nawet ci, którzy nie stosują mikrokontrolerów Microchip, na pewno chociażby słyszeli o darmowym stosach TCP/IP i USB oraz innych bibliotekach dla mikrokontrolerów PIC. Ten producent od lat kontynuuje swoją politykę udostępniania darmowych narzędzi, coraz to lepszych i coraz bardziej zaawansowanych.

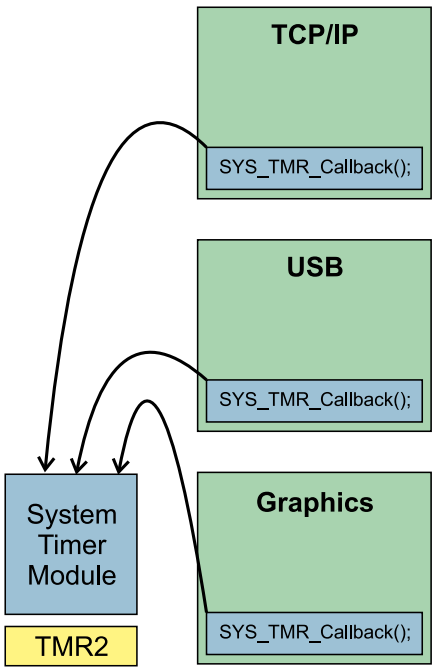
Pamiętam, gdy firma Microchip zaczęła oferować za darmo kompilator asemblera dla rodziny mikrokontrolerów PIC16C, bo wtedy tylko takie miała w ofercie. Darmowy asembler nie był wtedy czymś powszechnym i zazwyczaj za takie oprogramowanie trzeba było płacić. Producenci mikrokontrolerów skupiali się na produkcji układów, a niezależne firmy na tworzeniu oprogramowania narzędziowego. Strategia oferowania bezpłatnych programów narzędziowych była potem kontynuowana przez Microchip – oprócz asemblera zaoferowano wersje ewaluacyjne kompilatorów C, a potem biblioteki MLA (Microchip Libraries for Applications). Najbardziej znanym elementem MLA był całkowicie bezpłatny stos TCP/IP. Jedynym ograniczeniem licencyjnym było ograniczenie do stosowania z mikrokontrolerami PIC. Dzisiaj MLA oferują miedzy innymi stos TCP/IP, stos USB, system plików FAT16/32 oraz bibliotekę graficzną.

Oczywiście, dzisiaj inne firmy półprzewodnikowe działają podobnie. Sprzedaż mikrokontrolerów łączy się nierozerwalnie z oferowaniem bezpłatnego oprogramowania narzędziowego z zestawem bezpłatnych, rozbudowanych bibliotek. Przykładem takiego podejścia może być projekt *mbed* skupiony wokół mikrokontrolerów różnych producentów z rdzeniem ARM.

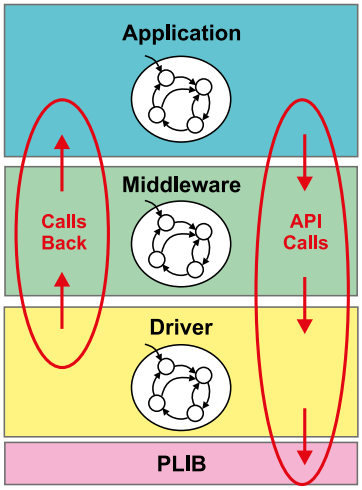
Wprowadzenie do oferty 32-bitowych mikrokontrolerów PIC32MX, a ostatnio PIC323MZ, zapewne było impulsem do uporządkowania bezpłatnego oprogramowania. Zdecydowano się zaprzestania rozwijania bibliotek w tej formie, a zamiast tego wprowadzono nowe rozwiązanie nazwane Harmony. Po co ta zmiana? Załóżmy, że jest programista lub zespół programistów, którzy projektują i wykonują oprogramowania dla konkretnego mikrokontrolera. Po zakończeniu oprogramowanie zostaje przetestowane i wdrożone. Ale z powodu zmiany wymagań klienta lub z powodu wymogów konkurencyjności rynku trzeba coś zmienić – na przykład zastosować nowszy, bardziej wydajny mikrokontroler lub odwrotnie, w celu zmniejszenia kosztów, zastosować prostszy i tańszy mikrokontroler. Jeżeli program był napisany w postaci warstw, od warstwy obsługującej sprzęt do warstwy aplikacji, to może nie będzie trzeba pisać wszystkiego od nowa. Ale tak czy inaczej, zmiana mikrokontrolera będzie wymagała zmiany procedur odpowiedzialnych za obsługę sprzętu. Jest to bardzo pracochłonne i łatwo przy tym



Rysunek 1. Schemat blokowy MPLAB Harmony



Rysunek 2. Działanie usługi systemowej dostępu do czasu systemowego



Rysunek 3. Współpraca modułów w MPLAB Harmony

popełnić błędy. Harmony oprócz tego, że dostarcza biblioteki, tak jak MLA, to te biblioteki jest łatwo implementować po zmianie typu mikrokontrolera i ogólnie dużo łatwiej i szybciej można zmodyfikować działanie całego projektu.

MPLAB Harmony – założenia ogólne

MPLAB Harmony prawie w całości napisano w języku C (ze wsparciem dla C++). Kluczowe elementy mają budowę modułową i są zorientowane obiektowo. Projekt może pracować w zależności od wymagań, wiedzy i doświadczenia programisty, pod kontrolą RTOS lub bez niego. Harmony zapewnia moduły programowe, które są łatwe w użyciu, konfigurowalne i współpracują ze sobą – jak zapewnia producent – w pełnej harmonii. Zapewne stąd wywodzi się marketingowa nazwa MPALB Harmony.

Jednym z ważnych założeń jest łatwa przenośność oprogramowania. Harmony zapewnia nieskomplikowane funkcje umożliwiające z jednej strony, obsługę układów peryferyjnych, a z drugiej pewien stopień abstrakcji. Przy zmianie mikrokontrolera trzeba wymienić funkcje na te odpowiednie dla danego typu układu. Ta wymiana następuje w dużej części automatycznie i jest wykonywana przez MPLAB Harmony. Oczywiście, nie zwalnia to programisty od orientowania się w szczegółach sprzętowych mikrokontrolera. Musi wiedzieć jak są zbudowane układy peryferyjne i jak działa system przerwań – Harmony tylko znacznie ułatwia ich użycie.

Jedną z głównych właściwości MPLAB Harmony jest stosowanie sterowników urządzeń – device drivers. Sterowniki urządzeń, to funkcje realizujące warstwę interfejsu pomiędzy elementarnymi funkcjami dostępu do sprzętu (PLIB), a wyższymi warstwami projektu. Device driver jest odpowiedzialny za zarządzaniem dostępem do układów peryferyjnych, aby żądania dostępu do nich z wyższych warstw nie kolidowały pomiędzy sobą. Zapewnia to niezakłóconą pracę urządzeń peryferyjnych.

Funkcje biblioteki peryferii *Peripheral Library PLIB* zapewniają dostęp do układów peryferyjnych konkretnego mikrokontrolera, umożliwiają ukrycie szczegółów rejestrów sterujących SFR i przez to ułatwiają funkcjom warstwy drivera urządzeń obsługę różnych mikrokontrolerów. Funkcje warstwy aplikacji nie powinny bezpośrednio używać funkcji biblioteki peryferii, mimo że zapewniają one pewien poziom abstrakcji. Jak już wspominałem, w celu bezkolizyjnego zarządzania dostępem do peryferii funkcje PLIB są wywoływane przez procedury drivera urządzeń. Zapobiega to na przykład sprzecznym żądaniom dostępu od różnych modułów programu.

Kolejną właściwością MPLAB Harmony jest budowa modułowa, co pozwala na podzielenie aplikacji na „części" i przydzielenie im zasobów, którymi każdy z modułów zarządza autonomicznie. Interfejs każdej biblioteki zawiera spójny zestaw funkcji (bez zmiennych globalnych lub współdzielenia rejestrów). Jeżeli jeden moduł musi korzystać z zasobów innego modułu, to wywołuje w nim funkcję interfejsu. Takie podejście pomaga w wyeliminowaniu konfliktów pomiędzy modułami i pozwala na budowanie aplikacji jakby z klocków.

Sterowniki urządzeń są przeznaczone do obsługi w miarę prostych układów peryferyjnych na przykład Timerów, UART, I²C, SPI, C/A, A/C itp. Bardziej skomplikowane peryferia na przykład USB, czy obsługa LAN z protokółem TCP/IP wymaga obsługi złożonych protokołów. Równie rozbudowane są procedury graficzne. Do obsługi tych skomplikowanych zadań przewidziano warstwę nazwaną **middleware** znajdującą się pomiędzy sterownikami urządzeń, a warstwą aplikacji użytkownika. MPALB Harmony ma w tej warstwie bezpłatne: stos obsługi USB, stos TCP/IP, bibliotekę graficzną. W wersji płatnej mogą być dostępne inne biblioteki, na przykład stos Bluetooth.

Uruchamianie skomplikowanych procesów w warstwie aplikacji i w warstwie middleware może spowodować kolizje dostępu do zasobów. Na przykład, jeżeli program korzysta ze stosu USB i jednocześnie z procedur graficznych, to będą one wykonywały swoje zadania sekwencyjnie. Te zadania mogą w pewnym momencie żądać np. jednoczesnego dostępu do systemowego licznika Timer2. Żeby temu zapobiec, stosuje się usługę systemową *SYS_TMR_Callback();* opartą o działanie licznika Timer2. Jej zadaniem jest zapobieganie kolizjom dostępu i jednocześnie zapewnienie dostępu do usługi czasowej każdemu z żądających procesów (**rysunek 2**). Każdy z procesów korzysta z odseparowanych żądań i dzięki temu nie przeszkadzają sobie nawzajem.

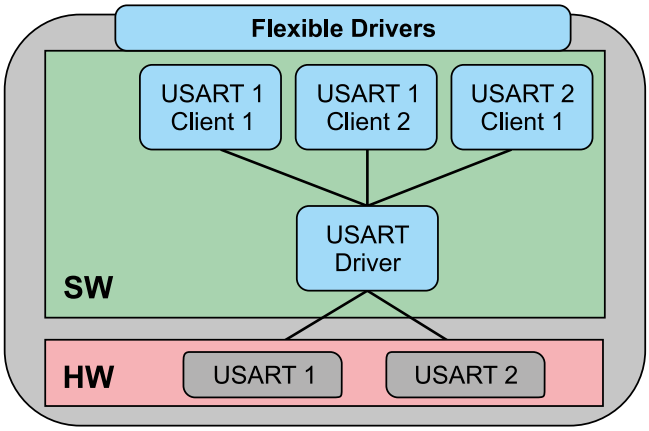
Usługa timera systemowego może być skonfigurowana według potrzeb. Domyślnie jest to Timer2, ale można użyć na przykład Timer3. Używanie usługi systemowej jest podobne do używania drivera urządzenia. Różnica polega tylko na tym, że driver wymaga użycia polecenia „open" dla utworzenia unikalnego połączenia klient-driver. Dla usługi systemowej nie jest to konieczne, bo są one współdzielone przez wielu klientów w systemie.

Moduły MPLAB Harmony: drivery urządzeń, usługi systemowe oraz middleware (bez modułów PLIB) są wstanie aktywnym. Oznacza to, że kiedy aplikacja wywołuje funkcję interfejsu modułu, to moduł odpowiada natychmiast a potem kontynuuje swoją pracę, żeby zakończyć operację. Większość modułów wystawia zgłoszenie i aplikacja (klient) „wie", kiedy operacja została zakończona.

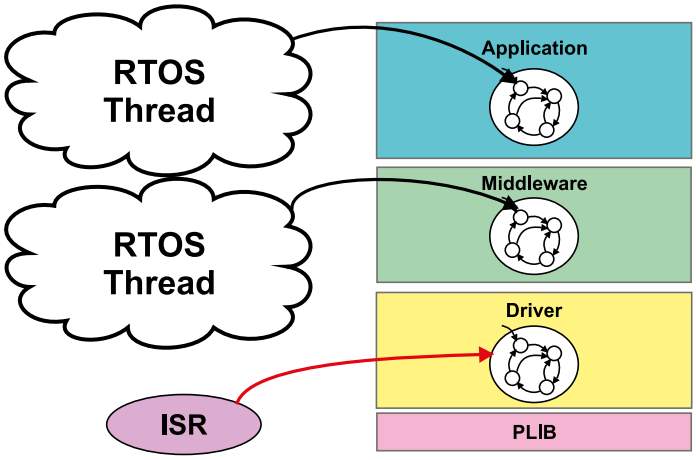
Moduły są implementowane jako maszyna stanu (*cooperative state machine*). Użytkownik definiuje zestaw dopuszczalnych stanów i wykonuje inicjalizację maszyny stanów. Każdy z modułów ma swoją funkcję inicjalizacji i jedną z lub więcej funkcji wykonujących zadania (*task functions*). Funkcje inicjalizacji realizujące między innymi stan początkowy maszyny stanu powinny być wykonane po restarcie mikrokontrolera tak szybko, jak to tylko możliwe. Moduły pod kontrolą warstwy aplikacji współpracują każdy z każdym przez wywoływanie funkcji interfejsu innych modułów. Po inicjalizacji w konfiguracji z poolingiem (bez RTOS) system przechodzi do nieskończonej pętli głównej, w której moduły są wywoływane cyklicznie i przez to utrzymywane przez system w stanie aktywnym do ewentualnego wykonania swojego zadania. Ta technika pozwala na prostą implementację wielozadaniowości.

Taka metoda nie jest uniwersalna i może się nie sprawdzić we wszystkich przypadkach. Dlatego w Harmony przewidziano inne konfiguracje. Jednak metoda maszyny stanów z poolingiem jest najprostsza do zrozumienia, łatwa w implementacji i debugowaniu. Ponadto pokazuje w najbardziej przystępny sposób jak moduły w MPLAB Harmony współdziałają ze sobą (**rysunek 3**).

Podstawowy model MPLAB Harmony, w którym moduły kooperują ze sobą i są konfigurowalne, zaspokaja potrzeby niemal każdego systemu wbudowanego. Na przykład, gdy chcemy użyć wielu identycznych peryferii Harmony wykonuje dynamiczną implementację drivera urządzenia. Wtedy wielu klientów może w tym samym czasie korzystać z układu peryferyjnego (przykład timera systemowego opisanego wcześniej). Dynamiczny driver w sposób inteligentny zarządza wieloma próbami dostępu od wielu klientów do jednego zasobu (**rysunek 4**).



Rysunek 4. przykład implementacji dynamicznego drivera USART'a



Rysunek 5. Zastosowanie RTOS

Taki driver jest oczywiście rozbudowany programowo. Jeżeli nasza aplikacja nie przewiduje wykorzystania dynamicznego drivera, to można zastosować konfigurację statyczną, zajmującą mniej pamięci procesora.

Są systemy, które wykorzystują middleware i muszą mieć „otwartych" kilka niezależnych aplikacji jednocześnie. W takim wypadku opisywana metoda poolingu używająca pętli głównej do sekwencyjnego wywoływania modułów może nie być wystraczająca. Tak się dobrze składa, że moduły Harmony mogą być uruchamiane bezpośrednio z procedurach obsługi przerwania ISR lub przez wątki systemu RTOS. Użycie przerwań z odpowiednio zaprogramowanymi priorytetami eliminuje opóźnienie spowodowane oczekiwaniem na uruchomienie przez inny moduł. To opóźnienie może być spowodowane oczekiwaniem na dokończenie pollinigu w pętli głównej maszyny stanów.

Gdy system jest na tyle skomplikowany, że trudno jest zapanować nad spełnieniem wykonania zadania przy użyciu maszyny stanu i systemu przerwań, to można zastosować system czasu rzeczywistego RTOS. Na szczęście, wszystkie funkcje, które są wywoływane poprzez pooling mogą być równie łatwo wywoływane przez RTOS wykorzystujący zbiór funkcji OSAL (*Operating System Abstraction Layer*). OSAL umożliwia wywoływanie modułów przez różne systemy operacyjne RTOS, ale może tez być użyte samodzielnie (bez RTOS).

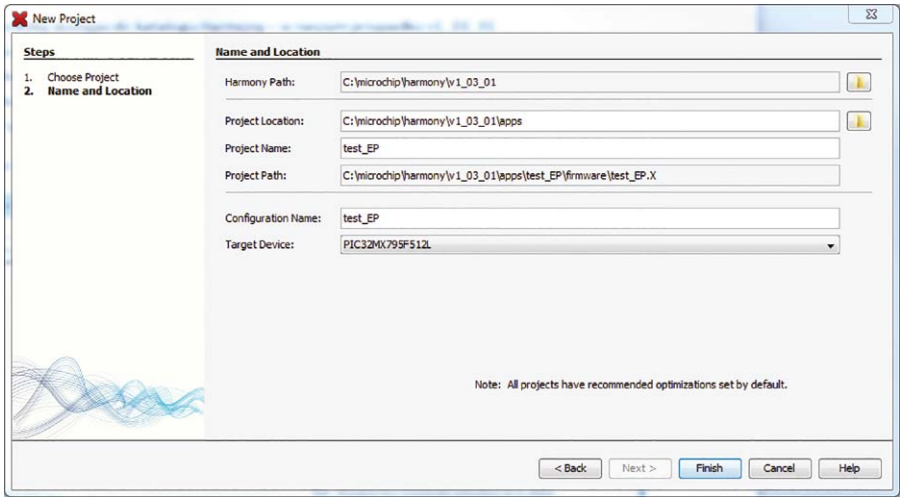
MPLAB Harmony wpiera różne możliwości konfiguracji:

- Wybór mikrokontrolera.
- Wybór metody: pooling lub przerwania.
- Konfiguracja driverów urządzenia: statyczna lub dynamiczna.
- Konfiguracja driverów urządzenia: pojedynczy klient lub wielu klientów.
- Konfiguracja innych.

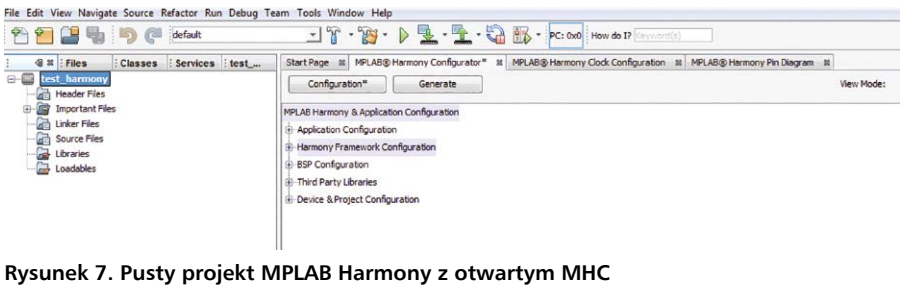
Projekt jest bardziej lub mniej skomplikowany, ale zawsze ma pewną minimalną ilość plików źródłowych i nagłówkowych. W przypadku bardziej skomplikowanych projektów stosuje się metodę tworzenia szkieletu projektu z niezbędnymi plikami źródłowymi i nagłówkowymi. W dokumentacji Harmony jest rozdział dokładnie opisujący wszystkie te pliki: ich zawartość „startową" i umiejscowienie w katalogach i podkatalogach projektu. Ja z ciekawości utworzyłem sobie taki szkieletowy projekt. Wymaga to sporo pracy i uwagi, gdyż łatwo się przy tym pomylić. Po tym doświadczeniu zacząłem szukać firmowych projektów szkieletowych i… niczego nie znalazłem! Nie znalazłem też takich projektów umieszczonych w Internecie przez innych użytkowników. Nie ma ich, bo po prostu są niepotrzebne. Tu dochodzimy do kolejnego poziomu wsparcia: wtyczki (plug-in) *MPLAB Harmony Configurator* – MHC. W MPLAB X IDE jest zaimplementowany mechanizm pobierania i instalowania wtyczek bezpośrednio z okna Tools-Plugins. W zakładce *Available Plugins* są wyświetlane dostępne wtyczki. Po zaznaczeniu *MPLAB Harmony Configurator* i kliknięciu na przycisk *Install* wtyczka zostanie automatycznie pobrana z serwera Microchipa i zainstalowana. Można ją uruchomić z okna *Tools* → *Embeded* → *MPLAB Harmony Configurator*.

Przykład 1 – zaświecenie diody LED

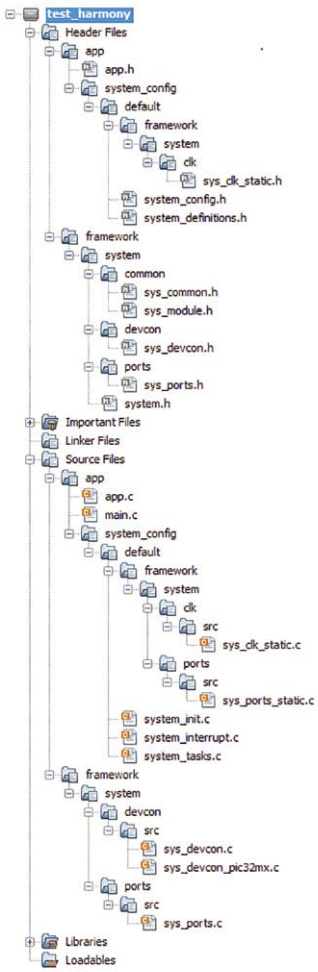
W katalogu *V1_03_01/apps/examples* umieściłem swój katalog o nazwie *test_harmony/fimware*. Następnie, utworzyłem nowy projekt *MPLAB Harmony Project* o nazwie *test_harmony* z mikrokontrolerem *PIC32MX795F512L*. Ten projekt nie zawiera ani żadnych plików źródłowych, ani nagłówkowych, ale ma przygotowane wszystkie niezbędne foldery. Na tym etapie musimy mieć pobrany i zainstalowany plug-in MHC. Jeżeli



Rysunek 6. Definicje projektu MPLAB Harmony



Rysunek 7. Pusty projekt MPLAB Harmony z otwartym MHC



Rysunek 8. Struktura plików projektu

tak jest, to zostanie on automatycznie otwarty (**rysunek 7**). Po kliknięciu na przycisk *Configurations* możemy:

- Zapisać bieżącą konfigurację z domyślną nazwą.
- Zapisać bieżącą konfigurację z nową nazwą.
- Otworzyć inną wcześniej zapisaną konfigurację.
- Zmienić ścieżkę dostępu do katalogu z MPLAB Harmony.

Na początek zapiszemy bieżącą konfigurację z domyślną nazwą w domyślnym katalogu projektu. Potem klikamy na przycisk *Generate* i MHC automatycznie tworzy szkielet projektu z wszystkimi plikami wynikającymi ze startowej domyślnej konfiguracji MHC. Struktura plików projektu Harmony jest dość rozbudowana nawet w tym projekcie, który na tym etapie nic nie robi (**rysunek 8**). Pliki projektu można umownie podzielić na:

- Konfiguracyjne: inicjalizacja systemu, opcje statycznej konfiguracji i procedury sterowania systemem.
- Bibliotek: inicjalizacja bibliotek i procedury bibliotek peryferii.
- Aplikacji: inicjalizacja i warstwy procedury aplikacji.

Po wygenerowaniu plików przez MPLAB Harmony Configurator mamy gotowy

Listing 1. Plik main.c

```
int main ( void )
{
//Inicjalizacja modułów MPLAB Harmony , oraz inicjalizacja aplikacji
    SYS_Initialize ( NULL );
//pętla poolingu
    while ( true )
    {
        /* funkcja maszyny stanów . */
        SYS_Tasks ( );
    }
    return ( EXIT_FAILURE );
}
```

Listing 2. Inicjalizacja systemu

```
void SYS_Initialize ( void* data )
{
    /* Inicjalizacja taktowania rdzenia  */
    SYS_CLK_Initialize( NULL );
    sysObj.sysDevcon = SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_
INIT*)&sysDevconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_SystemFrequencyGet());
/*wyłączenie interfejsu JTAG*/
    SYS_DEVCON_JTAGDisable();
/*inicjalizacja modułu portów*/
    SYS_PORTS_Initialize();
    /* Inicjalizacja driverów urządzen*/
    /* Inicjalizacja  usług systemowych*/
    /* Inicjalizacja  Middleware */
    /* Inicjalizacja aplikacji użytkownika */
    APP_Initialize()
}
```

szkielet projektu, na którym można osadzić naszą warstwę aplikacji.

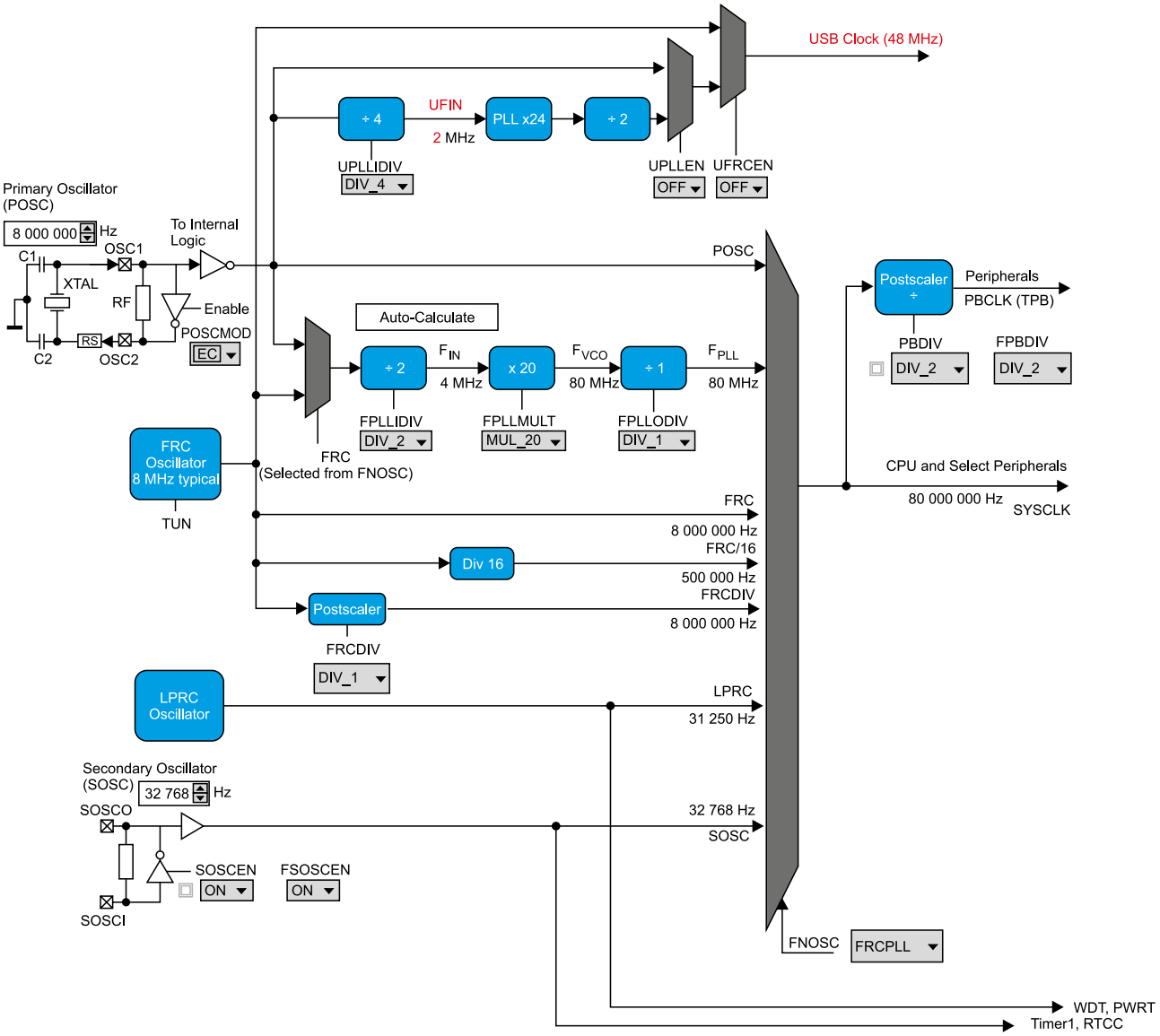
Wygenerowany przez MHC plik *main.c* zawiera tylko procedurę inicjalizacji *SYS_Initialize()* i niekończoną pętlę *while(true)* – **listing 1**. Ta pętla realizuje mechanizm poolingu, o którym pisałem wcześniej. Wywoływana w tej pętli procedura *SYS_Task()* realizuje funkcję jednej maszyny stanów. Inicjalizacja systemu *SYS_initialize()* (**listing 2**) polega głównie na zainicjowaniu układów taktowania rdzenia i układów peryferyjnych, domyślnym wyłączeniu interfejsu JTAG oraz zainicjowaniu układów portów. Na tym etapie nie używamy ani driverów urządzeń, ani middleware. Kiedy te moduły pojawią się w projekcie, to muszą tu być zainicjowane.

Inicjalizacja aplikacji polega na zdefiniowaniu stanu początkowego maszyny stanów. W pliku *app.h* jest umieszczona definicja stanu początkowego *APP_STATE_INIT*. Jeśli użytkownik będzie rozbudowywał swoją aplikację, to musi tu definiować kolejne stany maszyny stanów. Jedną z ważniejszych funkcji w systemie jest funkcja *APP_Tasks ()* umieszczona w pliku *app.c*. To właśnie w tej funkcji będzie implementowana maszyna stanów. *APP_Tasks ()* jest stale wywoływana w nieskończonej pętli głównej funkcji *main()*.

Tak wygląda projekt, który został domyślnie skonfigurowany przez plug-in MHC w trakcie tworzenia w środowisku MPLAB X IDE. Popatrzmy teraz jak można go samodzielnie skonfigurować do wykonywania prostych czynności. Do testów użyłem modułu **PIC32 USB Starter KIT II** z mikrokontrolerem **PIC32MX795F512L**. Jest to jeden z najbardziej zaawansowanych mikrokontrolerów rodziny PIC32MX. Pierwszą czynnością konfiguracyjną będzie ustalenie źródła i częstotliwości taktowania mikrokontrolera. System zegarowy powinien dostarczać trzy niezależnie programowane częstotliwości:

- SYSCLK do taktowania rdzenia.
- PBCLK do taktowania układów peryferyjnych.
- Częstotliwość 48MHz do taktowania interfejsu USB.

Zwykle do taktowania układu zegarowego wykorzystuje się dwa źródła sygnału zegarowego: oscylator kwarcowy (*Primary Oscillator*) i wbudowany, precyzyjny oscylator RC o częstotliwości 8 MHz. Oscylator kwarcowy może pracować w trybach EC, XT, lub HS. Typowo wybiera się tryb HS i kwarc o częstotliwości 8 MHz lub 12 MHz. Częstotliwość źródła (oscylator kwarcowy lub RC) może być następnie powielana w układach PLL do częstotliwości 80 MHz, czyli maksymalnej częstotliwości, z którą pracuje PIC32MX795F512L. Konfigurowanie zegara w MHC jest tak łatwe, jak to można sobie tylko wyobrazić. Służy do tego zakładka *MPLAB Harmony Clock Configurator* (**rysunek 9**). Najpierw w oknie FNOST wybieramy źródło sygnału zegarowego. *Oprócz Primary Oscillator* i *FRC 8MHz* można wybrać też *Secondary oscillator* lub generator *LPRC*. Są to oscylatory o małej częstotliwości i w normalnych warunkach nie będą używane. Mikrokontroler może być taktowany wprost z POSC lub FRC, albo częstotliwości taktujące mogą być powielane przez układ PLL. Ja wybrałem oscylator *FRC* z układem PLL, czyli *FNOST=FRCPLL*. Aby uzyskać żądaną częstotliwość trzeba zaprogramować współczynniki *FPLLIDIV*, *FPLLMULT* i *FPLLODIV*. Można to zrobić ręcznie, lub automatycznie klikając na przycisk *Auto-Calculate*. W oknie *Desired System Frequency* wpisujemy interesująca nas częstotliwość taktowania rdzenia i zostanie



Rysunek 9. Okno konfiguratora systemu taktowania

wyliczona częstotliwość możliwa do uzyskania z układu PLL oraz błąd częstotliwości w „%”. Dla FRC =8 MHz i SYSCLK=80 MHz *Auto-Calculate* wylicza współczynniki FPLLIDIV=DIV_2, FPLLMULT=MUL_20 i FPLLODIV=DIV_1. Częstotliwość zegara taktującego układy peryferyjne PBCLK jest uzyskiwana z SYSCLK po podzieleniu przez 1, 2, 4 lub 8 (FPBDIV). Jeżeli w projekcie jest używany interfejs USB, to jego taktowanie również trzeba zaprogramować, aby USB Clock był równy 48 MHz. W *Harmony Clock Configurator* nieprawidłowa częstotliwość USB Clock jest sygnalizowana na czerwono.

Po ustawieniu systemu taktowania trzeba zapisać konfigurację i wygenerować pliki projektu z poziomu *MPLAB Harmony Configurator* (*Generate*). Definicje bitów konfiguracyjnych zostaną automatycznie zaktualizowane i można je obejrzeć w oknie *Device & Project Configuration → PIC32MX795F512L Configuration*. Zmiany są zaznaczane przez podświetlenie na fioletowo (**rysunek 10**).

PO skonfigurowaniu taktowania możemy przystąpić do napisania pierwszego programu wykonującego jakąś czynność. Nie będzie to bardzo ambitne zadanie, bo spróbujemy zaświecić diodę LED umieszczoną na module PIC32 USB Starter KIT II. W module są umieszczone 3 diody LED połączone do linii portów RD0, RD1 i RD2, oraz 3 przyciski podłączone do linii RD6, RD7 i RD13. Jeżeli producentem mikrokontrolera, modułu ewaluacyjnego i oprogramowania jest ta sama firma, to często się zdarza, że moduły ewaluacyjne są wspierane. Do tego celu w MHC służy opcja konfiguracyjna *BSP Configuration* (**rysunek 11**).

Po skonfigurowaniu BSP PIC32MX USB Starter Kit2 i wygenerowaniu plików przez MHC w projekcie zostaną utworzone pliki *bsp_sys_init.c* i *bsp_config.h*. W *bsp_sys_init.c* są umieszczone funkcje inicjujące linie portów sterujących LED oraz funkcje sterujące diodami i czytające stany wejść podłączonych do przycisków. Można skorzystać z tych funkcji lub użyć funkcji bibliotecznych MPLAB Harmony.

Ponieważ obsługa portów ma ścisły związek ze sprzętem, to musi być umieszczona w warstwie PLIB. W dokumentacji Harmony można znaleźć dokładny opis wszystkich funkcji obsługi portów z warstwy Peripherial Library PLIB. W zasadzie nic nie stoi na przeszkodzie, by tych funkcji użyć i wszystko będzie działało. Ale pamiętamy, że mocno akcentowana idea programowania w tym środowisku nie zaleca stosowania funkcji PLIB przez naszą warstwę aplikacji. Ma to ścisły związek z możliwością dostępu w jednym czasie do tego samego zasobu (portów) przez różne moduły. Oczywiście w tak prostej aplikacji jak zapalenie jednej diody taki konflikt nie będzie możliwy, ale warto od początku stosować zalecane zasady.

Przed zaświeceniem diody LED trzeba skonfigurować linie RD0, RD1 i RD2 jako wyjścia. W mikrokontrolerach Microchipa po zerowaniu wszystkie linie portów są domyślnie wejściami, a linie, które są multipleksowanymi wejściami przetwornika A/C są dodatkowo ustawione jako wejścia analogowe. Do ustawiania kierunku przepływu danych jednej linii portu wykorzystywane są funkcje:

Listing 3. Definicja stanów maszyny stanów

```
typedef enum
{
//definicja stanu początkowego maszyny stanów
    APP_STATE_INIT=0,
} APP_STATES;
```

Listing 4. Funkcja inicjalizacji i implementacji maszyny stanów

```
void APP_Tasks ( void )
{
//testowanie stanów aplikacji
    switch ( appData.state )
    {
//stan początkowy aplikacji
        case APP_STATE_INIT:
        {
            break;
        }
        /* miejsce na implementacje maszyny stanów .*/
        /* default nie powinien być nigdy wykonany  */
        default:
        {
            /*sygnalizacja o obsługa błędu stanów maszyny stanów */
            break;
        }
    }
}
```

Listing 5. Inicjalizowanie RD0…RD2 jako linii wyjściowych

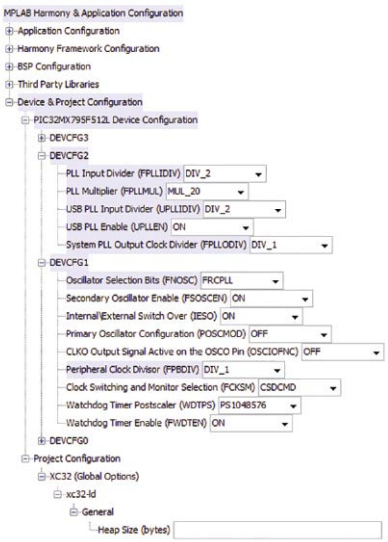
```
switch ( appData.state )
{
    /* Stan inicjalizacji aplikacji */
    case APP_STATE_INIT:
    {
        /* ustawienie linii RD0 jako wyjściowej*/
        SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_OUTPUT,
PORT_CHANNEL_D ,BSP_LED_1);
        /* ustawienie linii RD1 jako wyjściowej*/
        SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_OUTPUT,
PORT_CHANNEL_D ,BSP_LED_2);
        /* ustawienie linii RD2 jako wyjściowej*/
        SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_OUTPUT,
PORT_CHANNEL_D ,BSP_LED_3);
}
```

Listing 6. Definicja nowych stanów maszyny stanów

```
typedef enum
{
    /* stany maszyny stanów warstwy aplikacji  */
    APP_STATE_INIT=0, /*stan inicjalizacji */
    APP_LED_ON, /*stan zapalenia diody LED i */
    APP_NOP, /*stan nic nie rób */
} APP_STATES;
```

Listing 7. Zmodyfikowana funkcja APP_Tasks

```
void APP_Tasks ( void )
{
    /* Testowanie stanów maszyny stanów aplikacji  */
    switch ( appData.state )
    {
/* Stan inicjalizacji aplikacji */
        case APP_STATE_INIT:
        {
            SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_1);
            SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_2);
            /* Po inicjalizacji stan do zapalenia LED*/
           appData.state=APP_LED_ON;
            break;
        }
        case APP_LED_ON :
        {
        /* Zapalenie diod  LED1 i LED2 */
            SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_1);
            SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_2);
/* Po zapaleniu diod LED nic nie rób  */
            appData.state=APP_NOP;
        }
        case APP_NOP:
        {
         /* Stan nic nie rób */
            ;
        }
        default:
        {
            break;
        }
    }
}
```



Rysunek 10. Zmiany w konfiguracji układu taktowania

Rysunek 11. konfiguracja wsparcia modułu PIC32MX USB start Kit 2

- *PLIB_PORTS_PinDirectionOutputSet* dla warstwy Peripherial.
- *SYS_PORTS_PinDirectionSelect* dla warstwy drivera urządzeń.

Ustawienie kierunku linii można wykonać w procedurze inicjalizacji aplikacji, kiedy aktywny jest stan *APP_STATE_INIT* maszyny stanów (**listing 5**).

Argumentami funkcji SYS_PORTS_PinDirectionSelect są:

- *PORTS_ID_0* – identyfikacja modułu portu
- *SYS_PORTS_DIRECTION_OUTPUT* – linia konfigurowana jako wyjście.
- *PORT_CHANNEL_D* – kierunek linii portu PORTD.
- *BSP_LED_1* – identyfikacja linii w porcie.

Definicja linii *BSP_LED_1* jest zapisana w pliku *bsp_sys.init*.

Funkcja ustawiania linii w bibliotece PLIB wygląda podobnie – *PLIB_PORTS_PinDirectionOutputSet(PORTS_ID_0, PORT_CHANNEL_D, PORTS_BIT_POS_1);*

Teraz sobie zdefiniujemy kolejne dwa stany maszyny stanów w pliku *app.h* (*APP_STATES*) – **listing 6**. Stan *APP_LED_ON* ma za zadanie zaświecenie diody LED. Jeśli uaktywnimy stan *APP_NOP*, to nasza aplikacja nie będzie nic robiła. W kolejnym kroku modyfikujemy funkcję *APP_Tasks* dodając kolejne dwa stany, jak na **listingu 7**.

W trakcie inicjalizacji aplikacji jest wprowadzany stan *APP_STATE_INIT*, w którym ustawiamy linie RD0 (*BSP_LED_1*) i RD1 (*BSP_LED_2*) jako wyjścia i na końcu wybieramy stan *APP_LED_ON*. W następnym cyklu poolingu jest aktywny stan *APP_LED_ON*

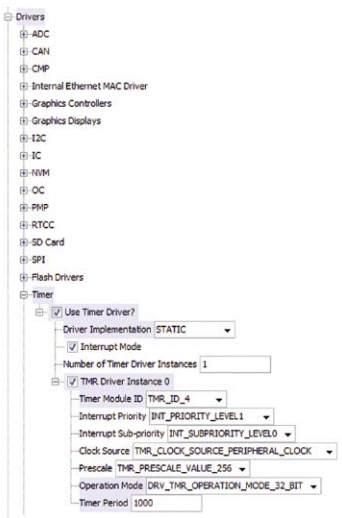
i jest wykonywane ustawianie linii RD0 i RD1 przez funkcję *SYS_PORTS_PinSet*. Po tej czynności jest wprowadzany stan *APP_NOP*. Ponieważ w „obsłudze" tego stanu nie zmiany na inny, to maszyna pozostaje w stanie *APP_NOP* w nieskończoność.

Biblioteka Harmony zawiera wiele funkcji obsługujących układ portów, od konfiguracji kierunku przesyłania danych, poprzez ustawienia typu PULLUP, do czytania stanów linii wejściowych i sterowania liniami wyjściowymi. Dla tak nieskomplikowanych zadań, jak na przykład ustawienie lub zerowanie linii portu, funkcja drivera portu wywołuje tylko funkcję biblioteczną układów peryferyjnych PLIB (**listing 8**).

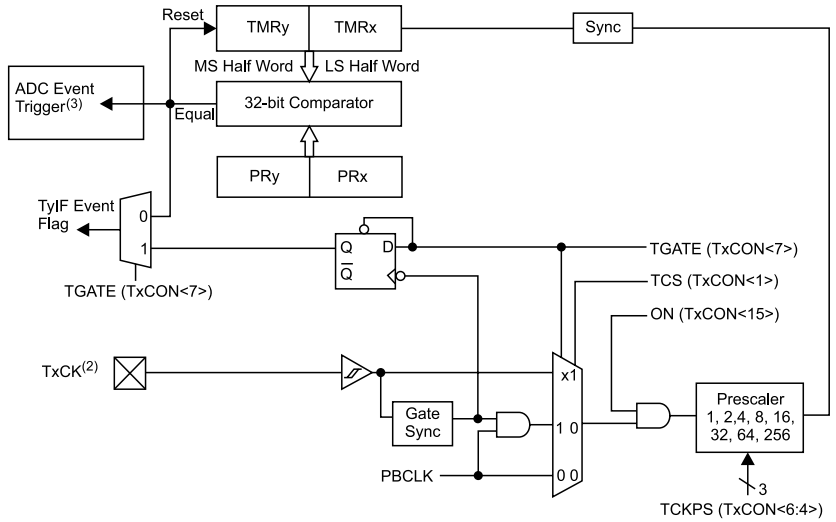
Przykład 2 – migająca dioda LED

Po zaświeceniu diody LED spróbujemy teraz zmodyfikować nasz program, aby dioda zaczęła migać. Aby to zrealizować, trzeba będzie odliczać opóźnienia. Można to zrobić programowo, ale lepiej wykorzystać układ licznikowy.

Mikrokontroler PI32MX795F512L ma wbudowanych 5 układów czasowo licznikowych Timer1….Timer5. Wszystkie liczniki są 16-bitowe, ale układy Timer2 i Timer3, oraz Timer4 i Timer5 można łączyć w pary tworząc liczniki 32-bitowe. My użyjemy 32-bitowego licznika Timer4/5. Jego schemat blokowy pokazano na **rysunku 12**. Jak każdy układ peryferyjny, Timer 4/5 musi przed użyciem zostać skonfigurowany. W tym celu programista musi sięgnąć do opisu rejestrów konfiguracyjnych SFR. W MPLAB Harmony konfiguracja Timera i drivera urządzenia może być przeprowadzona z poziomu wtyczki MHC. Otwieramy: *Harmony Framework Configuration* → *Drivers* i zaznaczamy *Use Timer Driver* (**rysunek 13**). W tym momencie musimy się zdecydować czy będzie to driver statyczny, czy dynamiczny. Ponieważ aplikacja jest bardzo prosta, wybieramy opcję *STATIC*. Kolejne ustawienie dotyczy przerwań. Po zaznaczeniu *Interrupt Mode* MHC



Rysunek 13. Konfiguracja drivera Timera



Rysunek 12. Schemat blokowy licznika Timer4/5

Listing 8. Funkcja SYS_PORTS_PinSet

```
bitPos );
} void SYS_PORTS_PinSet( PORTS_MODULE_ID index,
                         PORTS_CHANNEL channel,
                         PORTS_BIT_POS bitPos )
{
    PLIB_PORTS_PinSet( index, channel,
```

Listing 9. Procedura obsługi przerwania od Timer4

```
#include <xc.h>
#include <sys/attribs.h>
#include „app.h"
#include „system_definitions.h"
//*******************************************************
// Section: System Interrupt Vector Functions
//*******************************************************
void __ISR(_TIMER_5_VECTOR, ipl1AUTO) _IntHandlerDrvTmrInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_5);
}
```

wygeneruje pliki z funkcją do obsługi przerwania timera oraz skonfiguruje przerwania. Domyślnie funkcja obsługi przerwania tylko zeruje flagę przerwania. Użytkownik musi sam dodać swoje procedury.

Konfiguracja samego drivera Timera polega na:

- Wybraniu ID Timera. Ponieważ chcemy użyć 32-bitowego licznika złożonego z liczników Timer4/5, to wybieramy *TMR_ID_4*.
- Określeniu priorytetu i subpriorytetu przerwania.
- Wybraniu źródła zliczanych impulsów (w naszym przypadku PBCLK – zegar taktujący peryferiami).
- Wartość podziału preskalera.
- Konfiguracja licznika 16/32 bity – tu 32 bity.
- Ilość impulsów wpisywanych do rejestru PR. Po odliczeniu tych impulsów licznik jest zerowany, zgłasza przerwanie i odliczanie rozpoczyna się na nowo.

Samo konfigurowanie jest proste i intuicyjne. Eliminuje konieczność żmudnego wczytywania się w zawartość rejestrów konfiguracyjnych. Po zapisaniu zamian i wygenerowaniu plików w MPLAB Harmony Configurator można używać funkcji drivera DRV_TMR0.

Wróćmy, do naszego zadania, czyli do migania diodą LED. Mamy teraz do dyspozycji dwa mechanizmy: maszynę stanów i przerwania. Zacznijmy od prostszej w realizacji, czyli od przerwań. Prostszej, bo konfigurator skonfigurował nam licznik i system przerwań. Procedura obsługi przerwania jest umieszczana w pliku *system_interrupt.c* (katalog *Source Files/app/system_config/default*), a funkcje drivera w pliku *drv_tmr_static.c* (katalog *Source Files/app/system_config/default/frmework/driver/tmr/src*). Na **listingu 9** pokazano procedurę przerwania wygenerowana przez MHC.

Załóżmy, że dioda ma być zaświecona i zgaszona przez 1 sekundę. Przerwanie będzie zgłaszane też co 1 sekundę. Częstotliwość zegara PBCLK ustawiono na 80 MHz, a preskaler na 256, więc na wejściu licznika będzie przebieg o częstotliwości

Listing 10. Zmodyfikowana procedura obsługi przerwania

```
void __ISR(_TIMER_5_VECTOR, ipl1AUTO) _IntHandlerDrvTmrInstance0(void)
{
    BSP_LEDToggle( BSP_LED_2);//zmiana stanu diody LED
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_5);  //zerowanie flagi
przerwania
}
```

Listing 11. Fragment procedury APP_Tasks uzupełnionej o inicjalizację zliczania układu drivera licznika

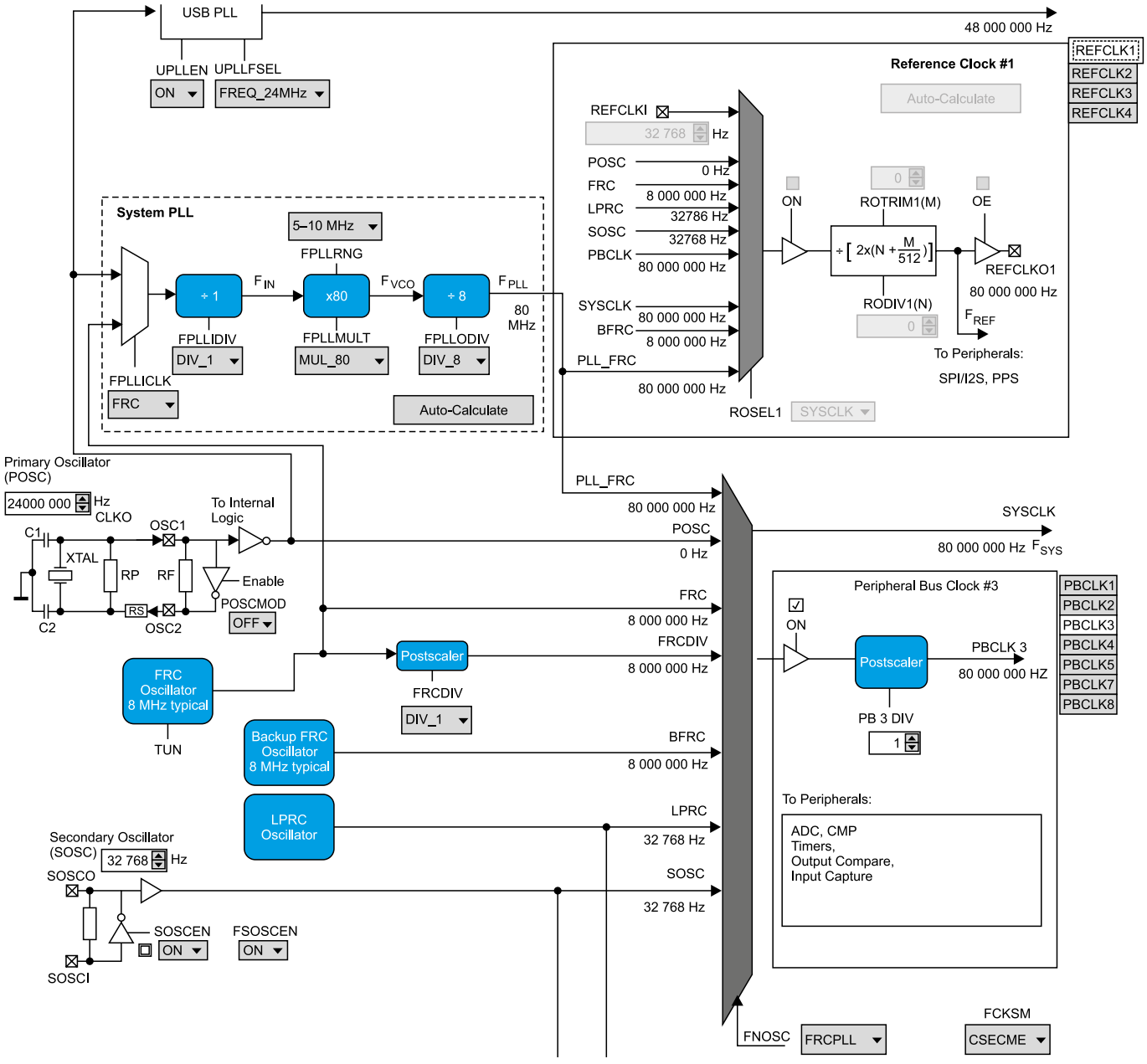
```
void APP_Tasks ( void )
{
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_1);
            SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_2);
            DRV_TMR0_Start();//uruchom licznik
            DRV_TMR0_CounterClear();//zeruj rejestr licznika
            appData.state=APP_LED_ON;
             break;
        }
```

312,5 kHz. Zatem do rejestru PR4 (*Time Period* w konfiguracji drivera w MHC) trzeba wpisać liczbę 312500. Ponieważ licznik jest 32-bitowy, to nie będzie z tym problemu. Procedurę obsługi przerwania uzupełniamy o funkcję *BSP_LEDToggle(BSP_LED_2)* negującą przy każdym wywołaniu poziom logiczny linii sterującej diodę LED2 modułu PIC32 USB Starter Kit II (**listing 10**). Aby całość działała poprawnie, trzeba jeszcze włączyć zliczanie i ewentualnie wyzerować rejestry licznika. Można to zrobić w procedurze *APP_Tasks* w aktywnym stanie *APP_STATE_INIT* (**listing 11**).

Statyczna implementacja drivera timera dostarcza tylko kilku podstawowych funkcji:

- DRV_TMR0_Start – rozpoczęcie zliczania.
- DRV_TMR0_Stop – zatrzymanie zliczania.
- DRV_TMR0_CounterClear – wyzerowanie licznika.
- DRV_TMR0_CounterValueSet – zapisanie licznika wartością początkową.
- DRV_TMR0_CounterValueGet – odczytanie wartości licznika.
- DRV_TMR0_Initialize – inicjalizacja timera.

Wykorzystując te funkcje można implementować nieblokujące opóźnienia w maszynie stanów. Ja do celów testowych zaimplementowałem maszynę 2-stanową: APP_LED_ON i APP_LED_OFF. Wykorzystałem tu też mechanizm przerwania do zmiany na wartość przeciwną najmłodszego bitu



Rysunek 14. Konfiguracja taktowania mikrokontrolera PIC32MZ

zmiennej (toogle bit). Wartość tego bitu jest testowana w maszynie stanów i na jej podstawie jest cyklicznie zaświecana lub gaszona dioda LED (**listing 12**).

Po zainicjowaniu systemu maszyna stanów przyjmuje stan *APP_LED_ON*. Kiedy ten stan jest aktywny, to jest sprawdzana wartość najmłodszego bitu zmiennej *led*. Jeżeli ten bit jest jedynką, to maszyna stanów ustawia linie, do których przyłączono diody LED_1 i LED_2, a potem zmienia stan na APP_LED_OFF. W stanie APP_LED_OFF jest wykonywane sprawdzenie czy najmłodszy bit zmiennej *led* jest wyzerowany. Po wyzerowaniu są zerowane linie sterujące diodami LED_1 i LED_2 oraz zostaje zmieniony stan maszyny na APP_LED_ON. W ten sposób, w takt zmiany wartości zmiennej *led* wykonywanej co 1 sekundę, w obsłudze przerwania od licznika Timer4/5 są zaświecane i gaszone diody LED_1 i LED_2.

Zobaczmy teraz jak wygląda możliwość przeniesienia naszego programu na inny mikrokontroler. Do tego celu zostanie wykorzystany moduł PIC32MZ EC Starter Kit z mikrokontrolerem PIC32MZ2048ECH144. Pracę z nowym mikrokontrolerem zaczynamy od utworzenia nowego projektu, uruchomienia MHC i skonfigurowania układu taktowania. Mikrokontrolery PIC32MZ mogą być taktowane częstotliwością do 200 MHz, ale dla uproszczenia taktowanie będzie się odbywało z taką samą częstotliwością, jak w module z PIC32MX, czyli 80 MHz. Na **rysunku 14** pokazano okno *Mplab Harmony Clock Configuration* dla układu taktowania mikrokontrolerów PIC32MZ. Źródłem sygnału zegarowego został wybrany wewnętrzny oscylator RC o częstotliwości 8 MHz. Ten sygnał jest podawany na układ PLL, gdzie jest potem powielany przez 80 i dzielony przez 8. Taktowanie układów peryferyjnych jest podzielone na 8 grup. Dla każdej z tych grup można zdefiniować różne częstotliwości taktowania. Nas interesuje moduł liczników taktowany przez sygnał PBCLK3. Tu również ustawimy częstotliwość 80 MHz.

W kolejnym kroku trzeba dodać konfigurację drivera Timer4/5 dokładnie tak samo, jak w poprzednim przykładzie oraz konfigurację BSP z modułem PIC32MZ EC Starter Kit. Kiedy te wszystkie czynności zostaną wykonane, to trzeba wygenerować pliki źródłowe (MHC – *Generate*) i zmodyfikować procedurę *APP_Tasks*. Ja po prostu skopiowałem całą funkcję z projektu z mikrokontrolerem PIC32MX. Włączenie opcji wsparcia modułów firmowych BSP spowodowało wygenerowanie definicji linii sterujących diodami LED. Jedyna modyfikacją, którą należy wprowadzić jest definicja portu. Diody LED są sterowane w module PIC32MX USB starter Kit II z portu PORTD, a w module PIC32MZ EC Starter Kit z portu PORTH.

Listing 12. Funkcja maszyny stanów do migania diodami LED

```
void APP_Tasks ( void )
{
    switch ( appData.state )
    {
        /* stan inicjalizacji aplikacji  */
        case APP_STATE_INIT:
        {
           SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_1);
           SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_D ,BSP_LED_2);
           DRV_TMR0_Start();//start zliczania impulsów
           DRV_TMR0_CounterClear();
           appData.state=APP_LED_ON;//zaczynamy do zapalenia diody
            break;
        }
/*testowanie stanu zapalenia diod 1 i 2*/
        case APP_LED_ON :
        {
            led=led&1;//czy zapalić diodę LED
            if(led==1)
            {SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_1);//
zapalenie diod 1i 2
          SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_2);
         appData.state=APP_LED_OFF;}//po zapaleniu sprawdzaj czy zgasić
        }

/*testowanie stanu zgaszenia diod 1 i 2*/
        case APP_LED_OFF :
        {
            led=led&1;
            if(led==0)
            {SYS_PORTS_PinClear(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_1);
          SYS_PORTS_PinClear(PORTS_ID_0,PORT_CHANNEL_D ,BSP_LED_2);
         appData.state=APP_LED_ON;}

        }
        case APP_NOP:
        {
            ;
        }
        /* The default state should never be executed. */
        default:
        {
            /* TODO: Handle error in application's state machine. */
            break;
        }
    }
}
```

Listing 13. Funkcja APP_Tasks dla projektu z PIC32MZ

```
void APP_Tasks (void)
{
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
           SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_H ,BSP_LED_1);
           SYS_PORTS_PinDirectionSelect(PORTS_ID_0, SYS_PORTS_DIRECTION_
OUTPUT, PORT_CHANNEL_H ,BSP_LED_2);
           DRV_TMR0_Start();
           DRV_TMR0_CounterClear();
           appData.state=APP_LED_ON;
            break;
        }
        case APP_LED_ON :
        {
            led=led&1;
            if(led==1)
            {SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_H ,BSP_LED_1);
          SYS_PORTS_PinSet(PORTS_ID_0,PORT_CHANNEL_H ,BSP_LED_2);
         appData.state=APP_LED_OFF;}
        }
        case APP_LED_OFF :
        {
            led=led&1;
            if(led==0)
            {SYS_PORTS_PinClear(PORTS_ID_0,PORT_CHANNEL_H ,BSP_LED_1);
          SYS_PORTS_PinClear(PORTS_ID_0,PORT_CHANNEL_H ,BSP_LED_2);
         appData.state=APP_LED_ON;}
        }
        case APP_NOP:
        {
            ;
        }
        /* The default state should never be executed. */
        default:
        {
            /* TODO: Handle error in application's state machine. */
            break;
        }
    }
}
```

Po skompilowaniu i przesłaniu kodu wynikowego do mikrokontrolera program działa, jak w wypadku PIC32MX (**listing 13**). Jak widać, przy użyciu MPALB Harmony i funkcji driverów urządzeń przenoszenie programów na rodzinę mikrokontrolerów z innym rdzeniem nie jest trudne i można je wykonać szybko, i bez większych problemów. Bardzo pomocna jest możliwość szybkiego skonfigurowania dość skomplikowanego układu taktowania.

Tomasz Jabłoński, EP