MPLAB Harmony – biblioteka graficzna

W pierwszej części artykułu opisującego zestaw bibliotek MPLAB Harmony pokazałem przykład z migającą diodą LED uruchomiony na module ewaluacyjnym PIC32 USB Starter Kit II. Wykonywanie na zaawansowanym 32 bitowym mikrokontrolerze tylko tak nieskomplikowanego działania w praktyce nie ma zastosowania. Zestaw narzędzi w postaci środowiska MPLABX IDE, kompilatora MPLAC XC32 i bibliotek MPLAB Harmony jest przeznaczony do tworzenia o wiele bardziej zaawansowanych aplikacji. Jednym z bardziej wymagających pod względem programowym i niezbędnych zasobów w jest implementacja interfejsu dotykowego z graficznym, kolorowym wyświetlaczem LCD.

Firma Microchip dostarcza bezpłatną bibliotekę graficzną przeznaczoną dla mikrokontrolerów z rodzin PIC32 i PIC24. Aby ułatwić projektowanie ekranów interfejsu, do pakietu MPLAB X IDE dostarczono wtyczkę **Graphic Display Designer X** (GDDX), która umożliwia umieszczanie na projektowanych ekranach graficznych obiektów (widżetów), definiowanie interakcji pomiędzy tymi elementami i generowanie kodu wynikowego dla tworzonego projektu.

Biblioteka i wtyczka GDDX są przez Microchipa stale rozwijane. Początkowo – do wersji 2.0 GDDX – nie było możliwości integracji z pakietem Harmony. Wraz z pojawieniem się Harmony powstała nowa wersja GDDX 2.0 i kolejne. Niestety, wszystkie projekty tworzone we wcześniejszych wersjach nie były kompatybilne z projektami przeznaczonymi do uruchamiania z wykorzystaniem MPALB Harmony. Tak było do momentu wprowadzenia najnowszej wersji pakietu IDE – **MPALB X IDE 3.0**. Zupełnie zrezygnowano w niej z wtyczki GDDX, a funkcję środowiska projektowego przeznaczonego do projektowania ekranów graficznych przejęło **MPLAB Harmony Configurator** (MHC). Podobnie jak w poprzednio, projekty tworzone za pomocą GDDX 2.xx nie mogą być otwierane przez MHC. To oczywiście frustrująca sytuacja dla wszystkich tych, którzy projektują interfejsy graficzne i chcieliby korzystać z kolejnych wersji biblioteki. Takie są konsekwencje korzystania z bezpłatnych narzędzi. Z drugiej strony jednak to dobra wiadomość, bo firma nadal rozwija biblioteki i pewnie w jakimś momencie tez zawirowania się skończą, a przecież stale są dostępne starsze wersje z MPALB X 2.35 i GDDX 2.xx.

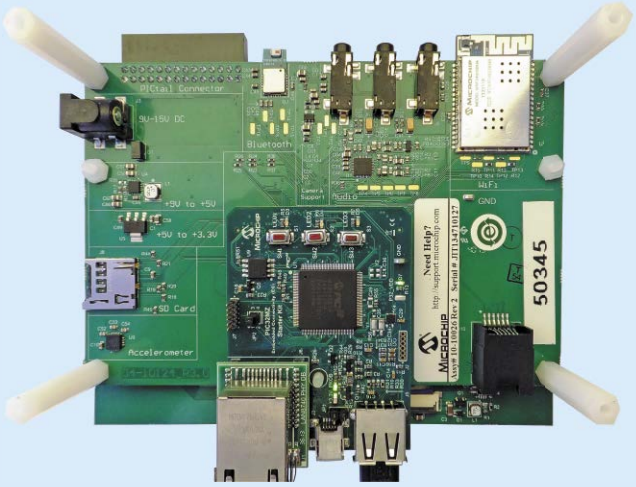
W momencie pisania tego artykułu możliwości biblioteki konfigurowanej prze MHC były nieco okrojone w porównaniu do ostatniej wersji GDDX, ale według zapewnień producenta ma się to zmienić w nowszych wersjach IDE.

Oprogramowanie dla mikrokontrolerów PIC32MX i PIC32MX tworzone z wykorzystaniem MPLAB Harmony raczej jest rozbudowane i skomplikowane. Użycie biblioteki graficznej na pewno nie upraszcza sprawy. Z tego powodu, przy pierwszym kontakcie z nowymi rozwiązaniami najlepiej jest użyć sprawdzonych i działających przykładów. Do najnowszego MPLAB Harmony jest dołączonych wiele przykładów, w tym przykłady wykorzystania biblioteki graficznej.

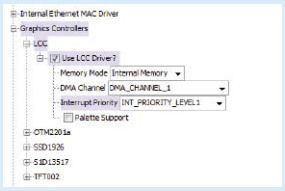
Oprócz rozwiązań programowych będzie nam potrzeby też sprzęt do testowania. Dzięki polskiemu biuru firmy Microchip miałem możliwość użycia zestawu **PIC32 Multimedia Expansion Board (MEB) II** i współpracującego z nim modułu **PIC32MZ Starter Kit** (**fotografia 1**).

Zestaw MEBII jest bogato wyposażony w układy peryferyjne, między innymi w: kamerę VGA, 24-bitowy kodek audio, moduły Wi-Fi i Bluetooth, akcelerometr, czujnik temperatury i inne. Dla potrzeb tego przykładu przyda się kolorowy wyświetlacz LCD o przekątnej 4,3 cala, rozdzielczości WQVGA, z pojemnościowym panelem dotykowym. Wyświetlacz nie ma wbudowanego sterownika. Przy okazji opisywania biblioteki graficznej Microchip wspomniałem, że wspiera ona obsługę kilku popularnych sterowników paneli LCD, ale też oferuje ma funkcjonalność sterownika LCD realizowanego przez mikrokontroler. To ostatnie rozwiązanie ma szereg zalet. Po pierwsze, nie musimy szukać wyświetlacza ze wspieranym sterownikiem lub samodzielnie tworzyć procedur obsługi. Po drugie, panele bez sterownika są tańsze. Jednak nie ma róży bez kolców – implementacja sterownika zajmuje zasoby.

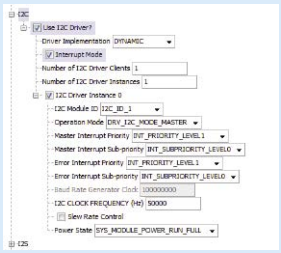
Pojemnościowy panel dotykowy jest obsługiwany przez sterownik MTCH6301. Komunikacja z mikrokontrolerem odbywa się za pomocą interfejsu I²C.



Fotografia 1. PIC32 Multimedia Expansion Board (MEB) II i moduł PIC32MZ Starter Kit



Rysunek 2. Konfiguracja sterownika LCC



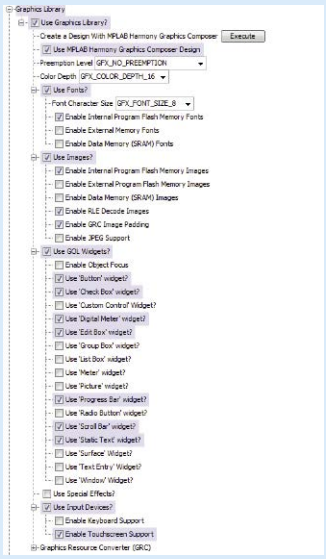
Rysunek 3. Konfiguracja drivera interfejsu I²C

Testowanie zestawu

Testowanie zastawu rozpoczynamy od otworzenia przykładu o nazwie composer umieszczonego w katalogu *harmony/V1_04_02/apps/gfx/composer*. Projekt jest przeznaczony dla mikrokontrolera typu PIC32MZ2048ECL144. W testowanym module zastosowano PIC32MZ2048ECH144, więc w pierwszym kroku należy zmienić typ mikrokontrolera. Aby poprawnie skompilować pliki źródłowe, trzeba użyć kompilatora MPLAB XC32 w wersji v1.34 lub nowszej.

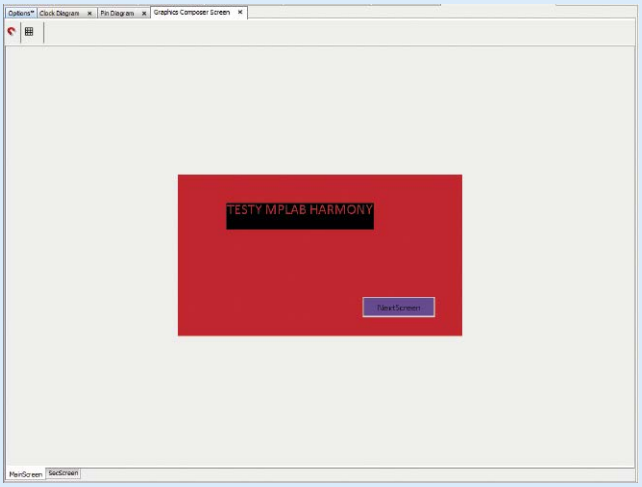
Przykładowy projekt *Composer* w wersji dostarczonej z MPLAB Harmony pozornie nie działa – po skompilowaniu i wgraniu do pamięci mikrokontrolera modułu PIC32MZ Starter Kit na ekranie wyświetlacza nic nie jest wyświetlane. Trzeba otworzyć wtyczkę *MPLAB Harmony Configurator (MHC)* i wczytać konfigurację zapisaną w projekcie z ustawieniami początkowymi:

- *Drivers → Graphics Controllers → LCC*. Opcja *LCC* oznacza sterownik wyświetlacza LCD zaimplementowany w mikrokontrolerze. Po wybraniu LCC możemy ustawić pamięć obrazu zaimplementowaną w mikrokontrolerze (*Internal Memory*) lub w układzie zewnętrznym (*External Memory*), kanał DMA do zapisywania tej pamięci oraz ustalić priorytet przerwań. Konfigurację sterownika grafiki pokazano na **rysunku 2**.
- *Drivers → I²C*. Interfejs I²C jest używany do komunikacji ze sterownikiem panelu dotykowego. Konfigurację I²C pokazano na **rysunku 3**. Warto zwrócić uwagę na to, że zaimplementowano driver dynamiczny i do obsługi magistrali wykorzystuje on przerwania.
- *BSP Configuration*. Jeśli w projekcie wybierzemy mikrokontroler z rodziny PIC32MZ, w zakładce *BSP Configuration* zostaną pokazane wszystkie moduły ewaluacyjne zbudowane w oparciu o wybrany mikrokontroler. Ja użyłem *PIC32MZ EC Starter Kit* i *Mutlimedia Expansion Board (MEB) II*. Po zaznaczeniu tej opcji konfigurator MHC wygeneruje plik **bsp_sys_init.c** zawierający procedury sterowania diodami LED, odczytywania stanu przycisków itd.



Rysunek 4. Menu konfiguracji biblioteki graficznej

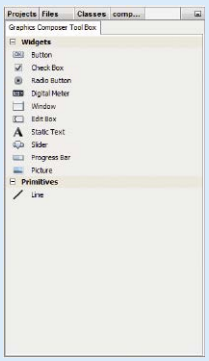
- Taktowanie rdzenia mikrokontrolera przebiegiem o częstotliwości 200 MHz, a układów peryferyjnych – 100 MHz. Taktowanie wybiera się w oknie *Clock Diagram* konfiguratora *MHC*
- Konfiguracja biblioteki graficznej – przy niej zatrzymamy się na dłużej. Konfiguracja biblioteki graficznej jest elementem *Harmony Framework*. Po zaznaczeniu *Use Graphics Library* otrzymujemy możliwość wyboru szeregi opcji (**rysunek 4**). Nas najbardziej będzie interesowała opcja Use MPLAB Harmony Graphics Composer Design. Po jej zaznaczeniu będziemy mogli projektować ekrany interfejsu graficznego w taki sam sposób, jak to się robi za pomocą wtyczki GDDX. Aplikacja *Graphics Composer Design* jest uruchamiana po kliknięciu na przycisk *Execute*. Okno tej aplikacji można podzielić na kilka obszarów:
  - **Graphics Composer Screen** – są w nim wyświetlane projektowane ekrany z umieszczanymi na nich widżetami i tzw. primitives – podstawowymi komponentami graficznymi, takimi jak linie, okręgi itp. Nawigowanie pomiędzy oknami odbywa się poprzez klikanie na zakładki z nazwami ekranów (**rysunek 5**).
  - **Graphics Composer Tool Box** z widżetami i primitives (**rysunek 6**). W aktualnej wersji Composera lista elementów jest uboższa w porównaniu z ostatnią wersją GDDX, ale ma to się zmienić. Elementy z tego okna są wybierane i umieszczane na projektowanych ekranach.
  - **Graphics Composer Properties** jest przeznaczone do zarządzania właściwościami wybranego elementu projektu: ekranu, widżetu itp. (**rysunek 7**). Można tu ustawić na przykład tekst wyświetlany na przycisku, wybrać predefiniowany sposób rysowania przycisku itp.
  - **Graphics Composer Management** przeznaczone do zarządzania komponentami projektu: obiektami, ekranami, schematami i zasobami. Wszystkie te komponenty można wybierać klikając na zakładki w dolnej części okna.



Rysunek 5. Okno edycji ekranów Graphic Composer Screen

Aplikacja przykładowa

Działanie aplikacji wykorzystującej bibliotekę graficzną pokażę na przykładzie. Najpierw zostaną zdefiniowane i nazwane dwa ekrany: *MainScreen* i *SecScreen*. Pierwszy będzie wyświetlany po włączeniu zasilania lub restarcie mikrokontrolera. Umieściłem na nim obiekt *Button* (przycisk) z etykietą *Next Screen* i tekst *„TESTY MPAB HARMONY”*.

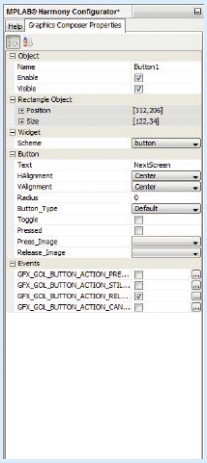


Rysunek 6. Okno Graphics Composer Tool Box

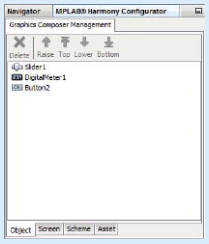
Na drugim ekranie umieściłem przycisk *Return* i obiekty *Slider* i *Digital Meter*. Po naciśnięciu przycisku *Next Screen* aplikacja wyświetla ekran *SecScreen*. Powrót do ekranu głównego następuje po naciśnięciu przycisku *Return*.

Aplikację możemy wygenerować automatycznie z poziomu *MPLAB Harmony Graphic Composer*. Na ekranie *MainScreen* klikamy na przycisk *Next Screen*, a w oknie *Properties* zaznaczamy jedną z akcji:

- **GFX_GOL_BUTTON_ACTION _PRESSED** – przycisk naciśnięty.
- **GFX_GOL_BUTTON_ACTION _STILPRESSED** – przycisk naciśnięty i przytrzymany.
- **GFX_GOL_BUTTON_ACTION _RELEASED** – przycisk zwolniony.
- **GFX_GOL_BUTTON_ACTION _CANCELPRESS** – przyciśnięcie anulowane.



Rysunek 7. Okno właściwości obiektu Button



Rysunek 8. Okno Graphics Component Management

Zaznaczyłem akcję *GFX_GOL_BUTTON_ACTION _RELEASED* wykonywaną po wykryciu zwolnienia przycisku. Po kliknięciu na ikonie z prawej strony zaznaczonej akcji otwiera się okno *Button1 GFX_GOL_BUTTON_ACTION_RELEASED event code generation* (**rysunek 9**).

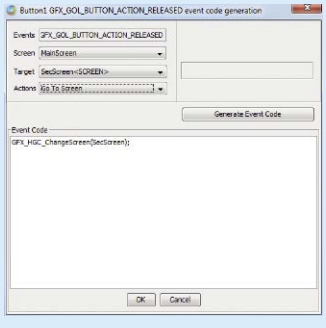
W tym oknie ustawiamy:

- *Screen* tj. ekran, na którym jest umieszczony obiekt (przycisk).
- *Target* tj. miejsce, w którym ma być wykonana akcja.

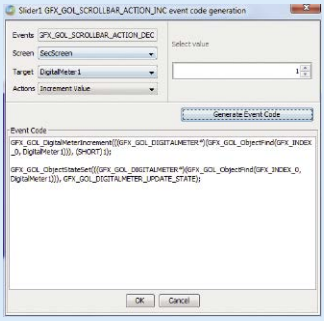
Zakładamy, że po naciśnięciu i zwolnieniu przycisku *Button1* aplikacja przejdzie do ekranu *SecScreen*. Dlatego wybieramy akcję *Go To Screen*. Po kliknięciu na przycisk *Generate Event Code* zostanie wygenerowany kod *GFX_HGC_ChangeScreen(SecScreen)* i automatycznie umieszczony w kodzie aplikacji w takim miejscu, aby po naciśnięciu i zwolnieniu przycisku nastąpiła zmiana ekranu. W tym momencie nie musimy nawet wiedzieć, gdzie ten kod zostanie umieszczony.

W bardzo podobny sposób generujemy kod powrotu z ekranu SecScreen do ekrany głównego po naciśnięciu przycisku *Button2*.

Na ekranie *SecScreen* umieściłem dwa dodatkowe elementy: suwak *Slider* do zadawania wartości i okno do wyświetlania wartości cyfrowych *Digital Meter*. Nasza aplikacja ma teraz za zadanie zmieniać wartość wyświetlaną przez *Digital Meter* w czasie przesuwania suwaku *Slider*. W tym celu klikamy na obiekt *Slider* i w oknie *Properties* wybieramy *Events* → *GFX_GOL_SCROLLBAR_ACTION_INC*. Otwieramy okno *Event Code Generation*. W oknie *Screen* wybieramy *SecScreen*, a w oknie *Target – Digital Meter1*. Przesuwanie suwaka w prawo lub do góry będzie wywoływało akcję w obiekcie *Digital Meter1*.



Rysunek 9. Okno Event Code Generation

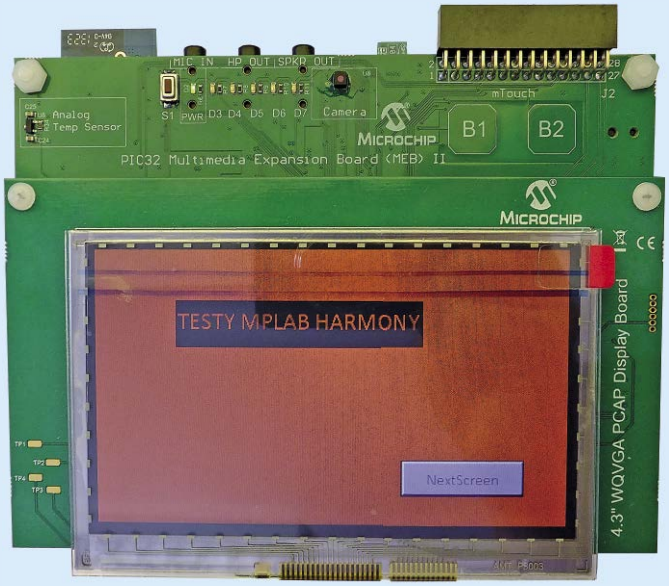


Rysunek 10. Definiowanie akcji zwiększania wartości

W oknie *Actions* można wybrać jedną z akcji: *Set Value*, *Increment Value*, *Decrement Value*, *Show Digital Meter* i *Hide Digital Meter*. Wybieramy *Increment Value*, a w oknie *Select Value* wpisujemy 1 (krok). Po kliknięciu na przycisk *Generate Event Code* zostanie wygenerowany i umieszczony kod akcji (**rysunek 10**). Identycznie postępujemy dla *Events* → *GFX_GOL_SCROLLBAR_ACTION_DEC* z tym, że w oknie *Action* wybieramy *Decrement Value*.

Moduły MPLAB Harmony (drivery urządzeń, usługi systemowe oraz middleware) są implementowane jako maszyna stanu. 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*). Po zainicjowaniu systemu moduły mogą być wywoływane w pętli nieskończonej poprzez odpytywanie (polling), wywoływane w obsłudze przerwania lub pod kontrola systemu RTOS. Metoda pollingu jest najłatwiejsza w implementacji, ale może powodować długi czas odpowiedzi na żądanie wykonania zadania. Dużo bardziej wydajna czasowo jest metoda wywoływania krytycznych sekwencji z wykorzystaniem mechanizmu przerwań, a dodatkowo można ją łączyć z metodą pollingu. W opisywanym przykładzie wykorzystano przerwania do wykrywania działania panelu dotykowego i odświeżania zwartości ekranu.

Na **listingu 1** pokazano funkcje obsługi przerwań używanych do obsługi wyświetlacza. *Funkcja DRV_TOUCH_MTCH6301_ReadRequest();* jest przeznaczona do wysyłania zapytania do podprogramu obsługi I²C, odbierania od niego danych odczytywanych za pomocą I²C ze sterownika MCTH6301 i umieszczania tych danych



Fotografia 11. Ekran MainScreen

Listing 1. Procedury obsługi przerwań
```
void __ISR(_EXTERNAL_1_VECTOR, IPL5AUTO) _IntHandlerExternalInterruptInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_EXTERNAL_1);
    DRV_TOUCH_MTCH6301_ReadRequest(sysObj.drvMtch6301);  //obsługa panelu dotykowego
}

void __ISR(_I2C1_MASTER_VECTOR, ipl1AUTO) _IntHandlerDrvI2CMasterInstance0(void)
{
    DRV_I2C_Tasks(sysObj.drvI2C0);  //obsługa transmisji I2C
}

void __ISR(_I2C1_BUS_VECTOR, ipl1AUTO) _IntHandlerDrvI2CErrorInstance0(void)
{
    SYS_ASSERT(false, „I2C Driver Instance 0 Error”);
}

void __ISR(_DMA0_VECTOR + DMA_CHANNEL_1, ipl1AUTO) _IntHandlerLCCRefresh(void)
{
    SYS_INT_SourceStatusClear(INT_SOURCE_DMA_0 + DMA_CHANNEL_1);
    DRV_GFX_LCC_DisplayRefresh();  //odświeżanie ekranu wyświetlacza
}
```

Listing 2. Funkcja SYS_Tasks();
```
void SYS_Tasks ( void )
{
    /* Maintain system services */
    SYS_DEVCON_Tasks(sysObj.sysDevcon);
    /* Maintain the DMA system state machine. */
    SYS_DMA_Tasks(sysObj.sysDma);
    SYS_MSG_Tasks( (SYS_OBJ_HANDLE) sysObj.sysMsg0 );
    SYS_TOUCH_Tasks(sysObj.sysTouchObject0);
    /* Maintain Device Drivers */
    DRV_TOUCH_MTCH6301_Tasks(sysObj.drvMtch6301);
    /* Maintain Middleware & Other Libraries */
    /* Maintain the gfx state machine. */
    GFX_Tasks(sysObj.gfxObject0);
    /* Maintain HGC generated graphics state machine. */
    GFX_HGC_Tasks(sysObj.gfxObject0);
    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

w kolejce. Potem tymi danymi – umieszczonymi w buforze *drvI2CReadFrameData* – „zajmuje się” procedura *DRV_TOUCH_MTCH_Tasks*.

W każdej aplikacji MPLAB Harmony funkcja *main* zawiera pętlę nieskończoną wywołującą cyklicznie funkcję *SYS_Tasks();*, jak pokazano na **listingu 2**.

Do obsługi ekranu dotykowego jest przeznaczona zamieszczona na **listingu 3** funkcja *DRV_TOUCH_MTCH6301_Tasks ();*. Przy konfigurowaniu akcji w *Graphics Composer* przypisaliśmy na przykład przyciskowi *Button1* funkcjonalność zmiany ekranu z *MainScreen* na *SecScreen*. Composer wygenerował kod i umieścił go w programie, więc rzeczywiście naciśnięcie

Listing 3. Obsługa ekranu dotykowego
```
void DRV_TOUCH_MTCH6301_Tasks ( SYS_MODULE_OBJ object )
{
    static int32_t  taskIndex = 0;
    uint8_t  touchpoint = 0;
    int16_t lastX;
    int16_t lastY;
    DRV_TOUCH_MTCH6301_OBJECT * pDrvObject = (DRV_TOUCH_MTCH6301_OBJECT *)object;
    if ( object == SYS_MODULE_OBJ_INVALID )
    {
        return;
    }
    if( pDrvObject->readRequest == 0)
    {
        return;
    }
    while(taskIndex < pDrvObject->readRequest)
    {
        if( pDrvObject->taskQueue[taskIndex].inUse == false )
        {
            return;
        }
        if( pDrvObject->taskQueue[taskIndex].taskState ==
            DRV_TOUCH_MTCH6301_TASK_STATE_INIT ||
            pDrvObject->taskQueue[taskIndex].taskState ==
            DRV_TOUCH_MTCH6301_TASK_STATE_DONE )
        {
            return;
        }
        if( pDrvObject->taskQueue[taskIndex].taskState
            == DRV_TOUCH_MTCH6301_TASK_STATE_DECODE_INPUT )
        {
            if( !(DRV_I2C_BUFFER_EVENT_COMPLETE &
            DRV_I2C_BufferStatus(pDrvObject->taskQueue[taskIndex].drvI2CReadBufferHandle)))
            {
                return;
            }
            if( (pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x01)&&
                !(pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x40) )
            {
                touchpoint = (pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x78)>>3;
                lastX  = pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[3]&0x7F;
                lastX |= (uint16_t)(( pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[4]&0x1F )<<7);
                lastY = pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[5]&0x7F;
                lastY |= (uint16_t)(( pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[6]&0x1F )<<7);
                if ( pDrvObject->orientation == 180 )
                {
                    PCapX[touchpoint] = pDrvObject->horizontalResolution\\
                    - (( lastX * pDrvObject->horizontalResolution ) >> 10);
                    PCapY[touchpoint] = pDrvObject->verticalResolution\\
                    - (( lastY * pDrvObject->verticalResolution ) >> 10);
                } else if ( pDrvObject->orientation == 90 )
                {
                    PCapX[touchpoint] = pDrvObject->verticalResolution\\
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 87550, pass: rxoagj8

Listing 3. c.d.

```
                    - (( lastY * pDrvObject->verticalResolution ) >> 10);
                    PCapY[touchpoint] = (( lastX * pDrvObject->horizontalResolution ) >> 10);
                } else if ( pDrvObject->orientation == 270 )
                {
                    PCapX[touchpoint] = (( lastY * pDrvObject->verticalResolution ) >> 10);
                    PCapY[touchpoint] = pDrvObject->horizontalResolution\\
                    - (( lastX * pDrvObject->horizontalResolution ) >> 10);
                } else
                {
                    PCapX[touchpoint] = ( lastX * pDrvObject->horizontalResolution ) >> 10;
                    PCapY[touchpoint] = ( lastY * pDrvObject->verticalResolution ) >> 10;
                }
            }
            if(!(pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x01)
                 && !(pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x40) )
            {
                touchpoint = (pDrvObject->taskQueue[taskIndex].drvI2CReadFrameData[2] & 0x78)>>3;
                PCapX[touchpoint] = -1;
                PCapY[touchpoint] = -1;
            }
            pDrvObject->taskQueue[taskIndex].taskState = DRV_TOUCH_MTCH6301_TASK_STATE_DONE;
            pDrvObject->taskQueue[taskIndex].inUse = false;
            taskIndex++;
        }
    }
    pDrvObject->readRequest = 0;
    taskIndex = 0;
    return;
}
```

Listing 4. Funkcja GFX_HGC_MagButtons

```
bool GFX_HGC_MsgButtons(uint16_t objMsg, GFX_GOL_OBJ_HEADER *pObj)
{
    switch (GFX_GOL_ObjectIDGet(pObj))
    {
        case Button1:
            if (objMsg == GFX_GOL_BUTTON_ACTION_PRESSED)
            {
                // Button Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_STILLPRESSED)
            {
                // Button Still Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_CANCELPRESS)
            {
                // Button Cancel Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_RELEASED)
            {
                // Button Release Event Code
                GFX_HGC_ChangeScreen(SecScreen);
            }
            return true;
        case Button2:
            if (objMsg == GFX_GOL_BUTTON_ACTION_PRESSED)
            {
                // Button Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_STILLPRESSED)
            {
                // Button Still Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_CANCELPRESS)
            {
                // Button Cancel Pressed Event Code
                //No events defined from HGC
            }
            if (objMsg == GFX_GOL_BUTTON_ACTION_RELEASED)
            {
                // Button Release Event Code
                GFX_HGC_ChangeScreen(MainScreen);
            }
            return true;
        default:
            return false; // process by default
    }
    return true;
}
```

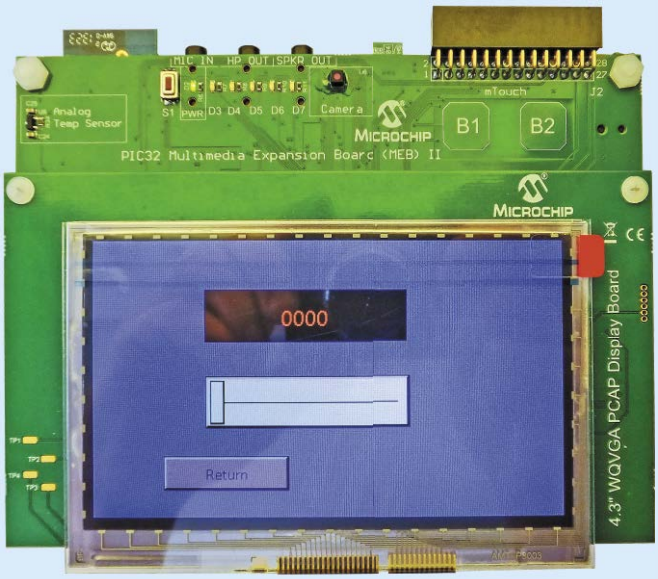
Listing 5. Funkcja GFX_HGC_MsgDigitalMeters

```
 bool GFX_HGC_MsgDigitalMeters(uint16_t objMsg, GFX_GOL_OBJ_HEADER *pObj)
{
    switch (GFX_GOL_ObjectIDGet(pObj))
    {
        case DigitalMeter1:
            if (objMsg == GFX_GOL_DIGITALMETER_ACTION_SELECTED)
            {
                                // Digital Meter select Event Code
            //No events defined from HGC
            }

            return true;
        default:
            return false; // default false as it is not processed
    }
}
```

Listing 6 Funkcja *GFX_HGC_MsgScrollBars*

```
bool GFX_HGC_MsgScrollBars(uint16_t objMsg, GFX_GOL_OBJ_HEADER *pObj)
{
    switch (GFX_GOL_ObjectIDGet(pObj))
    {
        case Slider1:
            if (objMsg == GFX_GOL_SCROLLBAR_ACTION_INC)
            {
            // scrollbar increment Event Code
                GFX_GOL_DigitalMeterIncrement(((GFX_GOL_DIGITALMETER*)\\
                (GFX_GOL_ObjectFind(GFX_INDEX_0, DigitalMeter1))), (SHORT)1);
                GFX_GOL_ObjectStateSet(((GFX_GOL_DIGITALMETER*)\\
               (GFX_GOL_ObjectFind(GFX_INDEX_0, DigitalMeter1))), GFX_GOL_DIGITALMETER_UPDATE_
STATE);
            }
            if (objMsg == GFX_GOL_SCROLLBAR_ACTION_DEC)
            {
                // Scrollbar decrement Event Code
                GFX_GOL_DigitalMeterDecrement(((GFX_GOL_DIGITALMETER*)\\
                (GFX_GOL_ObjectFind(GFX_INDEX_0, DigitalMeter1))), (SHORT)1);
                GFX_GOL_ObjectStateSet(((GFX_GOL_DIGITALMETER*)\\
                (GFX_GOL_ObjectFind(GFX_INDEX_0, DigitalMeter1))), GFX_GOL_DIGITALMETER_UPDATE_
STATE);
            }
            return true;
        default:
            return false; // default false as it is not processed
    }
}
```



Fotografia 12. Ekran SecScreen

przycisku powoduje zmianę ekranu. Obsługa zdarzenia od akcji przypisanej do przycisku jest wykonywana przez pokazaną na **listingu 4** funkcję *GFX_HGC_MagButtons()*. Ta funkcja poprzez szereg innych funkcji jest również wywoływana z *SYS_Tasks()*. Dla każdej z akcji użytkownik może dopisać swój własny kod. Composer umieścił tu automatycznie wywołania *GFX_HGC_ChangeScreen(SecScreen);* dla akcji zwolnienia przycisku *Button1* oraz *GFX_HGC_ChangeScreen(MainScreen)* dla akcji zwolnienia przycisku *Button2*. Aktywne akcje są ustawiane przez driver obsługi panelu dotykowego.

Dla obiektów *Digital Meter* i *Scroll Bars* oprogramowanie *Graphics Composer* wygenerowało odpowiednie funkcje: *GFX_HGC_MsgDigitalMeters* (**listing 5**) oraz *GFX_HGC_MsgScrollBars* (**listing 6**). W funkcji *GFX_HGC_MsgScrollBars* kreator umieścił wywołania funkcji odpowiedzianych za zmianę wyświetlanej wartości w obiekcie *Digital Meter*. Ekrany wyświetlane po skompilowaniu programu wygenerowanego przez MHC zostały pokazane na **fotografiach 11** i **12**.

Tomasz Jabłoński, EP