

Oprogramowanie Systemowe

Tetris w EFI - część implementacyjna

Spis treści

Elementy kodu specyficznego dla EFI zawarte w programie.	2
DebugLib.h.....	2
ASSERT_EFI_ERROR.....	2
ASSERT.....	2
Uefi.h.....	2
Punkt wejścia do aplikacji.....	2
Przygotowanie konsoli.....	3
UefiLib.h.....	3
Obsługa wyjścia na ekran w trybie tekstowym.	3
MemoryAllocationLib.h.....	3
AllocatePool i Free Pool.....	3
BaseMemoryLib.h.....	4
CopyMem.....	4
ZeroMem.....	4
UefiBootServicesTableLib.h.....	4
Obsługa timera.	4
Obsługa wejścia z klawiatury.....	5
Ustawianie pozycji kursora i atrybutów tekstu.	5
UefiRuntimeServicesTableLib.h.....	5
Pobranie aktualnego czasu.....	5
Obsługa przerwań w EFI.	6
Jak zbudować wersję 64 bitową aplikacji.	7

Elementy kodu specyficznego dla EFI zawarte w programie.

Zostały posegregowane na podstawie plików nagłówkowych (bibliotek), które należy dołączyć, aby wykorzystywać daną funkcję.

Każda z wykorzystanych bibliotek wymaga dodania w pliku .inf aplikacji (w naszym przypadku jest to tetris.inf) odpowiedniego wpisu w sekcji LibraryClasses. Struktura i lokalizacja pliku .inf są opisane w pierwszym sprawozdaniu.

```
[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib
  UefiBootServicesTableLib
  MemoryAllocationLib
  BaseMemoryLib
```

DebugLib.h

ASSERT_EFI_ERROR

```
ASSERT_EFI_ERROR( status );
```

Makro, które sprawdza czy podana do niego wartość jest inna niż EFI_SUCCESS i jeżeli jest to prawdą to wywołuje funkcję __debugbreak(), powodującą przerwanie wykonania programu. Jeżeli aplikacja nie jest budowana w trybie debug, to nie zostanie ono wykonane.

ASSERT

```
ASSERT( FALSE );
```

Odpowiednik makra ASSERT z języka C.

Uefi.h

Punkt wejścia do aplikacji.

```
EFI_STATUS EFIAPI UefiMain( IN EFI_HANDLE ImageHandle,
                           IN EFI_SYSTEM_TABLE *SystemTable ) {

    return EFI_SUCCESS;
}
```

W naszym programie wykorzystaliśmy punkt wejścia nie uwzględniający argumentów wejściowych *UefiApplicationEntryPoint*, możliwe jest jednak wykorzystanie *ShellCEntryLib*, który pozwala na przekazywanie argumentów wywołania do aplikacji tak jak w 'zwykłym' C.

EFI_STATUS jest zmienną typu int, wykorzystywaną przez większość funkcji EFI do zwracania kodu powrotu z funkcji.

EFIAPI jest makrem, które określa wykorzystywany *calling convention*, w naszym przypadku przyjmuje ono wartość `__cdecl`.

ImageHandle jest wskaźnikiem na kontekst sprzętowy związany z wywołaną aplikacją, jego wartość jest potrzebna przy wywołaniach niektórych funkcji.¹

SystemTable wskazuje na globalną tablicę systemową EFI, dającą nam dostęp do wskaźników na potrzebne nam funkcje oraz do innych podtablic² (jej globalnym odpowiednikiem jest gST z *UefiBootServicesTableLib.h*).

¹ <http://mjg59.dreamwidth.org/18773.html?thread=767573> (dostęp: 30.01.2015)

² Tamże.

Przygotowanie konsoli.

```
void prepareConsole( IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL* cout,
                    OUT EFI_SIMPLE_TEXT_OUTPUT_MODE* modeToStore ) {
    EFI_STATUS status;
    CopyMem( modeToStore, cout->Mode, sizeof( EFI_SIMPLE_TEXT_OUTPUT_MODE ) );

    status = cout->EnableCursor( cout, FALSE );
    if ( status != EFI_UNSUPPORTED ) { // workaround
        ASSERT_EFI_ERROR( status );
    }

    status = cout->ClearScreen( cout );
    status = cout->SetAttribute(cout, EFI_TEXT_ATTR(EFI_LIGHTGRAY,EFI_BLACK));
    status = cout->SetCursorPosition(cout, 0, 0);
}
```

Pierwszy parametr to wskaźnik na strukturę opisującą protokół obsługujący ekran konsoli, drugi wskazuje na strukturę opisującą atrybuty tegoż ekranu. Rozpoczynamy od zapisania aktualnych atrybutów, tak abyśmy mogli je przywrócić gdy nasza aplikacja zakończy działanie. Następnie czynimy kursor niewidocznym, w tym miejscu pomimo poprawnego wykonania operacji zwracany jest kod błędu, zastosowaliśmy więc obejście. W dalszej kolejności czyścimy ekran, ustawiamy odpowiednie atrybuty wypisywanego tekstu i pozycję kursora.

```
void restoreInitialConsoleMode( IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL* cout,
                               IN EFI_SIMPLE_TEXT_OUTPUT_MODE* storedMode ) {
    EFI_STATUS status;

    status = cout->EnableCursor( cout, storedMode->CursorVisible );
    status = cout->SetCursorPosition(cout, storedMode->CursorColumn,
                                     storedMode->CursorRow);
    status = cout->SetAttribute(cout, storedMode->Attribute);
    status = cout->ClearScreen(cout);
}
```

Przy kończeniu pracy aplikacji musimy przywrócić początkowy stan ekranu konsoli. W tym celu korzystamy z wcześniej zapisanych wartości, ustawiając je ponownie.

UefiLib.h

Obsługa wyjścia na ekran w trybie tekstowym.

```
Print( L" PRESS ANY KEY TO START " );
```

Funkcja Print jest wygodnym *wrapperem* na wywołanie:

```
gST->ConOut->OutputString( gST->ConOut, /* ... */ );
```

Jako argument przyjmuje ona łańcuch znaków zapisany w Unicode. Efektem jej wywołania jest wypisanie na ekran konsoli, zaczynając od aktualnej pozycji kursora, tekstu przy użyciu aktualnych atrybutów.

MemoryAllocationLib.h

AllocatePool i Free Pool

```
Piece* piece = AllocatePool( sizeof( Piece ) );
FreePool( this );
```

Są to odpowiedniki funkcji *malloc* i *free* z biblioteki standardowej języka C.

BaseMemoryLib.h

CopyMem

```
CopyMem( &this->bodies, target, sizeof( Bodies ) );
```

Odpowiednik funkcji memcpy z języka C.

ZeroMem

```
ZeroMem( piece, sizeof( Piece ) );
```

Ustawia ilość bajtów podaną jako drugi argument w obszarze wskazywanym przez pierwszy argument na zero.

UefiBootServicesTableLib.h

Obsługa timera.

Musimy przechowywać strukturę opisującą Event związany z timerem i przy starcie aplikacji utworzyć odpowiedni Event i przypisać go do wspomnianej wcześniej struktury. Dokładny opis poniższej funkcji można znaleźć w dokumentacji.³

```
// set up timer event
gBS->CreateEventEx( EVT_TIMER | EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
                  timerCallback, core, NULL, &core->timerEvent );
```

EVT_TIMER i **EVT_NOTIFY_SIGNAL** określają typ zdarzenia.

TPL_CALLBACK określa priorytet wykonywanej operacji.⁴

timerEvent to zmienna typu **EFI_EVENT** zdefiniowana w klasie **Core**.

Po utworzeniu odpowiedniej struktury można wywołać funkcję rozpoczynającą pracę zegara.

```
// start the timer
status = gBS->SetTimer( core->timerEvent, TimerPeriodic, TIMER_PERIOD );
```

TimerPeriodic to wartość całkowita określająca sposób działania zegara, zdefiniowana w **UefiSpec.h**.

TIMER_PERIOD to stała zdefiniowana w pliku **Core.h** oznaczająca okres timera (jak często będzie wywoływany callback).

timerCallback jest definiowaną przez nas funkcją, która jest wywoływana za każdym razem gdy dobiegnie końca pojedynczy cykl zdefiniowanego przez nas timera.

```
void timerCallback( EFI_EVENT event, void* context ) {
    //...
}
```

Jeżeli timer nie jest już potrzebny to należy zakończyć związany z nim Event, jest to wykonywane w destruktorze klasy **Core**.

```
gBS->CloseEvent( this->timerEvent );
```

³ http://www.uefi.org/sites/default/files/resources/2_4_Errata_B.pdf, s. 122 (dostęp: 30.01.2015)

⁴ Tamże, s. 115

Obsługa wejścia z klawiatury

```
void handleInput( Core* this ) {
    EFI_INPUT_KEY key;
    EFI_STATUS status;

    status = gST->ConIn->ReadKeyStroke( gST->ConIn, &key );

    if ( status != EFI_NOT_READY ) {
        ASSERT_EFI_ERROR( status );

        if ( key.UnicodeChar == L' ' ) {
            //...
        }
        else {
            switch ( key.ScanCode ) {
                case SCAN_UP:
                    //...
                    break;
                case SCAN_DOWN:
                    //...
                    break;
                //...
            }
        }
    }
}
```

Na początku tworzymy zmienne do przechowywania statusu operacji i informacji o wciśniętym klawiszu (kod jest pisane zgodnie ze standardem pre-C99, więc deklaracje MUSZĄ być na początku - inaczej kod się nie skompiluje). Następnie próbujemy odczytać wciśnięty klawisz, jeżeli żaden klawisz nie został wciśnięty funkcja *ReadKeyStroke* zwraca wartość *EFI_NOT_READY*. W przypadku gdy udało się odczytać wciśnięty klawisz przechodzimy do jego obsługi:

1. Sprawdzamy, czy nie wystąpił błąd.
2. Sprawdzamy wartość unicode klawisza - w ten sposób przebiega obsługa klawiszy nie posiadających *scan code*.
3. Sprawdzamy *scan code* klawisza przy użyciu predefiniowanych makr.⁵

Ustawianie pozycji kursora i atrybutów tekstu.

```
gST->ConOut->SetCursorPosition( gST->ConOut, x, y );
gST->ConOut->SetAttribute( gST->ConOut, EFI_TEXT_ATTR( color, EFI_BLACK ) );
```

Działanie tych funkcji jest oczywiste, opis możliwych do ustawienia atrybutów można znaleźć w specyfikacji UEFI.⁶

UefiRuntimeServicesTableLib.h

Pobranie aktualnego czasu.

```
EFI_TIME time;
gRT->GetTime( &time, NULL );
```

Tą funkcję wykorzystaliśmy do inicjalizacji generatora liczb pseudolosowych.

⁵ Tamże, s. 438

⁶ Tamże, s. 464

Obsługa przerwania w EFI.

W EFI nie występują przerwy sprzętowe, jedyne co jest dostępne to zegar (timer). Zamiast przerwania można korzystać z techniki poolingu, wykorzystując zdarzenia timera (przykład tworzenia we wcześniejszej części opracowania).

Wewnętrznie timer działa w następujący sposób:⁷

1. Funkcja `CoreTimerTick()` znajdująca się w warstwie DXE jest wywoływana w określonych odstępach czasowych, które są definiowane przez *Timer Architectural Protocol*.
2. `CoreTimerTick()` sprawdza wewnętrzne kolejki i podejmuje decyzję czy należy wygenerować zdarzenie *timera*.

Priorytety⁸

Ze zdarzeniami timera nieodłącznie związane są priorytety definiujące, które zdarzenia mogą przerywać pracę innych - zadania posiadające wyższy priorytet mogą przerywać tym z niższym.

Wyróżniamy trzy poziomy:

- `TPL_APPLICATION` - najniższy
- `TPL_CALLBACK` - pośredni
- `TPL_NOTIFY` - najwyższy
- `TPL_HIGH_LEVEL` - wyższy od `TPL_NOTIFY`, ale dostępny tylko dla firmware'u

Opis przeznaczenia każdego z poziomów priorytetów jest zawarty w dokumentacji interfejsu UEFI⁹.

EFI bazuje na sygnałach, nie na wątkach.¹⁰

- Podczas występowania przerwy zegarowej, firmware generuje odpowiednie zdarzenie timera (timer event) jeżeli jego *trigger time* dobiegł końca (trigger time to wartość `TIMER_PERIOD`, którą podawaliśmy w wywołaniu funkcji `SetTimer`).
- Dzięki temu zdarzenia o wyższym priorytecie mogą zostać wygenerowane nawet jeżeli aktualnie wykonuje się funkcja powiązana z obsługą zdarzenia o priorytecie niższym (przykładem takiej funkcji jest nasz `timerCallback`).
- Jeżeli *trigger time* któregoś ze zdarzeń wyczerpał się, ale aktualnie wykonuje się funkcja powiązana z zadaniem o wyższym priorytecie to zdarzenie zostanie wygenerowane po zakończeniu jej obsługi.
- Kiedy wszystkie oczekujące zdarzenia zostaną obsłużone wykonanie jest kontynuowane na poziomie `TPL_APPLICATION`.

⁷ <https://www.mail-archive.com/edk2-devel@lists.sourceforge.net/msg02724.html> (dostęp: 30.01.2015)

⁸ 2_4_Errata_B.pdf, s. 114, dz. cyt.

⁹ Tamże, s. 115

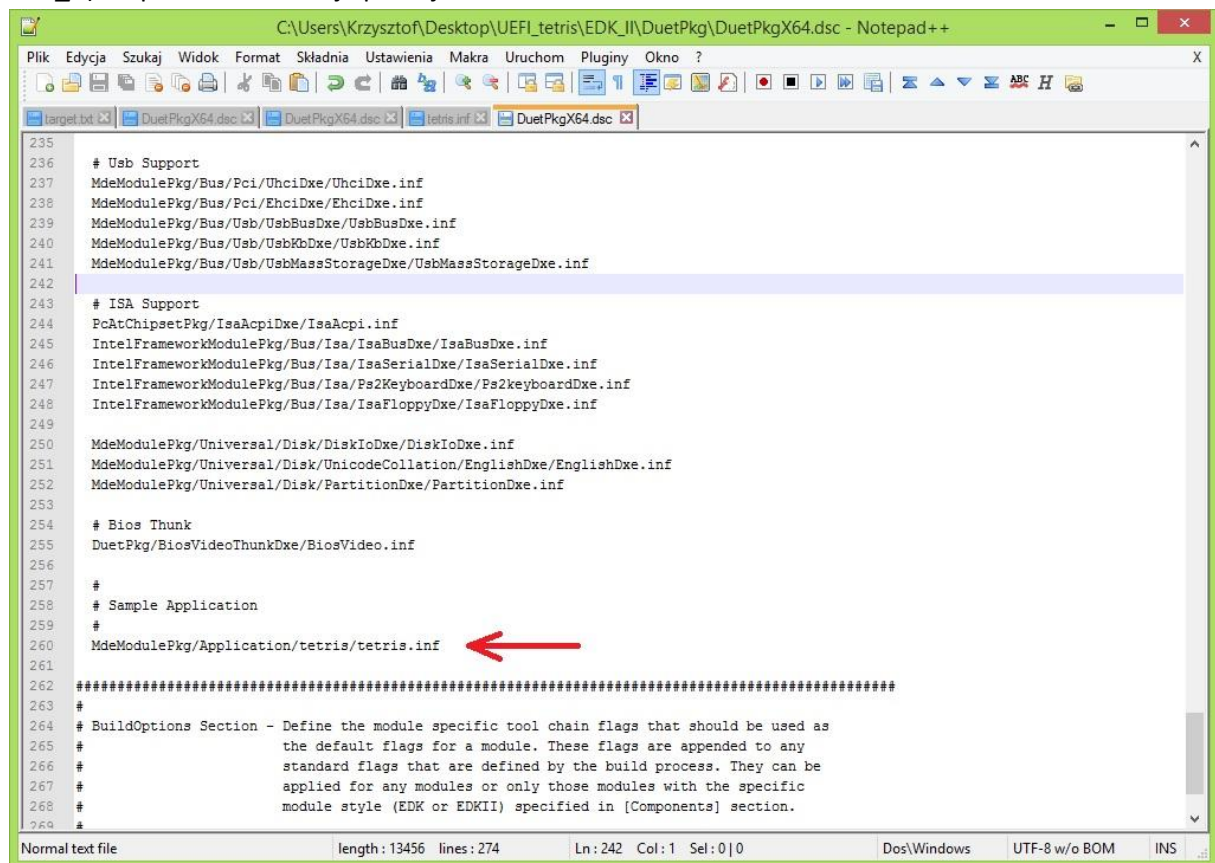
¹⁰ Tamże, s. 114

Jak zbudować wersję 64 bitową aplikacji.

W pierwszym sprawozdaniu opisany został proces konfiguracji środowiska i budowania aplikacji w trybie 32 bitowym, pozwalało to na uruchamianie jej w emulatorze Shella EFI. Poniżej przedstawiamy kroki, które były niezbędne do zbudowania programu w trybie 64 bitowym, tak aby można było go odpalić na fizycznym sprzęcie o architekturze 64 bitowej.

Aplikację w wersji x64 będziemy musieli zbudować w innym niż dotychczas module EDK. Do tej pory korzystaliśmy z MdeModulePkg, teraz wykorzystamy moduł DuetPkg. Nie oznacza to jednak, że przestaniemy korzystać z dołączanych wcześniej bibliotek z innych modułów (MdePkg, ShellPkg). Zbudowanie projektu przy użyciu innego modułu EDK nie będzie od nas wymagało kopiowania, ani przenoszenia folderu z źródłami aplikacji.

Po pierwsze, musimy zadeklarować naszą aplikację w odpowiednim pliku .dsc¹¹ modułu. W tym celu przechodzimy w naszym workspace'ie do folderu `.\EDK_II\DuetPkg` i edytujemy plik o nazwie `DuetPkgX64.dsc`. W sekcji `[Components]`, dodajemy wpis – ścieżkę względną (względem katalogu `EDK_II`) do pliku .inf¹² naszej aplikacji:



```
235
236 # Usb Support
237 MdeModulePkg/Bus/Pci/UhciDxe/UhciDxe.inf
238 MdeModulePkg/Bus/Pci/EhciDxe/EhciDxe.inf
239 MdeModulePkg/Bus/Usb/UsbBusDxe/UsbBusDxe.inf
240 MdeModulePkg/Bus/Usb/UsbKbdDxe/UsbKbdDxe.inf
241 MdeModulePkg/Bus/Usb/UsbMassStorageDxe/UsbMassStorageDxe.inf
242
243 # ISA Support
244 PcAtChipsetPkg/IsaAcpiDxe/IsaAcpi.inf
245 IntelFrameworkModulePkg/Bus/Isa/IsaBusDxe/IsaBusDxe.inf
246 IntelFrameworkModulePkg/Bus/Isa/IsaSerialDxe/IsaSerialDxe.inf
247 IntelFrameworkModulePkg/Bus/Isa/Ps2KeyboardDxe/Ps2KeyboardDxe.inf
248 IntelFrameworkModulePkg/Bus/Isa/IsaFloppyDxe/IsaFloppyDxe.inf
249
250 MdeModulePkg/Universal/Disk/DiskIoDxe/DiskIoDxe.inf
251 MdeModulePkg/Universal/Disk/UnicodeCollation/EnglishDxe/EnglishDxe.inf
252 MdeModulePkg/Universal/Disk/PartitionDxe/PartitionDxe.inf
253
254 # Bios Thunk
255 DuetPkg/BiosVideoThunkDxe/BiosVideo.inf
256
257 #
258 # Sample Application
259 #
260 MdeModulePkg/Application/tetris/tetris.inf
261
262 #####
263 #
264 # BuildOptions Section - Define the module specific tool chain flags that should be used as
265 # the default flags for a module. These flags are appended to any
266 # standard flags that are defined by the build process. They can be
267 # applied for any modules or only those modules with the specific
268 # module style (EDK or EDKII) specified in [Components] section.
269 #
```

¹¹ http://tianocore.sourceforge.net/wiki/Getting_Started_Writing_Simple_Application#6.29_Build_your_UEFI_Application (dostęp: 01.02.2015)

¹² http://tianocore.sourceforge.net/wiki/Getting_Started_Writing_MyHelloWorld.inf (dostęp: 01.02.2015)

Następnie przechodzimy do pliku konfiguracyjnego `target.txt`, znajdującego się w katalogu `.\EDK_II\Conf`. Zmieniamy w nim następujące parametry:

```
ACTIVE_PLATFORM = DuetPkg/DuetPkgX64.dsc
TARGET_ARCH = X64
```

Resztę pozostawiamy bez zmian –inne parametry zostały już wcześniej przez nas odpowiednio ustawione.

Na koniec musimy przystosować nasz projekt VS (możemy go oczywiście uprzednio skopiować i nadać mu inną, bardziej odpowiednią nazwę). Otwieramy plik `./NT32/NT32.sln` za pomocą Visual Studio 2013, z którego wcześniej już korzystaliśmy.

W panelu „*Solution Explorer*”, klikamy prawym przyciskiem myszy na nasz projekt (NT32) i z menu kontekstowego wybieramy opcję „*Properties*”. Ukáže nam się nowe okno. W drzewie konfiguracji, wybieramy gałąź „*Configuration Properties*”, następnie klikamy przycisk „*Configuration Manager...*”, w prawym górnym rogu okna. Chcemy w ten sposób utworzyć nową konfigurację dla platformy x64. Z rozwijanej listy „*Active Solution Platform*” wybieramy „*<New...>*”. Ukáže się nowe, małe okno, w którym wybieramy interesującą nas platformę (x64). W polu „*Copy settings from*” wybieramy dotychczas wykorzystywaną konfigurację. Klikamy „*OK*” oraz „*Close*”. Nowo utworzona konfiguracja (dla architektury x64), powinna automatycznie stać się aktywną.

Musimy jeszcze zmienić wykorzystywane (dołączane) biblioteki zewnętrzne (z platformy EDK II) na odpowiednie dla 64 bitów. W drzewie konfiguracji, w aktualnie otwartym oknie ustawień projektu „*NT32 Property Pages*”, przechodzimy do gałęzi „*Configuration Properties*” -> „*VC++ Directories*”. Odszukujemy i edytujemy wpis „*Include Directories*”. Zmieniamy jego wartość z:

```
$(SolutionDir)..\EDK_II\MdePkg\Include\Ia32;
$(SolutionDir)..\EDK_II\ShellPkg\Include;
$(SolutionDir)..\EDK_II\MdePkg\Include;$(IncludePath)
```

na:

```
$(SolutionDir)..\EDK_II\MdePkg\Include\X64;
$(SolutionDir)..\EDK_II\ShellPkg\Include;
$(SolutionDir)..\EDK_II\MdePkg\Include;$(IncludePath)
```

To wszystko. Możemy teraz zbudować projekt: *BUILD -> Build Solution (Ctrl + Shift + B)*.

Skompilowane programy modułu DuetPkg (w tym nasz *tetris.efi* w wersji 64 bitowej) znajdują się w odpowiednim podkatalogu katalogu `.\EDK_II\Build\DuetPkgX64`.