Kinetis Design Studio (2)

Nowe, bezpłatne środowisko programistyczne

Kontynuujemy prezentację nowego środowiska programistycznego firmy Freescale o nazwie Kinetis Design Studio, które docelowo ma zastąpić – jako narzędzie dla mikrokontrolerów Kinetis – cieszącego się dużą popularnością Code Warriora. W artykule przedstawiamy przykładową aplikację zrealizowaną pod jego „opieką”.

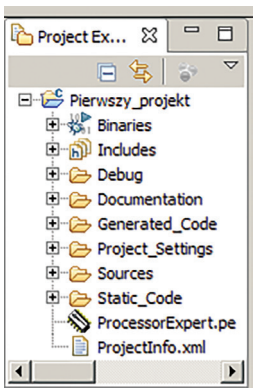
Po wygenerowaniu za pomocą Processor Experta interfejsu programistycznego do peryferiów mikrokontrolera praca z tym narzędziem jest zakończona. Można w tym momencie przejść do drugiego etapu tworzenia aplikacji, którym jest pisania kodu źródłowego aplikacji.

Pisanie kodu źródłowego aplikacji

Aby przejść do tworzenia kodu źródłowego, należy zmienić perspektywę programu Kinetis Design Studio z Hardware na C/C++. Tu programista posługuje się przede wszystkim dwoma narzędziami: menedżerem projektu oraz edytorem. Omówmy je po kolei.

Menedżer projektu znajduje się w oknie Project Explorer i jak sama nazwa wskazuje służy do zarządzania projektami programistycznymi oraz plikami tychże projektów. Narzędzie to pozawala między innymi na dodawanie plików do projektu, usuwanie plików z projektu i wybieranie, które z nich mają zostać otwarte do edycji. Widok okna CodeWrrior Projects z przykładowym projektem pokazano na **rysunku 10**.

Drzewo projektu składa się z sześciu folderów (katalogów) fizycznych (istniejących fizycznie na dysku twardym) i dwóch wirtualnych. Pierwszym folderem wirtualnym jest *Binaries*. Zawiera on plik wykonywalny (z rozszerzeniem *.elf*), który powstaje po skompilowaniu i linkowaniu projektu. Drugim folderem wirtualny jest *Includes*. Po jego otworzeniu programista uzyskuje dostęp do listy różnych plików nagłówkowych projektu. Foldery fizyczne to:



Rysunek 10. Okno zarządzania projektami i plikami projektów

- *Debug* – folder z różnymi plikami związanymi z procesem debugowania. Szczególnie ważny jest tu plik o nazwie makefile określający reguły kompilacji i linkowania programu.
- *Documentation* – folder z automatycznie tworzoną przez CodeWarrior dokumentacją projektu.
- *Generated_Code* – folder z zawierającymi kod źródłowy plikami wygenerowanymi przez *Processor Experta*.
- *Project_Settings* – folder z plikami konfiguracyjnymi projektu oraz plik startup.c zawierający kod uruchomieniowy dla mikrokontrolera.
- *Sources* – folder, który zawiera najważniejszy plik projektu – *main.c*.
- *Static_Code* – folder z plikami, w których zdefiniowane są nazwy rejestrów i bitów rejestrów mikrokontrolera, jak również nazwy własne używane w plikach generowanych przez *Processor Experta*.

Dodatkowo do projektu jest dołączony plik *ProcessorExpert.pe*. Jest to plik, który przechowuje listę wybranych za pomocą *Processor Experta* komponentów oprogramowania wraz z ich konfiguracją. Poprzez wywołanie tego pliku programista może w każdej chwili wrócić do wykonanej wcześniej konfiguracji interfejsu programistycznego peryferiów mikrokontrolera i ją edytować według potrzeb.

Warto w tym miejscu bliżej przyjrzeć się plikowi main.c, który powyżej określony został jako najważniejszy plik projektu. Jest tak, gdyż to w nim bezpośrednio, lub poprzez odwołania do innych plików są implementowane zadania przewidziane do realizacji przez mikrokontroler. Zawartość pliku *main.c* tuż po stworzeniu projektu, a więc jeszcze przed ingerencją weń programisty, można podzielić na dwie części. Pierwsza część to dyrektywy #include z odwołaniami do innych plików projektu:

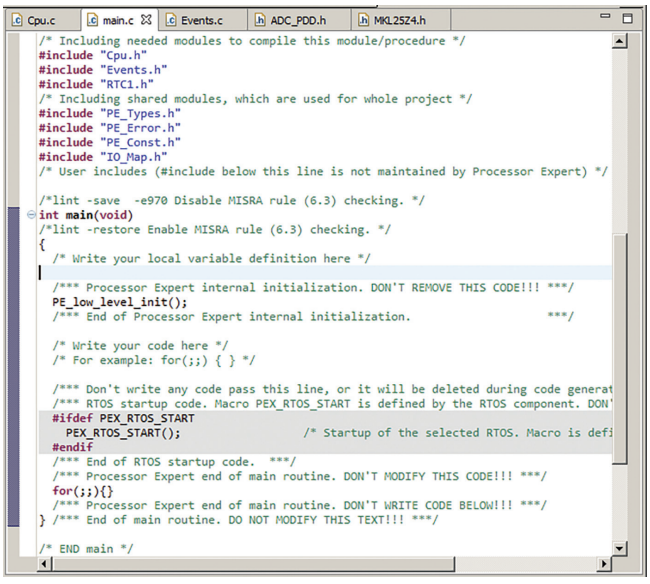
```
#include "Cpu.h"
#include "Events.h"
#include "RTC1.h"
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include „IO_Map.h"
```

Druga część to główna funkcja projektu – *main()*. W funkcji tej domyślnie znajdują się trzy bloki kodu. Pierwszy blok składa się z jednej linii kodu, którą jest wywołanie funkcji *PE_low_level_init*. W tej funkcji odbywa się inicjalizacja i konfiguracja peryferiów mikrokontrolera (zgodnie z wcześniejszymi ustawieniami w narzędziu Processor Expert). Drugi blok kodu odpowiada za zainicjowanie systemu operacyjnego, o ile programista zadeklarował wcześniej chęć korzystania z niego:

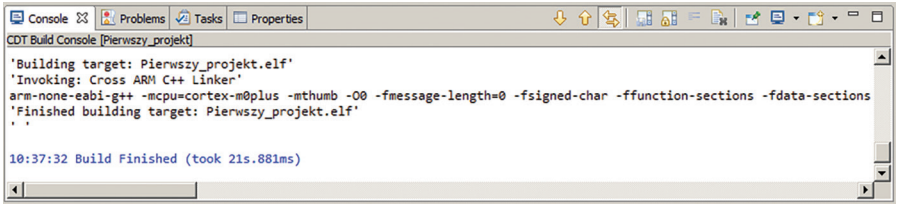
```
#ifdef PEX_RTOS_START
  PEX_RTOS_START();
#endif
```

Trzecim blokiem kodu jest pętla nieskończona typu *for*. Programista może w niej umieścić operacje, które mają być wykonywane cyklicznie przez mikrokontroler (*for(;;){}*).

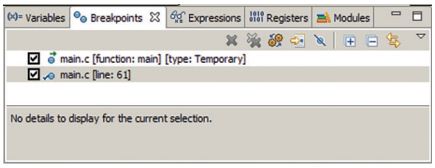
Na tym można zakończyć opis pierwszego z narzędzi perspektywy C/C++, a więc menedżera projektu. Drugim kluczowym narzędziem do tworzenia kodu źródłowego aplikacji jest edytor. Jest to narzędzie, które umożliwia edycję plików projektu programistycznego. Programista chcąc przejść do trybu edycji danego pliku wyszukuje go w oknie Project Explorer, a następnie dwukrotnie na niego klika. W efekcie plik zostanie otworzony do edycji w polu



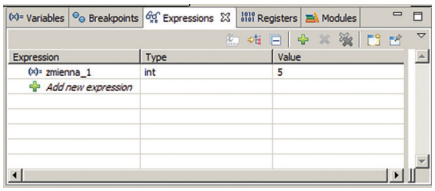
Rysunek 11. Widok edytora z otwartym plikiem main.c



Rysunek 12. Okno z logiem dotyczącym procesu kompilacji projektu programistycznego



Rysunek 13. Okno Breakpoints z listą pułapek ustawionych w kodzie źródłowym



Rysunek 14. Okno do podglądu wartości typów danych

edytora, które znajduje się w centralnej części okna Kinetis Design Studio. W edytorze może być otwarty więcej niż jeden plik. Każdy z otwartych plików ma przyporządkowaną zakładkę, dzięki czemu można się między nimi w wygodny sposób przełączać. Samo narzędzie edytora ma standardową funkcjonalność. Otwarte w nim pliki mają postać tekstu, na którym można wykonywać standardowe operacje np. dopisywania, usuwania, modyfikowania itp. Tekst w edytorze jest kodem źródłowym przeważnie napisanym w C. Programista używając elementów tego języka takich jak np. zmienne, tablice, struktury, wskaźniki, funkcje czy też instrukcje warunkowe może rozwijać kod źródłowy aplikacji. Edytor oferuje różne mechanizmy, które ułatwiają czytanie kodu. Są to między innymi: kolorowanie składni języka programowania i numerowanie linii kodu. Jako przykład **rysunek 11** pokazuje plik main.c otworzony w edytorze.

Gdy czynność tworzenia kodu źródłowego aplikacji jest zakończona, należy skompilować projekt (menu programu CodeWarrior: Project –> Build Project). Przebieg procesu kompilacji widoczny jest w oknie Console (**rysunek 12**).

Uruchamianie i debugowanie aplikacji

W pierwszym etapie tworzenia aplikacji wygenerowany został interfejs programistyczny do peryferiów mikrokontrolera. W ramach drugiego etapu programista napisał kod źródłowy aplikacji w oparciu o wygenerowany interfejs. W ostatnim, trzecim etapie poprawność działania aplikacji jest sprawdzana poprzez jej uruchomienie i debugowanie na platformie sprzętowej.

Zanim będzie można przystąpić do debugowania aplikacji, należy wybrać programator/debuger, który będzie używany. W tym celu należy z menu głównego *Kinetis Design Studio* wybrać *Run* → *Debug Configurations*. Zostanie otwarte okno o nazwie *Debug Configurations*. W jego lewej części jest dostępna lista, w której każdy z rekordów oznacza inną konfigurację debugowania. Wybrać należy *GDB PEMicro Interface Debugging*, gdyż taki właśnie programator/debuger udostępnia płytka FRDM-KL25Z. W wyniku wybrania konfiguracji debugowania stworzona zostanie nowa jej instancja o nazwie *<nazwa projektu> Debug*. Po prawej stronie okna *Debug Configurations* wyświetlone zostaną opcje konfiguracyjne, podzielone na kilka zakładek. W zakładce *Debugger* należy ustawić dwa parametry w związku z używaniem płytki FRDM-KL25Z. W polu Interface (interfejs debugera) wybrać należy *OpenSDA Embedded Debug* – USB Port, a w polu *Device Name* (symbol mikrokontrolera) zaznaczyć KL25Z128M4. Zaakceptowanie zmian potwierdzić należy wciśnięciem przycisku *Apply*.

Teraz już można przystąpić do debugowania aplikacji na płytce FRDM-KL25Z. W celu rozpoczęcia tego procesu z menu głównego programu Kinetis Design Studio należy wybrać opcję *Run* → *Debug*. W tym momencie środowisko programistyczne wymusi zapis pliku wykonywalnego aplikacji do pamięci mikrokontrolera. Następnie mikrokontroler zostanie zresetowany i przygotowany do rozpoczęcia wykonywania aplikacji. Jednocześnie w Kinetis Design Studio perspektywa C/C+ zostanie zmieniona na *Debug*. Dzięki temu programiście udostępnione zostaną narzędzia sterujące procesem debugowania i wizualizujące stan systemu na różnych etapach debugowania. Do sterowania debugowaniem przewidziany jest mechanizm pracy krokowej, pozwalający wykonywać kod etapami, a nawet linia po linii. Komendy do pracy krokowej dostępne są w podmenu *Run* menu głównego Kinetis Design Studio. Najważniejsze z nich to:

- *Resume* (wznowienie).
- *Suspend* (wstrzymanie).
- *Terminate* (zatrzymanie).
- *Step Into* (wejście do wewnątrz bloku kodu np. funkcji).
- *Step Over* (wyjście na zewnątrz aktualnego bloku kodu).
- *Run to Line* (wykonanie kodu do miejsca oznaczonego kursorem).

Komendy te wywoływać można nie tylko z poziomu menu Kinetis Design Studio. Są one również dostępne (częściowo) w formie ikon ulokowanych pod menu.

Uzupełnieniem dla pracy krokowej w sterowaniu procesu sterowania debugowaniem jest narzędzie do ustawiania tak zwanych pułapek (breakpointów). Pułapka to wybrana przez programistę linia kodu źródłowego, po wykonaniu której działanie aplikacji zostanie wstrzymane. Aby ustawić pułapkę należy w edytorze kliknąć na linię kodu, której wykonanie ma wyzwalać pułapkę, a następnie z menu Kinetis Design Studio wybrać *Run* → *Toggle Breakpoint*. Po ustawieniu pułapki obok wskazanej linii kodu pojawi się symbolizująca pułapkę niebieska kulka Pułapki można włączać i wyłączać bez konieczności trwałego ich usuwania. Zarządzanie pułapkami jest realizowane z poziomu okna *Breakpoints* (**rysunek 13**).

Ważnym narzędziem wizualizacji stanu systemu podczas debugowania jest wgląd do wartości typów danych aplikacji (np. zmiennych i struktur). Dostęp do tego narzędzia daje okno *Expressions* (**rysunek 14**), przy czym każdy typ danych, który programista chce obserwować, musi zostać dodany do tego okna ręcznie. Czynność tą wykonuje się poprzez zaznaczenie w edytorze wybranego typu danych, wywołanie prawym przyciskiem myszy menu kontekstowego oraz wybranie w wyświetlonej listy opcji *Add Watch Expression…*

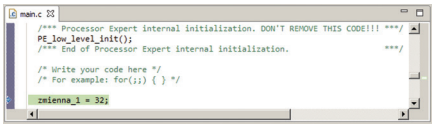
Ostatnim z głównych narzędzi debugowania jest edytor. Poprzez kolorowanie na zielono linii kodu i zaznaczanie jej dodatkowo strzałką edytor informuje programistę na jakim etapie wykonywania aplikacji jest w danej chwili debuger (**rysunek 15**).

Teraz zajmiemy się przygotowaniem przykładowej aplikacji. Wykorzystamy ją przede wszystkim do zademonstrowania sposobu, w jaki używać w praktyce narzędzi Kinetis Design Studio, dlatego cechuje ją wszystkim prostota. Aby połączyć charakter edukacyjny i funkcjonalny, zdecydowano się na aplikację demonstracyjną będącą prostym systemem zabezpieczenia budynku. Jako detektor wykorzystano czujnik kontaktronowy. Jako element sygnalizacyjny użyto generatora piezoelektrycznego (popularnie zwanego buzzerem). Do sterowania tymi komponentami wykorzystano mikrokontroler z płytki FRD-KL25Z. Do obsługi czujnika kontaktronowego i buzzera wystarczą porty wejścia/wyjścia mikrokontrolera.

Kontaktron jest czujnikiem pola magnetycznego. Gdy na kontaktron nie działa zewnętrzne pole magnetyczne, styki czujnika pozostają w stanie rozwarcia. Pod wpływem odpowiednio ukierunkowanego zewnętrznego pola magnetycznego w stykach kontaktronu indukuje się własne pole magnetyczne. W warunkach tych styki zaczynają się przyciągać i w efekcie zwierają się. Stan ten jest nietrwały (gdy pole przestanie oddziaływać na kontaktron, jego styki rozewrą się) i nie jest jednorazowy (można go wielokrotnie powtarzać). Z uwagi na tylko dwa stany styków (zwarte i rozwarte) kontaktron to typ czujnika, który określa się mianem detektora.

Kontaktron wraz z magnesem będącym źródłem pola magnetycznego tworzy czujnik kontaktronowy. Czujnik taki stosowany jest powszechnie do wykrywania zmiany położenia obiektu. Kontaktron jest montowany w punkcie stacjonarnym, umożliwiającym wykrycie zmiany położenia obiektu. Z kolei na zmieniającym swoje położenie obiekcie mocowany jest magnes trwały. Na to czy styki kontaktronu są zwarte czy rozwarte ma wpływ położenie magnesu względem kontaktronu oraz odległość magnesu od kontaktronu.

Czujniki kontaktronowe są często spotykanym elementem systemów zabezpieczenia budynku.



Rysunek 15. Okno edytora w trybie debugowania aplikacji

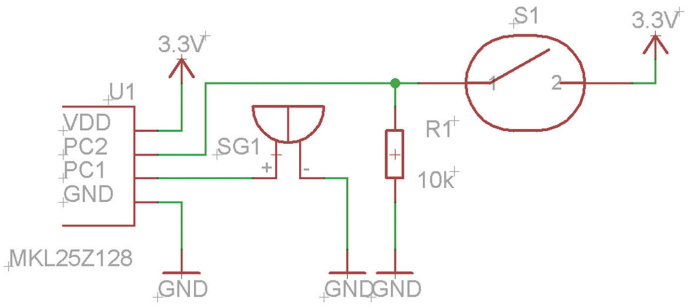
W takich zastosowaniach kontaktrony montowane są na framugach okien i drzwi, natomiast magnesy przytwierdzane są do okien i drzwi. Kontaktrony i magnesy mocowane są obok siebie, dzięki czemu gdy okna i drzwi są zamknięte, magnesy oddziałując na kontaktrony sprawiają, że styki kontaktronów są zwarte. Natomiast, gdy następuje otwarcie drzwi lub okien, odległość między magnesami i kontaktronami zwiększa się i tym samym styki kontaktronów zostają rozwarte. Fakt rozwarcia styków kontaktronów jest wykrywany przez system, który reaguje w odpowiedni sposób np. włączając alarm.

Zaprezentowana przykładowa aplikacja to właśnie prosty system zabezpieczenia budynku. Do jego budowy oprócz płytki FRDM-KL25Z wykorzystano jeden czujnik kontaktronowy służący do wykrywania otwarcia drzwi lub okna oraz jeden generator piezoelektryczny do dźwiękowej sygnalizacji stanu otwarcia wymiennych obiektów.

Schemat elektryczny systemu przedstawiono na **rysunku 16**. Wyprowadzenie buzzera oznaczone znakiem „–” jest połączone z masą. Wyprowadzenie „+” tego komponentu jest połączone z GPIO mikrokontrolera o nazwie PC1. W takiej konfiguracji, aby buzzer generował sygnał dźwiękowy, należy ustawić poziom logiczny 1 na GPIO PC1. Jedno z wyprowadzeń kontaktronu jest połączone z napięciem zasilania o potencjale 3,3 V. Drugie wyprowadzenie kontaktronu jest połączone z GPIO mikrokontrolera o nazwie PC2 oraz do masy przez rezystor R1. Gdy styki kontaktronu są rozwarte, na wyprowadzeniu PC2 wartość napięcia wynosi 0 V. Gdy styki kontaktronu zostaną zwarte, wartość napięcia na wyprowadzeniu PC2 wzrasta do 3,3 V. Napięcie 0 i 3,3 V to dla mikrokontrolera odpowiednio wartość logiczna 0 i 1, zatem stan styków kontaktronu można odczytywać bezpośrednio za pomocą portów wejścia wyjścia mikrokontrolera.

Aplikacja działa zgodnie ze schematem blokowym zaprezentowanym na **rysunku 17**. Mikrokontroler w momencie rozpoczęcia pracy wykonuje inicjalizację i konfigurację GPIO PC1 i PC2. PC1 zostaje skonfigurowane do pracy w trybie wyjściowym, natomiast PC2 zostaje skonfigurowane do pracy w trybie wejściowym. Następnie mikrokontroler przechodzi do wykonywania nieskończonej pętli. W pętli tej cyklicznie sprawdzany jest stan logiczny na wyprowadzeniu PC2. Jeśli odczytana zostanie wartość logiczna 0 (styki kontaktronu są w tym przypadku rozwarte) następuje ustawienie stanu logicznego 1 na wyprowadzeniu PC1 (buzzer zostaje włączony). W przeciwnym wypadku, gdy wartość logiczna na wyprowadzeniu PC2 jest równa 1, mikrokontroler zeruje wyprowadzenie PC1 wyłączając buzzer.

Realizację aplikacji należy rozpocząć od stworzenia projektu programistycznego dla mikrokontrolera z płytki FRD-KL25Z. Gdy

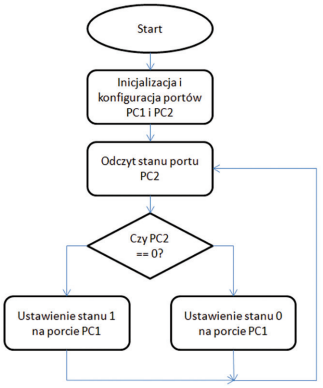


Rysunek 16. Fragment schematu elektrycznego przykładowego systemu wykorzystującego czujnik kontaktronowy (zrzut ekranu z programu Eagle)

projekt programistyczny jest już stworzony, konieczne jest wygenerowanie interfejsu programistycznego dla peryferiów mikrokontrolera używanych przez aplikację. Zarówno czynność tworzenia projektu jak i zasady generowania interfejsu programistycznego zostały przedstawione szczegółowo w poprzednim artykule, zatem mijałoby się z sensem zamieszczanie w tym miejscu ponownie tego opisu. Poniższy opis dotyczyć będzie zatem tylko konkretnej aplikacji: komponentów oprogramowania użytych do sterowania peryferiami, funkcji udostępnianych przez te komponenty i kodu źródłowego samej aplikacji.

Do sterowania GPIO użyto dwóch komponentów oprogramowania BitIO_LDD. Pierwszy z nich, nazwany Sensor1, odpowiada za wyprowadzenia PC2. Drugi komponent, o nazwie Buzzer, przyporządkowany jest do wyprowadzenia PC1. Konfigurację obu komponentów przedstawiono na **rysunku 18**.

Odczytywanie stanu wyprowadzenia PC2 zrealizowano za pomocą zaimplementowanej w komponencie BitIO_LDD funkcji o nazwie



Rysunek 17. Schemat blokowy algorytmu działania aplikacji

GetVal. Odczytana wartość zapisywana jest w zmiennej odczyt_czujnika. Wystawianie wartości logicznej na wyprowadzeniu PC1 zrealizowano za pomocą również należących do komponentu BitIO_LDD funkcji o nazwie SetVal i ClrVal. Kompletny listing z kodem źródłowym aplikacji przedstawiono na **listingu 1**.

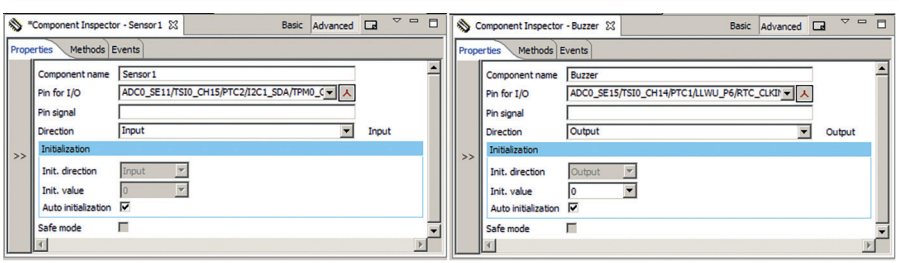
Szymon Panecki, EP

Listing 1. Kod źródłowy aplikacji wykorzystującej kontaktron i buzzer

```
#include „Cpu.h”
#include „Events.h”
#include „Buzzer.h”
#include „Sensor1.h”
#include „PE_Types.h”
#include „PE_Error.h”
#include „PE_Const.h”
#include „IO_Map.h”

unsigned char odczyt_czujnika = 0;

int main(void)
{
    PE_low_level_init();
    #ifdef PEX_RTOS_START
    PEX_RTOS_START();
#endif
    for(;;)
    {
        odczyt_czujnika = Sensor1_GetVal(NULL);
        if (odczyt_czujnika == 0) Buzzer_SetVal(NULL);
        else if (odczyt_czujnika == 1) Buzzer_ClrVal(NULL);
    }
}
```



Rysunek 18. Okna konfiguracyjne wykorzystanych w aplikacji komponentów oprogramowania typu BitIO_LDD