Envfs czyli zmienne środowiskowe w systemie ISIX-RTOS

Pisząc oprogramowanie dla mikrokontrolerów często spotykamy się z problemem zapisu danych konfiguracyjnych, aby je zachować, gdy urządzenie jest wyłączone. Z uwagi na to, iż zapis danych konfiguracyjnych jest jedną z najczęściej wykorzystywanych funkcjonalności aplikacji system ISIXRTOS został wyposażony w API służące do obsługi danych konfiguracyjnych w sposób niezależny od sprzętu.

Najczęściej spotykanym rozwiązaniem jest zastosowanie wewnętrznej lub zewnętrznej pamięci EEPROM oraz zapisywanie poszczególnych nastaw (zmiennych) bezpośrednio pod adresami fizycznymi w pamięci nieulotnej. Jedyną zaletą takiego rozwiązania jest prostota, natomiast podejście to ma szereg wad, do których możemy zaliczyć: zależność kodu aplikacji od konkretnego typu pamięci oraz jej architektury, brak możliwości dynamicznej zmiany rozmiaru danego rekordu czy brak możliwości usunięcia rekordu. Również istotną wadą jest nierównomierne obciążenie pamięci cyklami zapisu, gdzie wykorzystywany jest tylko początkowy obszar, prowadząc do szybszego uszkodzenia pamięci nieulotnej, która z reguły ma ograniczoną ilość cykli zapisu.

Z uwagi na to, iż zapis danych konfiguracyjnych jest jedną z najczęściej wykorzystywanych funkcjonalności aplikacji system ISIXRTOS został wyposażony w API służące do obsługi danych konfiguracyjnych w sposób niezależny od sprzętu. Interfejs ten stanowi część biblioteki *libfoundation*. Biblioteka również może być wykorzystana we własnych projektach pozbawionych systemu operacyjnego, ponieważ jest niezależna od jądra systemu. Architektura tego rozwiązania oparta jest o podział pamięci na równomiernej wielkości jednostki alokacji, i charakteryzuje się następującymi parametrami:

- API niezależne od architektury pamięci przechowującej dane.
- Wsparcie dla pamięci o zapisie swobodnym np. EEPROM oraz wsparcie dla pamięci zorganizowanych w postaci stron np. Flash.
- Równomierne wykorzystanie wszystkich komórek pamięci tzw. *Wear Leveling*.
- Identyfikacja zasobu/danych po kluczu (int).
- Możliwość dodawania oraz usuwania dowolnych zmiennych.
- Możliwość zapisu danych o dynamicznym rozmiarze.
- Zabezpieczenie integralności poszczególnych zmiennych za pomocą algorytmu CRC16.

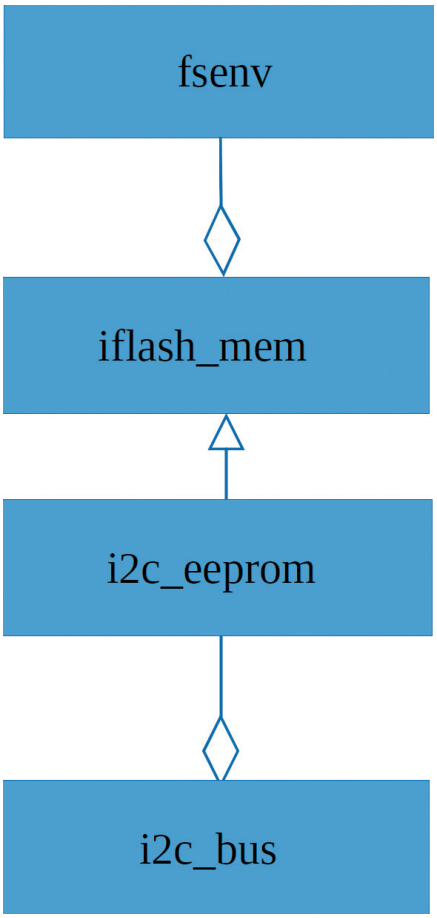
Architektura biblioteki

Biblioteka obsługująca zapis danych konfiguracyjnych zrealizowana została w sposób warstwowy, aby uniezależnić sposób organizacji przechowywanych od danych fizycznego układu pamięci służącego do ich przechowywania (**rysunek 1**).

Została ona napisana w języku C++ w dialekcie ISO/IEC 14882:2011 i z wykorzystaniem programowania obiektowego. Klasa *fsenv*, implementuje niezależny od sprzętu sposób organizacji zapisu danych jednocześnie stanowiąc interfejs API użytkownika. Konstruktor klasy przyjmuje referencję do abstrakcyjnej klasy bazowej *iflash_mem*, która stanowi interfejs reprezentujący pamięć masową. Jako drugi argument przyjmuje liczbę stron pamięci które będą zarezerwowane do przechowywania zmiennych nieulotnych:

```
fs_env( iflash_mem& flash_mem ,
unsigned n_pg = 0 )
```

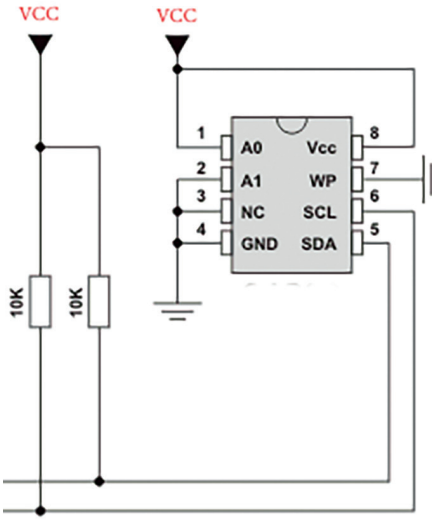
Dodatkowy argument bywa przydatny, gdy do przechowywania danych chcemy wykorzystać kilka stron pamięci zamiast całego dostępnego obszaru np. w celu wykorzystania kilku stron wolnej pamięci Flash mikrokontrolera. Parametr ten ma domyślnie zdefiniowaną wartość równą 0, co oznacza wykorzystanie całego dostępnego obszaru. Aktualnie w systemie ISIXRTOS dostępne są dwie implementację interfejsu *i2c_flash_mem* stanowiące część odrębnej biblioteki *libisixdrvstm32*. Klasa *i2c_eeprom* udostępnia sterownik dla pamięci EEPROM, natomiast klasa *stm32fmc* udostępnia sterownik wykorzystujący wolne strony pamięci FLASH mikrokontrolera rodziny STM32. Podczas tworzenia obiektu klasy *fsenv* możemy przekazać referencję do obiektu klasy *i2c_eeprom*, co będzie owocowało przechowywaniem danych konfiguracyjnych w zewnętrznej pamięci EEPROM lub referencję do obiektu klasy *stm32fmc*, w wyniku czego dane konfiguracyjne zostaną zapisane w wewnętrznej pamięci Flash mikrokontrolera. Liczba stron zarezerwowanych do tego celu możemy określić za pomocą parametru *n_pg*. Dzięki klasie *fsenv* potrafiącej obsługiwać pamięci o zapisie swobodnym jak i stronicowanej, użytkownik końcowy jest całkowicie uniezależniony od jej fizycznej organizacji i może skupić się jedynie na zapisie lub odczycie danych. Dopisując kolejne klasy implementujące interfejs *iflash_mem* możemy utworzyć dodatkowe sterowniki obsługujące inne rodzaje pamięci.



Rysunek 1. Warstwowa konstrukcja biblioteki

Przedstawiony wcześniej rysunek przedstawia hierarchię klas przy korzystaniu ze sterownika klasy *i2c_eeprom* obsługującej pamięci EEPROM typu I²C. Przy tworzeniu obiektu tego typu należy przekazać referencję do obiektu *i2c_bus* reprezentującego szynę I²C, do której jest dołączona pamięć oraz określić adres pamięci i jej rodzaj.

Klasa *fsenv* udostępnia prosty interfejs służący do przechowywania danych. Metody publiczne wzorowane są na linuksowych



Rysunek 2. Sposób dołączenia pamięci zewnętrznej 24C16 do zestawu *STM32Butterfly*

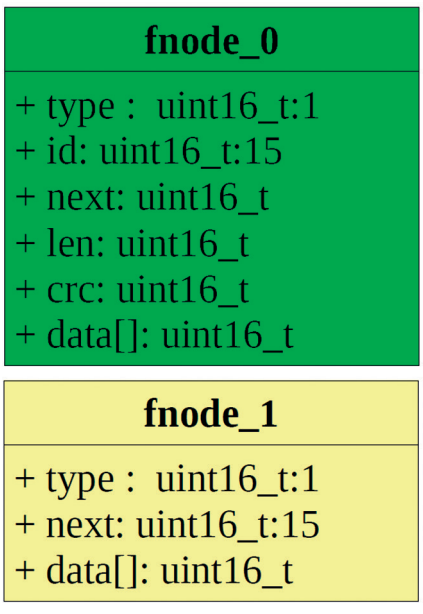
funkcjach z biblioteki *libc*, które w oryginalnej implementacji służą do zapisywania zmiennych środowiskowych. Różnią się jedynie rodzajem klucza, gdzie w przypadku funkcji linuksowych, identyfikatorem jest wartość tekstowa reprezentująca nazwę zmiennej, natomiast w przypadku klasy *fsenv* identyfikator stanowi wartość liczbowa.

- `int set( unsigned env_id, const void* buf, size_t buf_len );` Metoda *set* klasy *fsenv* służy do zapisania danych w pamięci nieulotnej. Parametr *env_id* stanowi identyfikator / klucz zapisywanej wartości. Parametr *buf* stanowi wskaźnik na dane do zapisania natomiast parametr *len* określa ich wielkość. Funkcja zwraca wartość *err_success(0)* w przypadku powodzenia, natomiast w przypadku wystąpienia problemów zwracany odpowiedni kod błędu z warstwy *fsenv* lub niższej.
- `int get( unsigned env_id, void* buf, size_t buf_len );` Metoda *get* jest analogiczną metodą służącą do odczytania danych z pamięci nieulotnej. Parametr *env_id* stanowi identyfikator / klucz odczytywanej danej. Parametr *buf* określa wskaźnik do bufora, gdzie dane będą przekopiowane natomiast parametr *buf_len* określa rozmiar bufora. Jeśli jego rozmiar będzie mniejszy niż ilość rzeczywistych danych przechowywanych w pamięci nieulotnej pod tym identyfikatorem, funkcja skopiuje tylko część danych o wielkości przekazanej przez zmienną *buf_len*. Funkcja zwraca wartość *err_success(0)* w przypadku powodzenia, natomiast w przypadku wystąpienia problemów zwracany odpowiedni kod błędu z warstwy *fsenv* lub niższej.
- `int unset( unsigned env_id );` Metoda *unset* klasy *fsenv* służy do usunięcia danych, z pamięci nieulotnej, których identyfikator / klucz przekazano jako parametr *env_id*. Funkcja zwraca wartość *err_success(0)* w przypadku powodzenia, natomiast w przypadku wystąpienia problemów zwracany odpowiedni kod błędu z warstwy *fsenv* lub niższej.

Przykład praktyczny

Aplikacja prezentująca użycie wspomnianej biblioteki dołączona jest do przykładów dla systemu ISIX i znajduje się w katalogu *advanced/envfs*. Kompilacja przykładu odbywa się za pomocą polecenia make. Aplikacja została przygotowana na platformę *STM32Butterfly*, która pozbawiona jest zintegrowanej pamięci EEPROM, zatem aby przetestować działanie należy do złącza I²C (Con7) dołączyć zewnętrzną pamięć EEPROM 24C16 (**rysunek 2**).

Aby zaobserwować efekt działania przykładu linie PD5 i PD6 należy dołączyć do RX i TX standardowego konwertera poziomów logicznych z układem MAX232. Należy również dołączyć zasilanie zestawu oraz programator JTAG, którym zaprogramujemy mikrokontroler. Skompilowany przykład wraz z kodami źródłowymi jest dostępny pod adresem http://goo.gl/h1q2R5. Aby zaprogramować zestaw do złącza programującego należy dołączyć dowolny JTAG zgodny z OCDLINK, a następnie wydać polecenie *make program*. Po zaprogramowaniu mikrokontrolera należy uruchomić program terminalowy skonfigurowany według następujących parametrów transmisji: *115200,n,8,1*. Po zaprogramowaniu na ekranie powinna pojawić się informacja o zapisaniu przykładowej wartości typu *int* w pamięci nieulotnej, a po 5 sekundach powinna pojawić się informacja o takiej samej wartości która została zapisana wcześniej.



Rysunek 3. Jednostka alokacji może przechowywać jeden z dwóch rodzajów danych, nagłówek oznaczony *fnode_0* oraz dane wraz ze wskaźnikiem następnej jednostki alokacji oznaczonym *fnode_1*

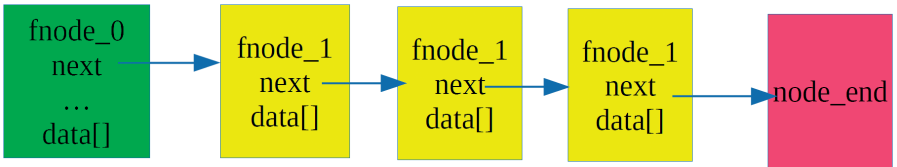
Ponieważ dostęp do danych konfiguracyjnych konieczny jest w wielu miejscach aplikacji, a ponadto w większości przypadków mamy tylko jedno miejsce gdzie przechowujemy dane nieulotne najwygodniejszym sposobem użytkowania powyższej klasy będzie stworzenie odpowiedniego pliku *cpp* z modułem, w którym globalnie utworzymy statyczny obiekt klasy *fsenv* oraz udostępnimy w oddzielnej przestrzeni nazw zestaw funkcji *setenv/getenv/unsetenv* o takich samych deklaracjach argumentów, jak metody klasy *fsenv*. W powyższym przykładzie utworzono pliki *app_env.cpp* oraz *app_env.hpp*, które udostępniają globalny interfejs do obsługi zmiennych konfiguracyjnych. W stosunku do metod publicznych dodano dodatkowe dwie funkcje korzystające z mechanizmu wzorców *setenv* oraz *getenv*:

```
template <typename T> int setenv(
unsigned env_id, const T& value
) {
        return setenv( env_id,
&value, sizeof( T ) );

template <typename T> int getenv(
unsigned env_id, T& value ) {
        return getenv( env_id,
&value, sizeof( T ) );
}
```

o takiej samej nazwie, jak pierwotne funkcje *setenv* oraz *getenv*, ale przyjmującej tylko dwa parametry. Dzięki takiemu rozwiązaniu, jeśli chcemy zapisać do pamięci nieulotnej wartość np. typu *int* wystarczy, że podamy jedynie klucz / identyfikator zmiennej oraz referencję do niej, a kompilator odpowiednio rozwinie wzorzec. Ponieważ magistrala I²C może być również wykorzystana do podłączenia innych układów, obiekt magistrali i2c nie jest tworzony jako część modułu *app_env*, tylko jest przekazywany przez referencję jako argument funkcji *init()*, która powinna być wywołana przed rozpoczęciem korzystania interfejsu *app_env*. Implementacja wspomnianych wcześniej funkcji jest bardzo prosta. Na początku modułu w anonimowej przestrzeni nazw zdefiniowano inteligentne wskaźniki typu *std::unique_ptr* do obiektu *fs_env* oraz *i2c_eeprom*

```
 namespace {
    static constexpr auto I2CA_
EEPROM = 0xA0;
    std::unique_ptr<fnd::i2c_
eeprom> m_eeprom;
    std::unique_
ptr<fnd::filesystem::fs_env> m_
fsenv;
    isix::semaphore m_lock {1,
1};
}
```



Rysunek 4. Przykład przedstawiający przydział jednostek alokacji dla zmiennej wymagającej kilku jednostek

Tworzenie obiektów klas oraz inicjalizacja wskaźników realizowana jest przez wspomnianą wcześniej funkcję init, której definicja wygląda następująco:

```
void initenv( fnd::bus::ibus& bus
)
{
    if( !m_eeprom ) {
        m_eeprom.reset(new
fnd::i2c_eeprom(bus, I2CA_EEPROM,
fnd::i2c_eeprom::type::m24c16));
        m_fsenv.reset( new
fnd::filesystem::fs_env( *m_eeprom
) );
    }
}
```

Na początku sprawdzamy czy moduł nie był wcześniej zainicjalizowany, jeśli nie wówczas tworzymy obiekt klasy *i2c_eeprom* oraz przypisujemy adres tego obiektu wskaźnikowi *m_eeprom*. Utworzenie tego obiektu wymaga podania referencji do magistrali, określenie typu pamięci i2c oraz podania jej fizycznego adresu na magistrali. Gdy już mamy gotowy obiekt klasy *i2c_eeprom* możemy przystąpić do utworzenia właściwego obiektu *fsenv* przekazując mu referencję do utworzonego wcześniej obiektu odpowiedzialnego za fizyczną obsługę pamięci. Po wykonaniu powyższych czynności inicjalizujących możemy dowolnie używać funkcji *setenv*/*getenv*/*unsetenv*. Przykład implementacji funkcji setenv modułu przedstawiono poniżej:

```
int setenv( unsigned env_id,
const void* buf, size_t buf_len )
{
    isix::sem_lock _lck( m_lock
);
    if( m_fsenv ) {
        return m_fsenv->set( env_
id, buf, buf_len );
    } else {
        return env::err_not_init;
    }
}
```

Funkcja ta sprowadza się do wywołania odpowiednich metod publicznych obiektu *m_fsenv*. Jedynym dodatkiem jest sprawdzenie czy obiekt *m_fsenv* istnieje oraz zabezpieczenie w postaci semafora, gdyby funkcje miały być wywoływane z wielu wątków. Realizacja współbieżności mogła być zrealizowana bezpośrednio w implementacji klasy *fsenv*, jednak z uwagi na chęć maksymalnego uniezależnienia implementacji klasy od systemu została ona wydzielona, aby można ją było wykorzystać samodzielnie bez ISIX-a. Mając gotowy zestaw funkcji umożliwiających zapis danych konfiguracyjnych możemy teraz przystąpić do ich wykorzystania w aplikacji, co zostało zaprezentowane w klasie *env_tester* znajdującej się w pliku *envfsmain.cpp*. Najistotniejszy fragment implementacji przedstawiono poniżej:

```
protected:
    virtual void main() {
        initenv( m_i2c );
        //! Set the env
        static constexpr int val
= 0x12345678;
        auto err = setenv( envid_
test, val );
        dbprintf(„Setenv code
%i”, err );
        //! Read the env
        isix::isix_wait_ms( 2000
);
        int val2 {};
        err = getenv( envid_test,
val2 );
        dbprintf(„Getenv code
%i value %08x”, err, val2 );
    }
private:
    stm32::drv::i2c_bus m_i2c {
stm32::drv::i2c_bus::busid::i2c1
, 400000 };
```

Klasa *env_tester* zawiera obiekt klasy *i2c_bus*, który jest tworzony przez listę inicjalizującą przyjmującą identyfikator szyny oraz prędkość transmisji. Referencja do tego obiektu jest następnie przekazywana do opisanej poprzednio funkcji *initenv*. Po zainicjalizowaniu biblioteki przystępujemy do zapisu przykładowej wartości z wykorzystaniem funkcji *setenv* w pamięci nieulotnej pod identyfikatorem o wartości *envid_test*. Następnie po odczekaniu 2 sekund wartość zmiennej reprezentowanej przez identyfikator *envid_test* jest ponownie odczytywana za pomocą funkcji *getenv* i wyświetlana na konsoli szeregowej.

Implementacji klasy fsenv (dla dociekliwych)

Działanie klasy *fsenv* oparte jest o ideę listy jednokierunkowej, oraz mechanizm podziału dostępnej pamięci na stałe jednostki alokacji. Rozmiar jednostki dobierany jest dynamicznie w zależności od wielkości pamięci. Jednostka alokacji może przechowywać jeden z dwóch rodzajów danych, nagłówek oznaczony *fnode_0* oraz dane wraz ze wskaźnikiem następnej jednostki alokacji oznaczonym *fnode_1* (**rysunek 3**).

Każda nowa zmienna rozpoczyna się od jednostki alokacji typu *fnode_0*, której znaczenie pól jest następujące:

type – identyfikator jednostki alokacji, gdzie wartość 0 oznacza typ **fnode_0,** a wartość 1 typ **fnode_1**,

id – identyfikator / klucz zmiennej,

next – adres następnej jednostki alokacji, jeśli zmienna nie mieści w pojedynczej jednostce,

len – rozmiar danych przechowywanych,

crc – suma CRC16 weryfikująca integralność danych,

data[] – dane do przechowania.

Jeśli dane która powinny zostać zapisana pod danym identyfikatorem zmieszczą się w polu *data[]* w nagłówku typu *fnode_0*, wówczas kolejna jednostka typu *fnode_1* nie będzie występować, a pole *next* przyjmie wartość *node_end*. Jeśli w polu *data[]* nagłówka nie uda się zmieścić wszystkich danych, zmienna *next* będzie zawierała adres następnej jednostki alokacji typu *fnode_1*. Jednostka alokacji typu *fnode_1* zawiera jedynie identyfikator pola *type = 1* oraz adres następnej jednostki alokacji, po czym następują właściwe dane do przechowania. Jeśli pole danych nie zmieści się w kolejnej jednostce alokacji typu *fnode_1*, wówczas pole *next* będzie zawierało adres kolejnej jednostki alokacji lub wartość *node_end*, jeśli jest to ostatnia jednostka alokacji. Na **rysunku 4** przedstawiono przykład przedstawiający przydział jednostek alokacji dla dłuższej zmiennej, która wymaga kilku jednostek.

Dodatkowego komentarza wymaga proces kasowania danych realizowany za pomocą metody *unset()*, Działanie tej metody jest zależne od typu pamięci, z którą klasa *fsenv* współpracuje. W przypadku pamięci o zapisie swobodnym np. EEPROM kasowanie zmiennej polega na przejściu przez wszystkie wykorzystywane jednostki alokacji i zapisaniu do ich pól *next* wartości *node_dirty = 0*. Tak oznaczona zmienna może być ponownie wykorzystywana (podobnie jak jednostka oznaczona jako *node_empty = 0xffff*) podczas wywołania metody *set()*. Zupełnie inaczej wygląda sytuacja w przypadku pamięci z podziałem na strony, gdzie ponowny zapis danych do danej komórki wymaga usunięcia zawartości całej strony pamięci. W tym przypadku działanie algorytmu jest odmienne i metoda *set()* alokując nowe jednostki wykorzystuje tylko zmienne typu *node_unused = 0xffff*, natomiast metoda *unset()* wpisuje w pole *next* wartości *node_dirty = 0*. Jeśli na danej stronie pamięci nie uda się znaleźć wolnej jednostki alokacji uruchamiany jest proces „odśmiecania” strony polegający na przeniesieniu używanych danych na nieużywaną stronę, a następnie skasowaniu nieużywanego bloku.

Lucjan Bryndza SQ5FGB, EP