Układ MPU w mikrokontrolerach STM32

Większość rozbudowanych mikrokontrolerów z rdzeniem Cortex-M3/4 (w tym z rodziny STM32) ma zintegrowany układ MPU (Memory Protection Unit) umożliwiający ograniczenie dostępu do wybranych regionów pamięci przez nieuprzywilejowany kod. W przeciwieństwie do układu MMU (Memory Management Unit) występującego w procesorach aplikacyjnych, MPU nie dostarcza mechanizmu pamięci wirtualnej, a jedynie zapewnia podstawową ochronę wybranych regionów pamięci fizycznej.

Jednostkę MPU można użyć do poprawienia niezawodności systemów wbudowanych. Można to osiągnąć poprzez:

- Rozdzielenie kodu i danych systemu operacyjnego od aplikacji użytkownika.
- Ochronę danych pomiędzy aplikacjami.
- Zabezpieczenie wybranych fragmentów kodu przed zapisem.
- Wykrywanie nieprawidłowości w działaniu programu np. przepełnienie stosu.

Najprostszym sposobem wykorzystania MPU jest wykrywanie dostępu do pamięci za pomocą pustego wskaźnika (*NULL*), lub wykrywanie przepełnienia stosu. Nie wymaga to dużych zmian w systemie oprócz włączenia jednostki ochrony pamięci. Separacja aplikacji użytkownika od systemu jest bardziej skomplikowana i wymaga większych nakładów pracy, oraz uwzględnienia w systemie obecności MPU. Na przykład wymaga zapewnienia mechanizmu dostępu do wspólnych obszarów pamięci (pamięć dzielona), czy wymusza wykonywanie wywołań systemowych za pomocą instrukcji SVC. Dodatkowo ograniczenia samego MPU nakładające restrykcje na rozmieszczenie chronionych obszarów w ściśle określonych miejscach pamięci powoduje, że rozwiązanie to przy obecnym trybie działania MPU nie jest zbyt wygodne.

Podstawy ochrony pamięci przez MPU

Dla przypomnienia, architektura **ARMv7m** (w stosunku do architektury **ARMv7a**) ma uproszczony model programowy (**rysunek 1**). Procesor może znajdować się w jednym z dwóch trybów wykonywania kodu: **Thread mode** (wówczas procesor wykonuje program główny) lub **Handler mode** (procesor wykonuje procedurę obsługi wyjątku). Dodatkowo procesor w danym momencie może znajdować się w jednym z dwóch trybów uprzywilejowania: **privileged** zarezerwowanym dla systemu operacyjnego, oraz **user** zarezerwowanym dla aplikacji użytkownika. Gdy procesor znajduje się w trybie wykonywania kodu programu (**thread)**, wówczas może pracować z uprawnieniami użytkownika **user** lub z uprawnieniami uprzywilejowanymi **privileged**. Podczas wykonania kodu obsługi wyjątku procesor zawsze pracuje na poziomie uprzywilejowania **privleged**. Procesor znajdujący się w trybie **thread** i poziomie uprzywilejowania **privileged**, może przejść do poziomu uprzywilejowania **user** poprzez zmianę bitu 0 w rejestrze **CONTROL**. Zmiana z poziomu uprzywilejowania **user** na poziom **privileged** może nastąpić jedynie w wyniku wystąpienia wyjątku lub wywołania instrukcji *SVC*, służącej do realizacji wywołań systemu operacyjnego.

Domyślnie w trybie uprzywilejowania **user** dostęp do rejestrów znajdujących się w przestrzeni **SCS** *(System Control Space)* jest zablokowany i odwołanie się do nich spowoduje wystąpienie wyjątku. Jednak dostęp do pozostałych obszarów pamięci *RAM* czy *Flash* pozostaje nieograniczony. Aby umożliwić w pełni konfigurowalna ochronę obszarów pamięci w architekturze *ARMv7m* wykorzystuje się jednostkę ochrony pamięci MPU. Zapewnia ona ochronę poprzez definicję regionów pamięci, którym można przypisać odpowiednie atrybuty ochrony, w zależności od poziomu uprzywilejowania procesora, oraz zdefiniować sposób działania pamięci cache. Domyślnie po restarcie jednostka ochrony pamięci jest wyłączona, a procesor korzysta z domyślnej mapy pamięci przedstawionej w **tabeli 1**.

Bez włączonej jednostki ochrony pamięci praktycznie cała pamięć jest dostępna dla obu poziomów uprzywilejowania z wyłączeniem obszaru rejestrów systemowych *E0000000-E00FFFFF* dostępnych jedynie z poziomu **privileged**. Bit **XN** *(Execute Never)* służy

Tabela 1. Domyślna mapa pamięci

Adres	Region	Typ	XN	Możliwość dzielenia	Opis
0x00000000-0x1FFFFFFF	Code	Normalny			Pamięć programu (Flash)
0x20000000-0x3FFFFFFF	SRAM	Normalny			Pamięć danych. Kod również może być wykonywany
0x40000000-0x5FFFFFFF	Peripheral	Device	XN		Rejon dostępu bitowego
0x60000000-0x9FFFFFFF	Ext RAM	Normalny			Pamięć zewnętrzna
0xA0000000-0xBFFFFFFF			XN	Dzielona	Zewnętrzna pamięć urządzeń
0xC0000000-0xDFFFFFFF				Niedzielona	
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	XN	Dzielona	Obszar rejestrów systemowych. NVIC, System timer, SCB
0xE0100000-0xFFFFFFFF		Device	XN		Zarezerwowane

do wyłączenia możliwości wykonywania kodu w wybranej puli adresowej, i w przypadku domyślnej mapy pamięci bit ten ustawiony jest jedynie dla przestrzeni rejestrów układów peryferyjnych. Po włączeniu jednostki **MPU** mapę pamięci możemy podzielić na 8 indywidualnie konfigurowalnych regionów. Dodatkowo dla trybu uprzywilejowania **privileged**, możemy włączyć dodatkowy domyślny region o najniższym priorytecie (–1), który jest tożsamy z domyślną mapą pamięci. Wykorzystanie regionu domyślnego zapewnia dostęp systemowi operacyjnemu do całego obszaru pamięci bez konieczności indywidualnej konfiguracji dodatkowych regionów, których jest stosunkowo niewiele.

Dostęp do regionu pamięci, który nie jest zdefiniowany, lub do którego dostęp jest niedozwolony powoduje wygenerowanie wyjątku *Memory Management Fault*.

Każdy region ma priorytet, przez co regiony mogą nachodzić na siebie. W takim wypadku pierwszeństwo ma region o najwyższym numerze. Na przykład, jeśli żądany adres znajduje się w obszarze zdefiniowanym przez regiony 2 i 3 nachodzące na siebie, uprawnienia skonfigurowane w regionie 3 będą miały pierwszeństwo nad uprawnieniami z regionu 2 (**rysunek 2**). Najniższy priorytet ma region domyślny (–1), który jest dostępny jedynie z poziomu uprzywilejowania **privileged**. Jeśli procesor znajduje się w trybie **user**, dostęp do tego obszaru spowoduje wystąpienie wyjątku.

Każdy z ośmiu dostępnych regionów konfigurowalny jest za pomocą bazowego rejestru adresowego **MPU_RBAR** oraz rejestru atrybutów i rozmiaru **MPU_RASR** i ma następujące atrybuty:

– Numer regionu (Rejestr RBAR).
– Adres bazowy regionu (Rejestr RASR).
– Atrybuty oraz rozmiar regionu (Rejestr RASR).

Obszar zajmowany przez region determinowany jest przez bity rejestru adresowego **ADDR** określające adres początkowy oraz bity **SIZE** i **SRD** w rejestrze **RASR**, określające rozmiar. Niestety z definicją rozmiaru regionu wiąże się przykre ograniczenie wynikające z tego, że rozmiar regionu definiowany jest przez 5 bitów **SIZE** zgodnie ze wzorem $2^{(SIZE+1)}$ oraz dodatkowo adres bazowy regionu musi być wyrównany w pamięci do wielkości regionu. Na przykład, jeśli wielkość regionu jest równa 64 kB, adres bazowy również musi być podzielny przez 64 kB, co stanowi to istotną trudność w zarządzaniu pamięcią przez system operacyjny.

Jeśli rozmiar regionu jest większy niż 128 bajtów, może on zostać podzielony 8 dodatkowych podregionów, które mogą być indywidualnie wyłączane za pomocą 8 bitów rejestru **SRD** *(Subregion disable)*. Każdy z 8 bitów SRD odpowiada za 1/8 regionu. Ustawienie poszczególnych bitów powoduje, że dany obszar pamięci odpowiadający za dany fragment regionu będzie z niego wyłączony.

Najistotniejszymi atrybutami z punktu widzenia użytkownika są atrybuty praw dostępu do regionu **AP** (Access Permission), określające uprawnienia. Umożliwiają one niezależne zdefiniowanie praw dostępu w zależności od poziomu uprzywilejowania procesora według **tabeli 2**.

Tabela 2. Prawa dostępu w zależności od uprzywilejowania procesora

AP	Privileged	User	Opis
000	Denied	Denied	Obszar wyłączony – każdy dostęp powoduje wyjątek
001	RW	Denied	Dostęp jedynie z trybu uprzywilejowanego
010	RW	RO	Próba zapisu z trybu użytkownika powoduje wyjątek
011	RW	RW	Pełny dostęp
100	---	---	Zarezerwowane
101	RO	Denied	Tylko do odczytu w trybie uprzywilejowanym
110	RO	RO	Tylko do odczytu w obu trybach
111			

	privileged	User
Obsługa wyjątku	Handler Mode	
Wykonanie programu	Thread Mode	Thread Mode

Rysunek 1. Procesor może znajdować się w jednym z dwóch trybów wykonywania kodu

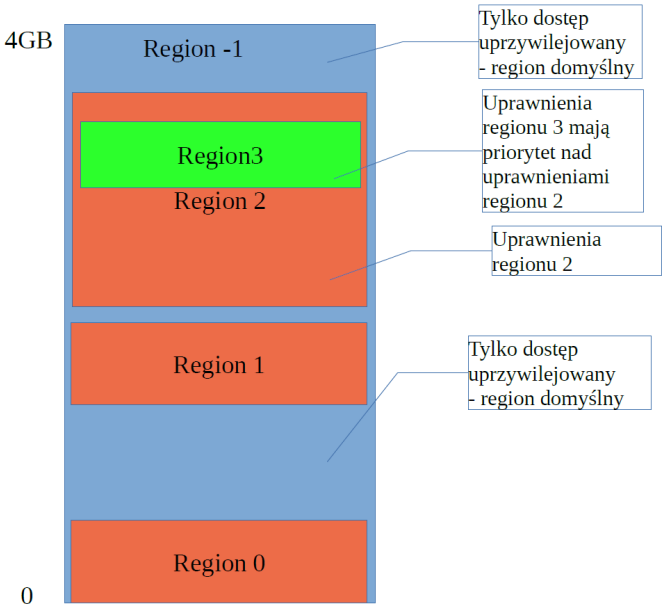
Oprócz poziomu uprawnień dla danego regionu istotny jest również bit **XN** *(Execute Never)* zabraniający wykonywania instrukcji z danego obszaru pamięci, co umożliwia implementację mechanizmu ochrony przed przepełnieniem bufora. Poza uprawnieniami dostępu możemy za pomocą dodatkowych bitów **S**, **C**, **B**, oraz **TEX** określić zachowania pamięci cache oraz skonfigurować sposób współdzielenia pamięci. Z uwagi na to że mikrokontrolery STM32 z rdzeniem *Cortex-M3/4* nie posiadają pamięci cache oraz pracują w trybie jedno-procesorowym ustawienia te mają jedynie wpływ na buforowanie zapisu i ewentualnie na kontroler pamięci.

Biblioteka obsługi MPU w systemie ISIX

Mimo iż jednostka ochrony pamięci jest stosunkowo prostym układem z niewielką ilością rejestrów konfiguracja poszczególnych regionów jest stosunkowo skomplikowana. Szczególnie, jeśli chcemy konfigurować obszary z wykorzystaniem wyłączania podregionów (bity SRD). Aby zapewnić maksymalnie uproszczoną konfiguracją biblioteka *libstm32* w systemie ISIX została uzupełniona o wsparcie dla układu **MPU**. Jest ona wykorzystywana przez system do zapewnienia podstawowej ochrony pamięci polegającej na wykrywaniu przepełnienia stosu, oraz próby dostępu do danych przez wskaźnik pusty. Biblioteka może być również wykorzystana samodzielnie w aplikacjach niekorzystających z systemu i ma nieskomplikowane API składające się z kilkunastu funkcji. Najistotniejszą częścią biblioteki jest funkcja *mpu_set_region_size* służąca do konfiguracji wybranego regionu *void mpu_set_region_size(uint32_t region, uintptr_t addr, size_t len, uint32_t flags);*. Jako argument **region** należy podać numer regionu (0…7). Jako **addr** należy podać adres początkowy regionu, jako argument **len** należy podać wielkość danego regionu. Jako parametr **flags** należy przekazać flagi służące do konfiguracji regionu. Dla uproszczenia biblioteka zapewnia szereg predefiniowanych flag:

– Zestaw flag **MPY_RGN_PERM** służy do ustawienia uprawnień do regionu dla trybu uprzywilejowania ***privileged*** oraz ***user*** według następujących kombinacji:
  • MPU_RGN_PERM_PRV_NO_USR_NO – dostęp zabroniony.
  • MPU_RGN_PERM_PRV_RW_USR_NO – dostęp tylko z poziomu **privileged**.
  • MPU_RGN_PERM_PRV_RW_USR_RO – tylko do odczytu z poziomu **user**.
  • MPU_RGN_PERM_PRV_RW_USR_RW – pełen dostęp.
  • MPU_RGN_PERM_PRV_RO_USR_NO – odczyt tylko z poziomu **privileged**.
  • MPU_RGN_PERM_PRV_RO_USR_RO – tylko do odczytu.
– Flaga **MPU_RGN_PERM_NX** jest wykorzystywana do wyłączenia możliwości wykonywania kodu w danym regionie.
– Flaga **MPU_RGN_PERIPH** lub **MPU_RGN_MEMORY** określa, czy dany obszar będzie traktowany jako obszar urządzeń peryferyjnych, czy jako region zwykłej pamięci. Definicję te zawierają odpowiednią kombinację flag TEX, oraz B, C odpowiedzialnych za pamięć cache oraz współdzielenie obszarów pamięci.
– Ustawienie flagi **MPU_RGN_ENABLE** lub **MPU_RGN_DISABLE** określa czy wybrany region jest aktywny.

Funkcja na podstawie przekazanego rozmiaru dobierze rozmiar **SIZE** oraz ustawi bity subregionów **SRD** w taki sposób, aby żądany region był równy lub większy niż żądana wielkość przekazana za pomocą argumentu *len*. Oprócz możliwości konfiguracji i włączenia



Rysunek 2. Jeśli żądany adres znajduje się w obszarze zdefiniowanym przez regiony 2 i 3 nachodzące na siebie, uprawnienia skonfigurowane w regionie 3 będą miały pierwszeństwo nad uprawnieniami z regionu 2

lub wyłączenia danego regionu mamy możliwość włączenia lub wyłączenia samego układu MPU, za pomocą funkcji:

```
void mpu_enable( uint32_t config );
void mpu_disable(void);
```

Funkcja włączająca **MPU** przyjmuje kombinację flag **MPU_CONFIG_PRIV_DEFAULT** włączającą domyślną mapę pamięci (region -1) dla trybu uprzywilejowanego, oraz M**PU_CONFIG_HARDFLT_NMI** włączającą mapę domyślną dla obsługi wyjątku *hard fault*, lub **MPU_CONFIG_NONE** powodującą wyłączenie regionu domyślnego.

Do wykrywania obecności układu MPU podczas działania programu możemy wykorzystać funkcję uint32_t mpu_get_region_count(void);. Funkcja ta zwraca liczbę dostępnych regionów. W przypadku, gdy zwracana jest wartość jest równa zero jednostka **MPU** jest niedostępna. Wywołanie funkcji związanych z **MPU** powinno być realizowane przy wyłączonych przerwaniach, (np. za pomocą instrukcji **cpsid i**), aby podczas manipulacji obszarami regionów nie aktywowało się żadne przerwanie.

Bardzo popularnym mechanizmem w funkcjach zwracających wskaźnik do obiektu/struktury jest zwracanie wartości specjalnej *NULL/nullptr (0)* w przypadku wystąpienia błędu. Często zdarzają się sytuację, w których wskaźnik zwracany przez funkcję nie jest prawidłowo sprawdzany i następuje próba zapisu/odczytu za pomocą pustego wskaźnika. W mikrokontrolerach STM32 dodatkowo nie pomaga fakt, iż obszar 0x00000000 w mapie pamięci stanowi alias dla pamięci Flash znajdującej się pod adresem 0x08000000. O ile instrukcja wykonująca zapis pod adres zerowy spowoduje wygenerowanie wyjątku, o tyle instrukcja odczytu spowoduje odczytanie danych z początku pamięci Flash, cco może powodować ujawnienie błędu w miejscu zupełnie innym niż wystąpiła przyczyna. Aby zapobiec takiej sytuacji możemy wykorzystać jeden z regionów MPU do zablokowania możliwości dostępu do obszaru znajdującego się w początkowej części mapy pamięci.

Innym bardzo często spotykanym problemem jest błąd związany z przepełnieniem bufora, który może zostać wykorzystany do uruchomienia niepożądanego kodu. Dla mikrokontrolerów STM32 domyślnie zdefiniowana mapa pamięci umożliwia wykonanie kodu bezpośrednio z pamięci RAM. Aby zapobiec temu zjawisku możemy wykorzystać jeden z regionów MPU celem ustawiania flagi **XN** *(Execute Never)*, w obszarze pamięci RAM rozpoczynającej się od adresu 0x20000000.

Listing 1. Nieskomplikowane zabezpieczenie dostępu do pamięci

```
#include <arm-v7m/mpu.h>

int main()
{
    // SET XN bit in the internal RAM area
    mpu_set_region_size( 0, 0x20000000, 0x4000000 )
            MPU_RGN_PERM_PRV_RW_USR_RW|
            MPU_RGN_PERM_NX|
            MPU_RGN_MEMORY );
    //!Protect region of the flash code alias
    mpu_set_region( 1, 0x0, 0x100000
            MPU_RGN_PERM_PRV_NO_USR_NO|
            MPU_RGN_MEMORY|
            MPU_RGN_PERM_NX );

    mpu_enable_region(0);
    mpu_enable_region(1);
    mpu_enable( MPU_CONFIG_PRIV_DEFAULT );

    // Your program here
    return 0;

}
```

Przykład użycia biblioteki **libstm32** ustawiający wyżej wspomniane zabezpieczenia dostępu do pamięci przedstawiono na **listingu 1**. W programie najpierw ustawiamy region 0 na obszar pamięci RAM i blokujemy możliwość wykonania kodu ustawiając flagę MPU_RGN_PERM_NX. Następnie region 1 konfigurujemy tak, aby zablokować całkowicie dostęp do aliasu pamięci FLASH znajdującego się pod adresem 0x0. Następnie za pomocą funkcji **mpu_enable_region** włączamy regiony 0-1. Na koniec włączamy MPU jednocześnie aktywując domyślną mapę pamięci dla poziomu uprzywilejowania **privileged.** Po skonfigurowaniu i uruchomieniu MMU próba uruchomienia kodu z obszaru wewnętrznej pamięci RAM, czy próba dostępu do do adresów w zakresie 0x0-0x100000 spowoduje wygenerowanie wyjątku *Memory Management Fault*.

MPU w systemie ISIX

W systemie ISIX wykorzystano uproszczony model ochrony polegający na ochronie programu przed wykonaniem instrukcji dostępu do pamięci z wykorzystaniem wskaźnika pustego, oraz ochronę przed wykonywaniem kodu z obszaru pamięci RAM. Dodatkowo zastosowano mechanizm ochrony granicy stosu służący do wykrywania przepełnienia stosu. Uproszczony model ochrony podyktowany został dość istotnym ograniczeniem MPU wynikającym z konieczności wyrównania adresu początkowego regionu do rozmiaru obszaru. Wykrywanie przepełnienia stosu jest realizowane przez ustawienie ochronnego regionu z flagami MPU_RGN_PERM_PRV_NO_USR_NO o wielkości 32 bajtów na granicy stosu w momencie zaszeregowania wątku do wykonania. Przy próbie zapisu w rejon obszaru chronionego zostanie wygenerowany wyjątek, który pozwoli wykryć przepełnienie stosu. Naturalnie tak mały obszar chroniony nie zabezpiecza użytkownika wszystkimi sytuacjami, np. zapisem przekraczającym obszar chroniony. System ISIX domyślnie po uruchomieniu sprawdza dostępność jednostki MPU i jeśli zostanie ona wykryta, wówczas automatycznie ustawia podstawowy tryb ochrony pamięci. Zatem korzystając z systemu nie musimy konfigurować samodzielnie MPU. Aby zademonstrować podstawowe działanie mechanizmów ochrony pamięci przygotowano bardzo prostą aplikację dla zestawu ZL41ARM.

Aby ją uruchomić dodatkowo będziemy potrzebowali:

- Przejściówki USB-Serial w standardzie TTL dołączonej do linii: *PD5 – TX* oraz *PD4 – RX*.
- 3 przełączników zwierających linię: *PC12*, *PC13*, *PC14* do masy.
- Diody LED dołączonej za pomocą rezystora *470R* do linii portu *PE14*.

Po przyłączeniu urządzeń peryferyjnych należy zaprogramować zestaw ZL41ARM plikiem wynikowym *mpudemo.hex*, a następnie uruchomić terminal szeregowy w konfiguracji 115200, n, 8, 1. Po uruchomieniu zestawu na terminalu powinien pojawić się komunikat powitalny, oraz dioda podłączona do linii PE14 powinna mrugać.

Listing 2. Podstawowe informacje przesyłane za pomocą portu szeregowego

```
ISIX panic! Exception in [USER] mode.
Last executed task TCB is 20001588
CPU core regs:
    [R0=00000000]   [R1=200015f8]  [R2=200015fc]  [R3=000184b0]
    [R12=20000114]  [LR=080033a5]  [PC=080033b4]
CPSR flags:
    EXCNO:0 ICI10:0 T:1 ICI25:0 Q:0 V:0 C:1 Z:0 N:0
HFSR Hard Fault Status Register bits:
    DEBUGEVT:0 FORCED:1 VECTBL:0
Memory Management Fault exception occured. MMAR status:
    MMARVALID:1 STKERR:0 UNSTKERR:0 DACCVIOL:1 IACCVIOL:0
    Bus Fault address BFAR: 200015fc
Stack dump:
    20001db8: 00000000 200015fc 20000114 08003399
    20001dc8: 20000000 60000000 00000000 200015fc
    20001dd8: 20000114 0800331f 20000114 0800169d
    20001de8: 00000000 00012010 ffffffff ffffffff
    20001df8: ffffffff ffffffff ffffffff ffffffff
    20001e08: ffffffff ffffffff ffffffff ffffffff
    20001e18: ffffffff ffffffff ffffffff ffffffff
    20001e28: ffffffff ffffffff ffffffff ffffffff
```

Listing 3.

```
1  for(;;)
2  {
3      bool knull = stm32::gpio_get( KEY_PORT, NULL_PIN );
4      bool kexec = stm32::gpio_get( KEY_PORT, EXEC_PIN );
5      bool kstk =  stm32::gpio_get( KEY_PORT, STK_PIN );
6      if( !knull && p_null ) {
7          dbprintf( „Trying to read from null pointer” );
8          mpu_demo* nptr = nullptr;
9          dbprintf( „Test %i”, nptr->p_null );
10     }
11     if( !kexec && p_exec ) {
12         dbprintf( „Trying to exec code from ram” );
13         void(*pfn)() = reinterpret_cast<void(*)()>(0x20000000);
14         pfn();
15     }
16     if( !kstk && p_stk ) {
17         dbprintf( „Trying to overflow stack” );
18         int lvar;
19         volatile auto pvar = &lvar;
20         for(int i=0;i<100000;++i ) {
21             *pvar-- = 0;
22         }
23     }
24     p_null = knull; p_exec = kexec; p_stk = kstk;
25     isix::wait_ms( 25 );
26 }
```

Po wciśnięciu przycisku podłączonego do linii PC12, PC13, PC14, będziemy mogli wygenerować odpowiednio: próbę odczytu za pośrednictwem pustego wskaźnika, próbę wykonania kodu z pamięci RAM, oraz próbę zapisu danych do obszaru przekraczającego rozmiar stosu. W wyniku wykonania błędnej instrukcji nieuprawniony dostęp do pamięci zostanie wykryty przez MPU, co spowoduje wygenerowanie wyjątku. W procedurze obsługi wywoływana jest funkcja diagnostyczna *cm3_hard_hault_regs_dump* wypisująca na port szeregowy podstawowe informację na temat wyjątku oraz stanu rejestrów procesora (**listing 2**).

Powyższy obraz zawiera informację wypisaną na konsoli szeregowej powstałej w wyniku próby zapisu poza obszarem stosu. Jak pamiętamy na końcu stosu znajduje się 32 bajtowy obszar ochronny, w który trafia błędna instrukcja co powoduje wygenerowanie wyjątku *Memory Management Fault*. Na podstawie rejestru **PC** możemy określić adres instrukcji która spowodowała wyjątek, natomiast na podstawie rejestru **BFAR** *(Bus Fault Address Register)*, możemy sprawdzić adres pamięci do której błędna instrukcja próbowała uzyskać dostęp. Ustawienie bitu **DACCVIOL** w rejestrze **MMAR** informuje o tym, że mamy do czynienia z naruszeniem ochrony dostępu przez instrukcje próbującą odczytać dane.

Zaprezentowany przykład oparty jest o dwa wątki: jeden watek realizowany w klasie **ledblink** mruga diodą dołączoną do linii *PE14*, natomiast drugi realizowany przez klasę **mpu_demo**, sprawdza stan wciśnięcia klawiszy i wywołuje fragmenty kodu symulujący błędy. Główną pętle wątku z klasy *mpu_demo* odpowiedzialną za wygenerowanie błędów przedstawiono na **listingu 3**.

Na początku odczytywany jest stan klawiszy i jeśli wykryte zostanie zbocze opadające wywołana odpowiednia funkcja symulująca błąd. Dla pinu przypisanego do portu *NULL_PIN (PC12)* jest to odpowiednio przypisanie wartości **nullptr** do wskaźnika na klasę **mpu_demo**, a następnie próba odczytu danych z tego wskaźnika. Po wykryciu wciśnięcia przycisku *EXEC_PIN (PC13)* tworzony jest wskaźnik do funkcji, do którego przypisywany jest adres początku obszaru pamięci RAM, a następnie próbujemy wywołać tą funkcję. W przypadku wykrycia wciśnięcie klawisza *STL_PIN (PC14)*, wskaźnik ***lvar*** inicjowany jest zawartością zmiennej lokalnej, a następnie przechodząc przez kolejne fragmenty nadpisujemy obszar stosu przekraczając jego rozmiar. Wszystkie te czynności wywołane za pomocą wciśnięcia odpowiedniego klawisza spowodują wykrycie naruszenia ochrony pamięci, i wygenerowanie wyjątku.

Zakończenie

Większość mikrokontrolerów z rdzeniem Cortex-M3/4 posiada wbudowaną jednostkę MPU, której wykorzystanie pozwala zwiększyć niezawodność budowanych urządzeń. Skonfigurowanie jednostki MPU do wykrywania podstawowych problemów jest zadaniem stosunkowo prostym i poza konfiguracją samego układu nie wymaga, żadnych dodatkowych zmian w programie. Jeśli chcemy zapewnić pełną ochronę i separacje między zadaniami, konieczne są większe i bardziej inwazyjne zmiany. Niestety konieczność spełnienia warunku, że adres początku chronionego regionu musi być wielokrotnością tego regionu, oraz brak możliwości ustalenia dokładnego rozmiaru regionu powoduje iż wykorzystanie z MPU realizacji całkowitej ochrony międzyprocesowej jest stosunkowo kłopotliwe i powoduje marnotrawienie cennych zasobów pamięciowych, których w mikrokontrolerze mamy do dyspozycji stosunkowo nie wiele.

Lucjan Bryndza, EP