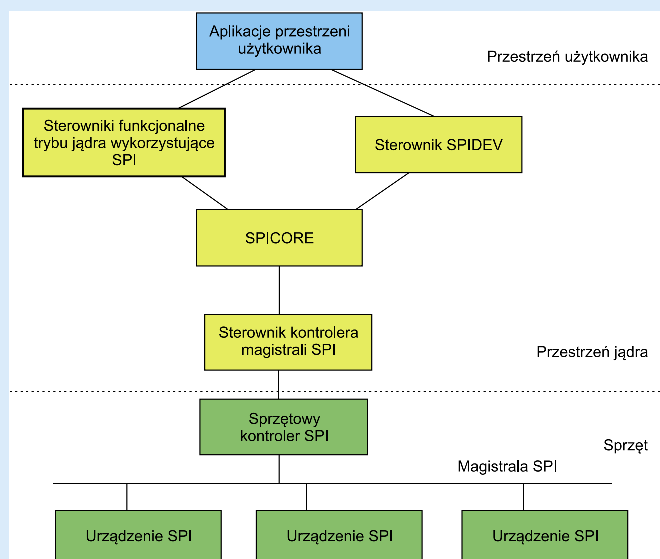
Wprowadzenie do Linuksa embedded (7)

Obsługa interfejsu SPI

W związku z ogromnym zainteresowaniem artykułami z cyklu „Wprowadzenie do Linuxa Embedded" oraz związanymi z nimi pytaniami o obsługę urządzeń dołączonych do magistrali SPI, w ramach uzupełnienia cyklu opisujemy sposób obsługi SPI z poziomu jądra systemu Linuksa.

Interfejs SPI zawdzięcza swoją popularność przede wszystkim swojej niezwykłej łatwości zastosowania oraz dużej szybkości działania, która może być znacznie większa od szybkości transmisji za pomocą I²C. Implementacja interfejsu jest łatwa zarówno po stronie urządzenia peryferyjnego, jak i mikrokontrolera. Dodatkowo, wymaga niewielkich zasobów sprzętowych. Może ona być zrealizowana za pomocą specjalizowanego układu peryferyjnego, interfejsu sprzętowego mikrokontrolera lub emulowana za pomocą linii GPIO. Podobnie jak dla interfejsu I²C w Linuksie istnieje specjalny podsystem obsługi SPI, pozwalający używać funkcji obsługi tego interfejsu w modułach przestrzeni jądra. A dodatkowo, sterownik spidev umożliwia dostęp z poziomu aplikacji przestrzeni użytkownika.

W artykule zapoznamy się z: podsystemem SPI; mechanizmem mapowania sterowników jądra do urządzeń ,a w przykładzie praktycznym ze sposobem obsługi interfejsu SPI z poziomu aplikacji użytkownika na przykładzie modułu KAMOD-MEMS2, który zawiera akcelerometr komunikujący się przez SPI. W przykładzie użyjemy modułu spidev, a rezultat działania będziemy wyświetlać na standardowym wyjściu (konsoli).



Rysunek 1. Model podsystemu magistrali SPI

Obsługa interfejsu SPI w systemie Linux

Implementacja podsystemu SPI charakteryzuje się modelem warstwowym dzięki czemu system jest bardzo elastyczny i uwzględnienia możliwości działania wielu magistral, wielu kontrolerów magistrali, jak i wielu urządzeń podłączonych do nich. Jest jest bardzo podobny do omówionego w poprzednim podrozdziale podsystemu dla magistrali I²C. Na **rysunku 1** przedstawiono model podsystemu magistrali SPI.

Sercem podsystemu jest SPICORE, który z jednej strony komunikuje się ze sterownikiem sprzętowego kontrolera magistrali SPI, zaś z drugiej udostępnia niezależne od sprzętu API, umożliwiające innym sterownikom trybu jądra korzystanie z jednolitego API. Sterownik kontrolera magistrali SPI jest odpowiedzialny za komunikację ze sprzętowym kontrolerem SPI, udostępniając jednolite API, które jest wykorzystywane przez SPICORE. Sterownik może stanowić moduł obsługujący specjalizowany sprzętowy kontroler SPI (dla platformy at91 jest to atmel_spi), lub w wypadku jego braku może to być moduł generujący sygnały magistrali SPI programowo za pomocą linii GPIO. Istnieje również możliwość bezpośredniego dostępu do magistrali z poziomu aplikacji użytkownika, co realizuje moduł SPIDEV, który udostępnia dobrodziejstwa API SPI trybu jądra aplikacjom przestrzeni użytkownika. Jednak w przeciwieństwie do omówionego wcześniej interfejsu I²C, z tego powodu, że adresowanie urządzeń dołączonych do SPI odbywa się z wykorzystaniem linii sprzętowych CS oraz na różne dostępne tryby pracy SPI, podsystem wymaga zdefiniowania i zarejestrowania struktury platform device w kodzie jądra, która będzie zawierać informację o parametrach konfiguracji SPI przypisanych do danej magistrali oraz linii CS.

Każde urządzenie SPI, przypisane do danej linii CS, będzie widoczne w przestrzeni użytkownika jako plik /dev/spiX.Y (gdzie X to numer porządkowy interfejsu SPI, natomiast Y to numer porządkowy linii CS). Jak już wielokrotnie wspominano we wcześniejszych odcinkach, sam interfejs SPI nie jest urządzeniem realizującym konkretne zadanie. Poprawnym sposobem wykorzystania urządzenia dołączonego do SPI (oraz innych interfejsów) jest napisanie sterownika w przestrzeni ją-

dra, udostępniającego jego interfejs funkcjonalny, z wykorzystaniem API jądra zamiast używania dostępu do samej magistrali i sterowanie urządzeniem bezpośrednio z aplikacji. Niemniej jednak, bezpośredni dostęp do magistrali może być bardzo użyteczny do pisania prototypowego kodu, sterowników, czy dla nietypowych urządzeń wykorzystywanych tylko z pojedynczej aplikacji, gdy użytkownik spodziewa się, że kod będzie zmieniany dosyć często.

Konfigurowanie urządzeń do dołączonych do SPI w jądrze

Mikrokontroler AT91RM9200 ma zintegrowany kontroler SPI (niestety tylko jeden), który jest w pełni sprawny w przeciwieństwie do kontrolera I²C. Ten sterownik ma możliwość sprzętowego generowania sygnałów CS za pomocą 4 linii, a zatem możemy do niego dołączyć 4 niezależne urządzenia SPI slave. Sterownik ma możliwość pracy w 4 podstawowych trybach SPI, z konfigurowalną długością słowa od 8 do 16 bitów i kolejnością bitów. Sprzętowy kontroler SPI mikrokontrolera RM9200 jest obsługiwany przez moduł sterownika atmel-spi. Z uwagi na konieczność zdefiniowania trybu pracy urządzenia oraz linii CS, nawet wykorzystanie bezpośredniego sterowania z aplikacji użytkownika za pomocą modułu spidev wymaga skonfigurowania odpowiednich struktur platform device w kodzie jądra odpowiedzialnego za konfigurację maszyny (plik /arch/arm/mach-at91/board-boff210.c). W podsystemie SPI urządzenie reprezentowane jest przez strukturę spi_device (**listing 1**).

Pole max_speed_hz określa maksymalną dopuszczalną częstotliwość sygnału CLK urządzenia. Pole chip_select określa numer linii CS, do której jest dołączone dane urządzenie W mikrokontrolerze RM9200 będzie to odpowiednio 0 – NPCS0, 1- NPCS1, 2-NPCS2 i 3-NPCS3. Pole mode określa tryb pracy urządzenia, które mogą być wartościami od SPI_MODE_0 do SPI_MODE3, co zapewnia możliwość współpracy w zasadzie z każdym urządzeniem wyposażonym w SPI. Dodatkowo, można wyspecyfikować flagi konfiguracyjne, które będą określały zachowanie się kontrolera. Na przykład, ustawienie dodatkowego bitu SPI_LSB_FIRST spowoduje, że najstarszy bit będzie wysyłany jako pierwszy. Pole bits_per_word określa ile bitów będzie przesyłanych w wypadku transmisji pojedynczego słowa, gdzie dopuszczalne wartości mieszczą się w zakresie 8…16. Pole modalias określa nazwę modułu (sterownika urządzenia), który ma zostać załadowany, aby obsłużyć urządzenie. Jeśli w jądrze istnieje gotowy sterownik funkcjonalny, należy tutaj wpisać jego nazwę, natomiast jeśli brak, to można wpisać spidev, co spowoduje przypisanie temu urządzeniu sterownika interfejsu SPI przestrzeni użytkownika.

W przykładzie praktycznym demonstrującym przykład użycia urządzeń z magistralą SPI wykorzystano moduł KAMOD-MEMS2, który zawiera akcelerometr LIS35, dołączony do linii NPCS1. Akcelerometr ten w konfiguracji z interfejsem SPI pracuje w trybie SPI_MODE_3, MSB. Do magistrali SPI , na płycie BF210 jest dołączona także pamięć Flash z interfejsem SPI , która wykorzystuje linię wyboru NPCS0. W wypadku tych dwóch urządzeń w pliku board-boff210 definiujemy tablicę, jak na **listingu 2**.

Listing 1. Struktura spi_device

```
#define SPI_CPHA     0x01   /* clock phase */
#define SPI_CPOL     0x02   /* clock polarity */
#define SPI_MODE_0   (0|0)  /* (original MicroWire) */
#define SPI_MODE_1   (0|SPI_CPHA)
#define SPI_MODE_2   (SPI_CPOL|0)
#define SPI_MODE_3   (SPI_CPOL|SPI_CPHA)
#define SPI_CS_HIGH 0x04  /* chipselect active high? */
#define SPI_LSB_FIRST 0x08  /* per-word bits-on-wire */
#define SPI_3WIRE    0x10   /* SI/SO signals shared */
#define SPI_LOOP     0x20   /* loopback mode */
#define SPI_NO_CS    0x40   /* 1 dev/bus, no chipselect */
#define SPI_READY    0x80   /* slave pulls low to pause */

struct spi_device {
  struct device       dev;
  struct spi_master  *master;
  u32  max_speed_hz;
  u8   chip_select;
  u8   mode;
  u8    bits_per_word;
  int   irq;
  void  *controller_state;
  void  *controller_data;
  char  modalias[SPI_NAME_SIZE];
};
```

Listing 2. Tablica zdefiniowana dla pamięci Flash i akcelerometru

```
static struct spi_board_info bf_spi_devices[] = {
  {  /* DataFlash chip */
    .modalias       = "mtd_dataflash",
    .chip_select    = 0,
    .max_speed_hz   = 15 * 1000 * 1000,
  },
  {
    .modalias = "spidev",
    .chip_select  = 1,
    .max_speed_hz = 10 * 1000 * 1000,
    .mode = SPI_MODE_3
  }
};
```

Pierwszy element tablicy opisuje konfigurację pamięci Flash, której przypisano sterownik mtd_dataflash oraz linię wyboru układu NPCS0. Drugi element tablicy opisuje konfigurację akcelerometru, któremu przypisano sterownik spidev, linię wyboru układu NPCS1 oraz tryb pracy SPI_MODE3. Tak przygotowana tablica opisująca urządzenia SPI jest następnie rejestrowana w platform data za pomocą wywołania:

```
/* SPI */
at91_add_device_spi(bf_spi_devices,
ARRAY_SIZE(bf_spi_devices));
```

Należy tutaj zaznaczyć, że w wypadku użycia sterownika przestrzeni użytkownika spidev istnieje możliwość zmiany zdefiniowanych wcześniej parametrów konfiguracyjnych za pomocą odpowiednich wywołań ioctl(). Jedyną rzeczą, której nie można później zmienić jest przypisanie linii CS w polu chip_select. W wypadku opisanej konfiguracji, pamięć Flash (CS= NPCS0) będzie obsługiwana przez sterownik funkcjonalny podsystemu MTD (Memory Technology Devices), natomiast akcelerometr (CS=NPCS1) będzie obsługiwany przez sterownik bezpośredniego dostępu do magistrali SPI (spidev). Plik reprezentujący to urządzenie będzie miał nazwę /dev/spi0.1.

API dostępu do magistrali SPI z aplikacji przestrzeni użytkownika

Bezpośredni dostęp do magistrali SPI , z aplikacji przestrzeni użytkownika i obsługa układu bezpośrednio z tej aplikacji może być rozwiązaniem wartym rozważenia na etapie prototypowania kodu sterownika czy komunikowania się z innym komputerem/mikrokontrolerem, gdzie potencjalnie może wystąpić częsta zmiana protokołu. Jest to również stosunkowo dobre rozwiązanie w wypadku, gdy nie mamy jeszcze dużego doświadczenia w pisaniu sterowników jądra i nie będzie potrzeby dostępu do sterowanego układu z wielu

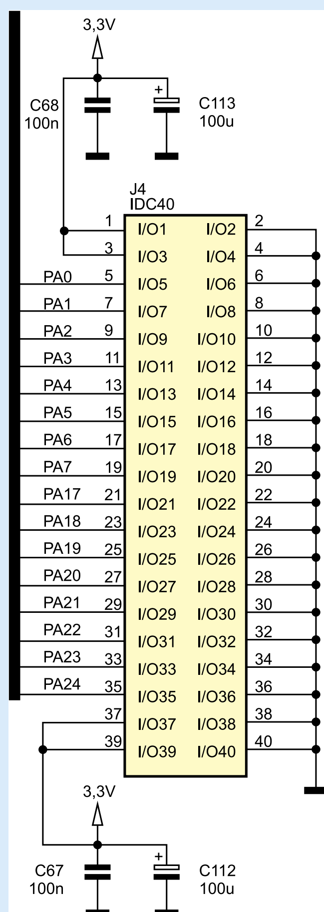
Listing 3. Definicja struktury opisującej transfer danych za pomocą SPI

```
struct spi_ioc_transfer {
  __u64           tx_buf;
  __u64           rx_buf;
  __u32           len;
  __u32           speed_hz;
  __u16           delay_usecs;
  __u8            bits_per_word;
  __u8            cs_change;
  __u32           pad;
};
```

Tabela 1. Lita dostępnych wywołań

Argument SPI_IOX_xxx	Opis/parametry
SPI_IOC_RD_MODE SPI_IOC_WR_MODE	Umożliwia ustawienie lub odczytanie trybu pracy magistrali SPI, Dopuszczalne parametry to SPI_MODE_0 do SPI_MODE_3 , które umożliwiają wybór sposobu pracy magistrali.
SPI_IOC_RD_LSB_FIRST SPI_IOC_WR_LSB_FIRST	Umożliwiają odczyt lub zmianę kolejności bitów wysyłanych magistralą. Wartość równa zero oznacza tryb pracy MSB, natomiast wartość różna od zera tryb pracy LSB
SPI_IOC_RD_BITS_PER_WORD SPI_IOC_WR_BITS_PER_WORD	Umożliwia odczytanie lub zapisanie, liczby bitów które będą przesyłane magistralą SPI podczas transmisji pojedynczego słowa
SPI_IOC_RD_MAX_SPEED_HZ SPI_IOC_WR_MAX_SPEED_HZ	Pozwala na zapisanie lub odczytanie maksymalnej dopuszczalnej prędkości transmisji SPI.

aplikacji. Funkcjonalność taką udostępnia moduł spidev, który zapewnia aplikacji przestrzeni użytkownika dostęp do API SPI. Poszczególne urządzenia, zdefiniowane w strukturach platform data, które zadeklarowały użycie sterownika spidev, widoczne będą jako następujące pliki: /dev/spi0.0 – urządzenie używające jako linii CS NPCS0, /dev/spi0.1 – urządzenie używające jako linii CS NPCS1 itd. Aby uzyskać dostęp do API SPI należy dołączyć plik nagłówkowy `#include <linux/spi/spidev.h>`. Aby skorzystać z dostępu do magistrali SPI danego urządzenia, wystarczy otworzyć plik odpowiadający temu urządzeniu np. /dev/spi0.1 za pomocą wywołania systemowego `open( "/dev/spidev0.1",O_RDWR )`, a następnie możemy używać standardowych wywołań systemowych read(), write() w celu zapisania lub odczytania danych z magistrali SPI. Jeżeli wcześniej prawidłowo zdefiniowaliśmy wpisy w strukturze spi_board_info, nie będzie potrzeby wywoływania, dodatkowej konfiguracji SPI. Niestety, samo wywołania read() lub write() umożliwiają jedynie odczyt danych w jednym kierunku, a jak wiemy magistrala SPI pracuje



Rysunek 2. Sposób dołączenia modułu KAMOD-MEMS2 do płytki prototypowej BF210 (KAMOD-MEMS2: V+ - J4PIN1, SS- J4PIN13, MOSI-J4PIN7, MISO-J4PIN5, SCK-J4PIN9, GND – J4PIN2)

w pełni duplexowo i następuje jednoczesne odbieranie i wysyłanie danych. Zatem w przypadku wywołania write(), dane odebrane będą tracone, lub w przypadku wywołania read() odbierane dane będą przekazywane do bufora, natomiast do linii nadawczej będzie przekazywany ciąg bajtów 0xff. W sytuacji, gdy istnieje konieczność komunikacji w trybie full-duplex, z jednoczesnym nadawaniem i odbiorem znaków, należy użyć wywołania systemowego `ioctl(handle, SPI_IOC_MESSAGE(n), xfer)`. Jako parametr n należy przekazać liczbę elementów tablicy xfer. Zmienna ta powinna być zadeklarowana jako tablica struktur spio_ioc_transfer, która opisuje transfery danych na magistrali SPI. Definicję struktury przedstawiono na **listingu 3**.

Pole tx_buf powinno zawierać adres bufora z danymi, które będą zapisane na magistralę SPI. Pole rx_buf zawiera adres bufora, gdzie trafią dane odczytane podczas transferu z magistrali SPI. Pole len zawiera wielkość transferu SPI wyrażoną w bajtach. Pole speed_hz , zawiera prędkość transferu danych danego pakietu. Jeżeli wartość tego pola jest równa zero, wówczas zostanie użyta domyślna prędkość transmisji, zdefiniowana w strukturach platform device. Jeżeli pole delay_usec jest różne od 0, określa ono czas wyrażony w mikrosekundach, po którym zostanie deaktywowany sygnał CS i będzie możliwy kolejny transfer. Pole cs_change, określa czy pomiędzy każdym transferem, zdefiniowanym w tablicy struktur spi_ioc_transfer, będzie ponownie aktywowany i deaktywowany sygnał CS. Definiując zatem tablicę struktur określającą transfery mamy możliwość pełnej kontroli sterowania magistralą SPI, i wykorzystania transmisji dupleksowej. Po otwarciu urządzenia możemy od razu przejść do transmisji danych, ponieważ wykorzystana będzie bieżąca konfiguracja z trybu jądra. Niemniej jednak z przestrzeni użytkownika istnieje możliwość, zmiany praktycznie wszystkich domyślnych ustawień (poza linią wyboru sygnału CS), za pomocą wywołań ioctl(). Struktura wywołań jest następująca:

```
long param;
ioctl(handle, SPI_IOC_xxx, &param);
```

gdzie jako SPI_IOC_xxx należy podać typ parametru SPI, który chcemy odczytać lub zmienić. Listę dostępnych wywołań wraz z krótkim opisem przedstawiono w **tabeli 1**.

Korzystając zatem z odpowiedniego wywołania mamy możliwość, zmiany parametrów, które zostały wcześniej zdefiniowane w kodzie inicjalizacyjnym w jądrze, bez konieczności modyfikacji i rekompilacji jądra.

Przykład praktyczny

Do zademonstrowania przykładu praktycznego wykorzystania bezpośredniego dostępu do magistrali SPI, użyjemy modułu KAMOD-MEMS2. Moduł ten zawiera bardzo popularny obecnie w urządzeniach akcelerometr LIS35D, który do komunikacji może wykorzystywać magistralę SPI. Działanie przykładu będzie polegać na cyklicznym odczytywaniu aktualnej pozycji czujnika ,x,y,z i wyświetlaniu z odświeżaniem jednosekundowym aktualnej pozycji na standardowym wyjściu. Sposób dołączenia modułu KAMOD-MEMS2, do płytki prototypowej BF210 przedstawiono na **rysunku 2**.

Akcelerometr LIS35D charakteryzuje się następującymi parametrami:

- Pobór mocy poniżej 1 mW.

DR	PD	FS	0(1)	0(1)	Zen	Yen	Xen

Rysunek 3. Bity rejestru CR1 (0x20)

SIM	BOOT	--	FDS	HP_FF_WU2	HP_FF_WU1	HP_coeff2	HP_coeff1

Rysunek 4. Bity rejestru CR2 (0x21)

- Komunikacja za pomocą interfejsów SPI oraz I²C.
- Dwa programowalne generatory przerwań.
- Detekcja pojedynczych i podwójnych puknięć oraz swobodnego spadku.
- Wbudowany filtr górnoprzepustowy.

Jednym z ograniczeń pisania sterowników w przestrzeni użytkownika, jest niemożność obsługi przerwań, z tego właśnie powodu w bieżącym przykładzie nie została wykorzystana możliwość zgłaszania przerwań przez akcelerometr. Do obsługi podstawowej funkcjonalności akcelerometru potrzebować będziemy potrzebować znajomości jedynie kilkunastu rejestrów. Bity rejestru CR1 (0x20) przedstawiono na **rysunku 3**.

Bit DR umożliwia konfigurację aktualizacji pozycji, znaczenie jego jest następujące: 0- aktualizacja 100 Hz, 1 – aktualizacja 400 Hz. Bit PD umożliwia przejście akcelerometru w tryb uśpienia (0 – układ nieaktywny, 1 – układ aktywny). Bity Xen, Zen, Yen umożliwiają włączenie lub wyłączenie pomiarów dla danej osi (0 – oś wyłączona , 1 – oś włączona). Bity rejestru CR2 (0x21) przedstawiono na **rysunku 4**.

Bit SIM określa tryb pracy interfejsu SPI (0 – tryb pracy 4 przewodowy, 1 – tryb pracy 3 przewodowy) Bit BOOT, jest używane dla odświeżenia zawartości wewnętrznych rejestrów zapisanych w bloku pamięci Flash powinien być aktywowany po podłączeniu zasilania. Bit FDS określa czy filtr górnoprzepustowy jest włączony (1), czy wyłączony (0). Bity HP_coeff2 i HP_coeff1 umożliwiają konfigurację pasma filtru górnoprzepustowego. Kolejnymi rejestrami wykorzystanymi w naszej aplikacji są rejestry OUT_X (0x29), OUT_Y (0x2b) OUT_Z (0x2d), które zawierają – 8 bitową liczbę ze znakiem określającą pozycje osi X, Y, Z.

Opis aplikacji

Kod źródłowy aplikacji demonstrujący użycie magistrali SPI znajduje się w pliku źródłowym spiaccel.c, który następnie kompilowany jest do postaci wykonywalnego pliku binarnego o nazwie spiaccel. Kod pętli głównej programu przedstawiono na **listingu 4**.

Na początku otwierany jest plik /dev/spidev0.1 reprezentujący magistralę SPI – NPCS1 , do zapisu oraz do odczytu. Następnie za pomocą funkcji dumpstat są wypisywane informacje o aktualnym trybie pracy magistrali. Kolejną czynnością jest inicjalizacja czujnika lis35 za pomocą wywołania funkcji lis35_initialize, po czym program wchodzi do pętli nieskończonej, w której cyklicznie odczytywana jest pozycja x,y,z za pomocą wywołania funkcji lis35_getpos, a następnie wypisywana na wyświetlaczu za pomocą funkcji printf. Kod funkcji dumpstat() wypisującej aktualne ustawienia dotyczące magistrali SPI przedstawiono na **listingu 5**.

Działanie tej funkcji jest nieskomplikowane i sprowadza się do wywołania funkcji systemowej ioctl z argumentami pobierającymi ustawienia (zgodnie z tab. 1), a następnie wypisaniu ich za pomocą funkcji printf na standardowym wyjściu. Za inicjalizację akcelerometru LIS35 odpowiedzialna jest funkcja lis35_initialize (**listing 6**).

Funkcja ta korzysta z funkcji lis35_write_registers() oraz lis35_read_registers(), które za pomocą API dostępowego, wykonuje fizyczny zapis lub odczyt rejestrów układu. Na początku przywracane są podstawowe ustawienia fabryczne rejestrów poprzez ustawienie bitu BOOT w rejestrze CR2 [1]. Następnie w [2] jest włączany akcelerometr oraz pomiary dla osi X, Y, Z poprzez usta-

Listing 4. Pętla główna programu

```
//Main function
int main(void)
{
  //Open spi device
  int spifd = open( "/dev/spidev0.1",O_RDWR );
  error(spifd<0);
  //Dump spi status
  dumpstat("spidev0.1", spifd);
  //Initialize the lis35
  lis35_initialize( spifd );
  for(lis35_pos_t pos;;)
  {
    lis35_getpos( spifd, &pos );
    printf("x: %d y: %d z: %d\n",pos.x, pos.y,pos.z);
    sleep(1);
  }
  //Close the SPI device
  close(spifd);
  return 0;
}
```

Listing 5. Kod funkcji dumpstat() wypisującej aktualne ustawienia SPI

```
static void dumpstat(const char *name, int fd)
{
  __u8    mode, lsb, bits;
  __u32   speed;
  if (ioctl(fd, SPI_IOC_RD_MODE, &mode) < 0) {
    perror("SPI rd_mode");
    return;
  }
  if (ioctl(fd, SPI_IOC_RD_LSB_FIRST, &lsb) < 0) {
    perror("SPI rd_lsb_fist");
    return;
  }
  if (ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits) < 0) {
    perror("SPI bits_per_word");
    return;
  }
  if (ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed) < 0) {
    perror("SPI max_speed_hz");
    return;
  }
  printf("%s: spi mode %d, %d bits %sper word, %d Hz
max\n",
  name, mode, bits, lsb ? "(lsb first) " : "", speed);
}
```

Listing 6. Inicjalizacja akcelerometru LIS35

```
static inline void lis35_reboot_memory(int handle)
{
  const char val = LIS35_REG_CR2_BOOT;
  lis35_write_registers( handle, LIS35_REG_CR2, &val, sizeof(val) );
}

static void lis35_initialize(int handle)
{
  lis35_reboot_memory( handle ); //[1]
  char reg;
  const char lis35cfg = LIS35_REG_CR1_XEN | LIS35_REG_CR1_YEN | LIS35_REG_CR1_ZEN | LIS35_REG_CR1_ACTIVE;
  lis35_write_registers(handle, LIS35_REG_CR1, &lis35cfg, sizeof(lis35cfg) ); //[2]
  lis35_read_registers( handle, LIS35_REG_CR1, &reg, sizeof(reg) ); //[3]
  if(reg != lis35cfg)
  {
    fprintf(stderr,"Error!! probably LIS35 is not connected to the SPI0.1 %02x!=%02x\n",reg,lis35cfg);
    exit(-1);
  }
}
```

```
Listing 6. Funkcja lis35_getpos()
typedef struct lis35_pos_s
{
  signed char x;
  signed char dummy1;
  signed char y;
  signed char dummy2;
  signed char z;
} __attribute__((packed)) lis35_pos_t;

static inline void lis35_getpos(int handle, lis35_pos_t *position)
{
  lis35_read_registers(handle, LIS35_REG_OUTX, position, sizeof(*position) );
}
```

```
Listing 7. Funkcja lis35_write_register odpowiedzialna za odczyt danych
//Write data to the LIS35 hardware device
static void lis35_write_registers(int handle, char addr, const void *val, size_t nregs)
{
  //Create bytes to write
  char buf[nregs+1];
  buf[0] = addr|LIS35_WRITE|LIS35_ADDR_INC;
  memcpy(&buf[1], val, nregs);
  //Write to the SPI dev
  int res = write( handle, buf, sizeof(buf));
  error(res!=(int)sizeof(buf));
}
```

```
Listing 8. Funkcja odczytująca rejestry układu LIS35
//Read lis35 register
static void lis35_read_registers(int handle, char addr, void* regs, size_t nregs)
{
  addr |= LIS35_READ | LIS35_ADDR_INC; //[1]
  struct spi_ioc_transfer xfer[2]; //[2]
  memset(xfer, 0, sizeof xfer);
  xfer[0].tx_buf = (unsigned long)&addr;
  xfer[0].len = sizeof(addr);
  xfer[1].rx_buf = (unsigned long)regs;
  xfer[1].len = nregs;
  int status  = ioctl(handle, SPI_IOC_MESSAGE(2), xfer); //[3]
  error(status<0);
}
```

wienie odpowiednich bitów w rejestrze CR1. W [3] ponownie jest odczytywana zawartość rejestru CR1 w celu stwierdzenia faktycznego zapisu, ponieważ w wypadku SPI nie ma żadnego mechanizmu potwierdzającego fizyczny zapis danych. Gdy wartość zapisana różni się od wartości odczytanej, najprawdopodobniej oznacza to, że układ nie jest dołączony do magistrali, a więc na ekranie zostaje wyświetlony komunikat o błędzie i program zostaje zakończony za pomocą wywołania exit(). Za odczyt pozycji osi X, Y, Z jest odpowiedzialna pokazana na listingu 6 funkcja lis35_getpos().

Działanie tej funkcji jest bardzo proste i sprowadza się do wywołania funkcji lis35_read_registers, która odczytuje rejestry układu LIS35, a następnie wpisuje je do struktury lis35_pos_s. Funkcja lis35_write_register jest odpowiedzialna za odczyt danych z rejestrów układu za pośrednictwem magistrali SPI (**listing 7**).

Jako argumenty przyjmuje kolejno, uchwyt do otwartego pliku reprezentującego magistralę, adres pierwszego rejestru układu LIS35, wskaźnik z zawartością rejestrów które mają być zapisane oraz liczbę rejestrów. Działanie tej funkcji sprowadza się to utworzenia bufora składającego się z adresu pierwszego rejestru oraz ciągu wartości rejestrów, a następnie przesłania danych na magistralę SPI za pomocą wywołania systemowego write(). Ponieważ przy zapisie do rejestrów zapisujemy tylko ciąg bajtów, a dane odczytane są nieistotne, zatem wystarczy zwykłe wywołanie write(). Czytelnicy zapewne zauważyli, że wielkość bufora znaków do wysłania buf, tworzonego jako zmienna lokalna, nie jest stała, a zależy od zmiennej nregs. Nie jest to dozwolone w ANSI-C, a jedynie w rozszerzeniu GNU-99.

Za odczyt rejestrów układu LIS35 odpowiada funkcja lis35_read_registers(), która podobnie jak poprzednio przyjmuje uchwyt do pliku reprezentującego magistralę, adres pierwszego rejestru oraz liczbę rejestrów.

Funkcja ta jest trochę bardziej skomplikowana z uwagi na konieczność odbierania oraz wysyłania danych podczas pojedynczego transferu, bez deaktywacji linii CS, co jest realizowane z wykorzystaniem interfejsu synchronicznego, przedstawionego w rozdziale (2.3) Na początku w, bajcie reprezentującym adres układu LIS35, ustawiane są bity LIS35_READ oraz LIS35_ADDR_INC informujące o tym, że będzie to tryb odczytu, oraz aby z każdym odczytanym bajtem zwiększany był automatycznie wewnętrzny licznik adresu układu. Następnie w [2] tworzona jest tablica struktur spi_ioc_transfer reprezentująca transakcję SPI. Będzie się ona składała z dwóch transferów. Pierwszym transferem będzie zapis wpisujący adres sprzętowy rejestru, natomiast drugim transferem będzie odczyt rejestrów układu. Po zainicjalizowaniu zmienna xfer jest następnie przekazywana jako argument wywołania ioctl(), której wywołanie zrealizuje fizyczny transfer danych magistralą SPI.

Uruchomienie przykładu na BF210

Skompilowany przykład wraz kompletnym systemem Linux, dostarczany jest w postaci obrazu example4.img. Przykład należy nagrać na kartę SD tak jak to zostało opisane w pierwszym odcinku. Po nagraniu karty, włożeniu jej do BF210, oraz uruchomieniu systemu, należy zalogować się do konsoli jako root, a następnie uruchomić przykład wpisując polecenie spiaccel, co spowoduje uruchomienie programu. Jeżeli moduł KA-MOD-MEMS2 został dołączony poprawnie na terminalu (konsoli), powinna być widoczna aktualna pozycja osi czujnika, aktualizowana co 1 s. Zakończenie działania programu możliwe jest przez wciśnięcie kombinacji klawiszy Ctrl+C.

Lucjan Bryndza, EP