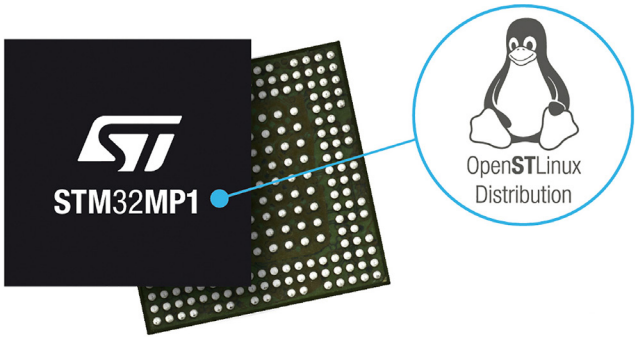
OpenSTLinux

dla procesorów z rodziny STM32MP1 (4)

FTP POPRZEDNIE ODCINKI SĄ DOSTĘPNE NA STRONIE WWW.MEDIA.AVT.PL

Od czasu, kiedy powstała pierwsza część cyklu, firma ST opublikowała nową wersję ekosystemu do procesorów z rodziny STM32MP. Dlatego w tej części omówimy krótko budowanie nowej wersji – dunfell oraz przyjrzymy się uruchamianiu aplikacji na rdzeniu Cortex-M4.



Do tej pory pracowaliśmy na wersji systemu **thud** i ekosystemie 1.2.0, natomiast najnowsza dostępna wersja w chwili pisania tego artykułu to **dunfell** z ekosystemu 2.1.0. Budowanie nowej wersji systemu wygląda analogicznie jak poprzednia, jednak należy pamiętać o aktualizacji wszystkich używanych komponentów. W pierwszej kolejności powinniśmy upewnić się, że mamy najnowszą dostępną wersję narzędzia *repo*, które z kolei wymaga zainstalowanego Pythona w wersji co najmniej 3.6. Najnowsze *repo* możemy pobrać ze strony projektu i umieścić w katalogu */usr/bin*, pamiętając o nadaniu mu uprawnień do wykonywania:

```
wget https://storage.googleapis.com/
git-repo-downloads/repo
sudo mv repo /usr/bin/repo
sudo chmod a+x /usr/bin/repo
```

Teraz możemy powtórzyć omawiane wcześniej kroki kompilacji systemu OpenSTLinux, tym razem dla wersji dunfell:

```
cd <working directory path>/Distribution-Package
mkdir openstlinux-5.4-dunfell-mp1-20-11-12
cd openstlinux-5.4-dunfell-mp1-20-11-12
repo init -u https://github.com/
STMicroelectronics/oe-manifest.git -b refs/tags/
openstlinux-5.4-dunfell-mp1-20-11-12
repo sync
cd layers/meta-st
git clone -b dunfell https://github.com/SoMLabs/
openst-meta-somlabs.git meta-somlabs
cd ../../
```

Szczegóły dotyczące nowego ekosystemu można znaleźć na stronie Wiki ST: http://bit.ly/3kzOBGh. Główne zmiany w stosunku do wersji **thud** dotyczą aktualizacji jądra oraz *u-boota*. **Dunfell** oferuje je odpowiednio w wersjach 5.4.56 i 2020.01. Modyfikacjom uległa również warstwa meta-somlabs. W wersji dunfell jądro oraz *u-boot* pobierane są z repozytoriów na stronie http://bit.ly/3dLunHX. Dostępne tam źródła zawierają już niezbędne łaty dla modułów VisionSOM, dzięki czemu nie są one już potrzebne w plikach warstwy. Repozytoria te są definiowane w plikach *bbappend* jądra i *u-boota*: *recipes-kernel/linux/linux-stm32mp_5.4.bbappend* oraz *recipes-bsp/u-boot/u-boot-stm32mp_2020.01.bbappend*. Oprócz tego, w warstwie meta-somlabs są dostępne używane podczas kompilacji pliki konfiguracyjne jądra i *u-boota*, dzięki czemu można dodać wymagane opcje bez potrzeby zmieniania źródeł, co miało miejsce w przykładach z poprzedniego odcinka.

Na koniec warto wspomnieć, że zmieniły się też nazwy dostępnych urządzeń, ponieważ wprowadzona została obsługa pamięci eMMC. Możemy więc zbudować system dla modułu z pamięcią eMMC lub SD i jednym z interfejsów wyświetlacza: DSI lub RGB:

```
stm32mp157a-visionsom-rgb-emmc-mx
stm32mp157a-visionsom-rgb-sd-mx
stm32mp157a-visionsom-dsi-emmc-mx
stm32mp157a-visionsom-dsi-sd-mx
```

Budowanie nowej wersji systemu OpenSTLinux

W przykładzie użyjemy modułu z pamięcią SD i wyświetlaczem RGB. Wywołujemy więc następujące polecenia, które rozpoczynają budowanie obrazu systemu:

```
DISTRO=openstlinux-weston MACHINE=stm32mp157a-
visionsom-rgb-sd-mx source layers/meta-st/scripts/
envsetup.sh
bitbake st-image-weston
```

Podobnie jak poprzednio, musimy użyć skryptu tworzącego obraz dla karty SD, jednak tym razem musimy użyć konfiguracji trusted, która jest od tej pory jedyną dostępną:

```
cd tmp-glibc/deploy/images/
stm32mp157a-visionsom-rgb-sd-mx
./scripts/create_sdcard_from_flashlayout.sh
./flashlayout_st-image-weston/trusted/FlashLayout_
sdcard_stm32mp157a-visionsom-rgb-sd-mx-trusted.tsv
```

Plik obrazu możemy skopiować na kartę SD poleceniem dd:

```
sudo dd if=FlashLayout_sdcard_stm32mp157a-
visionsom-rgb-sd-mx-trusted.raw of=/dev/sdX bs=1M
```

Obsługa Cortex-M4

W pierwszej części cyklu przygotowaliśmy program w języku C, który był następnie uruchamiany w systemie Linux działającym na rdzeniu aplikacyjnym Cortex-A7. Tym razem przekonamy się, w jaki sposób uruchomić rdzeń Cortex-M4, a następnie załadować i uruchomić przygotowaną dla niego aplikację.

Zmiany opisane w tym rozdziale można wprowadzić, korzystając z opisanego projektu *openst-cube-mx* dla STM32CubeMX, jednak tym razem musimy pamiętać o pobraniu wersji dunfell z repozytorium:

```
git clone https://github.com/SoMLabs/openst-cube-mx.
git -b dunfell
```

Listing 1. Sekcja konfigurująca rdzeń Cortex-M4 w device tree

```
&m4_rproc{
    status = "okay";
    /* USER CODE BEGIN m4_rproc */
    memory-region = <&retram>, <&mcuram>, <&mcuram2>,
        <&vdev0vring0>, <&vdev0vring1>, <&vdev0buffer>;
    mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
    mbox-names = "vq0", "vq1", "shutdown";
    interrupt-parent = <&exti>;
    interrupts = <68 1>;
    wakeup-source;
    /* USER CODE END m4_rproc */
    m4_system_resources{
    status = "okay";
    /* USER CODE BEGIN m4_system_resources */
    /* USER CODE END m4_system_resources */
    };
};
```

Listing 2. Sekcje pamięci używane przez rdzeń Cortex-M4

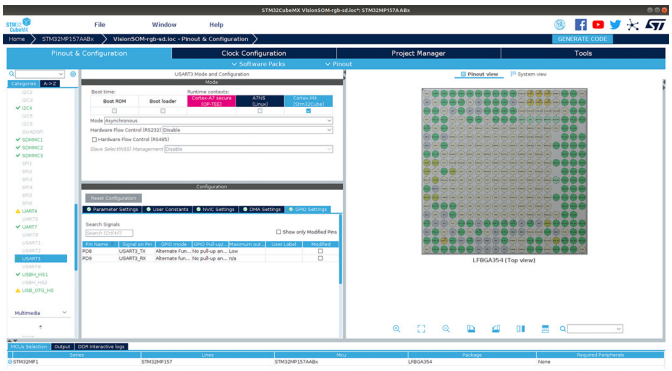
```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    /* USER CODE BEGIN reserved-memory */
    fb_reserved: fb@fd000000 {
        reg = <0xfd000000 0x8000000>;
        no-map;
    };
    mcuram2: mcuram2@10000000 {
        compatible = "shared-dma-pool";
        reg = <0x10000000 0x40000>;
        no-map;
    };
    vdev0vring0: vdev0vring0@10040000 {
        compatible = "shared-dma-pool";
        reg = <0x10040000 0x1000>;
        no-map;
    };
    vdev0vring1: vdev0vring1@10041000 {
        compatible = "shared-dma-pool";
        reg = <0x10041000 0x1000>;
        no-map;
    };
    vdev0buffer: vdev0buffer@10042000 {
        compatible = "shared-dma-pool";
        reg = <0x10042000 0x4000>;
        no-map;
    };
    mcuram: mcuram@30000000 {
        compatible = "shared-dma-pool";
        reg = <0x30000000 0x40000>;
        no-map;
    };
    retram: retram@38000000 {
        compatible = "shared-dma-pool";
        reg = <0x38000000 0x10000>;
        no-map;
    };
    /* USER CODE END reserved-memory */
};
```

Listing 3. Wpis konfigurujący moduł IPCC

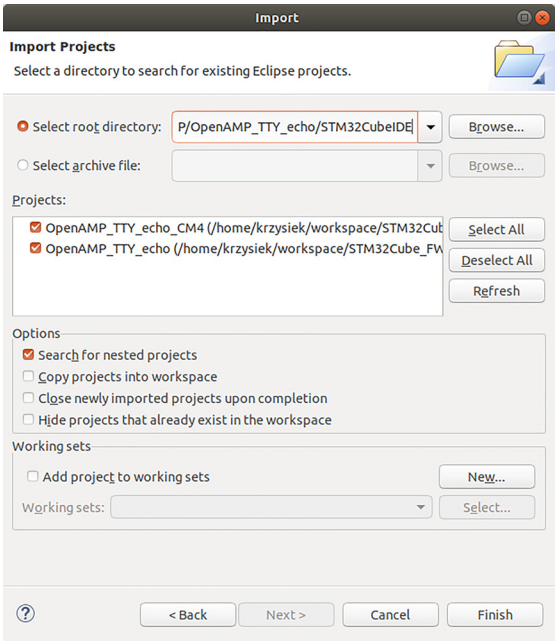
```
&ipcc {
    status = "okay";
};
```

Konfiguracja Cortex-M4 sprowadza się do dodania odpowiednich wpisów w device tree. Musimy uzupełnić sekcję m4_rproc z pliku *CA7/DeviceTree/VisionSOM-rgb-sd/kernel/stm32mp157a-visionsom-rgb-sd-mx.dts* projektu openst-cube-mx, w której definiujemy m.in. obszary pamięci oraz tzw. skrzynki pocztowe, dzięki którym aplikacja na Cortex-M4 będzie komunikować się z systemem Linux. Wpis został pokazany na **listingu 1**.

Kolejnym krokiem jest dodanie dopisanych przed chwilą obszarów pamięci do sekcji reserved memory w tym samym pliku, tak jak zostało to pokazane na **listingu 2**.



Rysunek 1. Konfiguracja USART3 dla rdzenia Cortex-M4



Rysunek 3. Importowanie przykładowego projektu z pakietu STM32CubeMP1

Na koniec uruchamiamy IPCC do komunikacji pomiędzy rdzeniami, dodając do device tree sekcję &ipcc na końcu pliku pomiędzy znacznikami *USER CODE*, tak jak na **listingu 3**.

Skonfigurowany w ten sposób rdzeń Cortex-M4 będzie w stanie wykonać skompilowaną dla niego aplikację, jednak aby móc skorzystać z jakiegokolwiek peryferium, musimy uruchomić je tak jak poprzednim razem, jednak tym razem przydzielając je do rdzenia Cortex-M4. W przykładzie użyjemy portu USART3, który musimy skonfigurować, używając narzędzia STM32CubeMX, tak jak zostało to pokazane na **rysunku 1**.

Tak przygotowane pliki device tree możemy skopiować do odpowiedniego miejsca w katalogu warstwy meta-somlabs (*meta-somlabs/mx/visionsom-rgb-sd-mx*), podmieniając znajdującą się tam konfigurację domyślną. Po przebudowaniu obrazu i uruchomieniu systemu możemy zweryfikować działanie rdzenia Cortex-M4, używając do tego przygotowanej wcześniej aplikacji demo. W tym celu musimy pobrać spakowany plik *elf* i poinformować system, aby użył go jako programu dla rdzenia Cortex-M4:

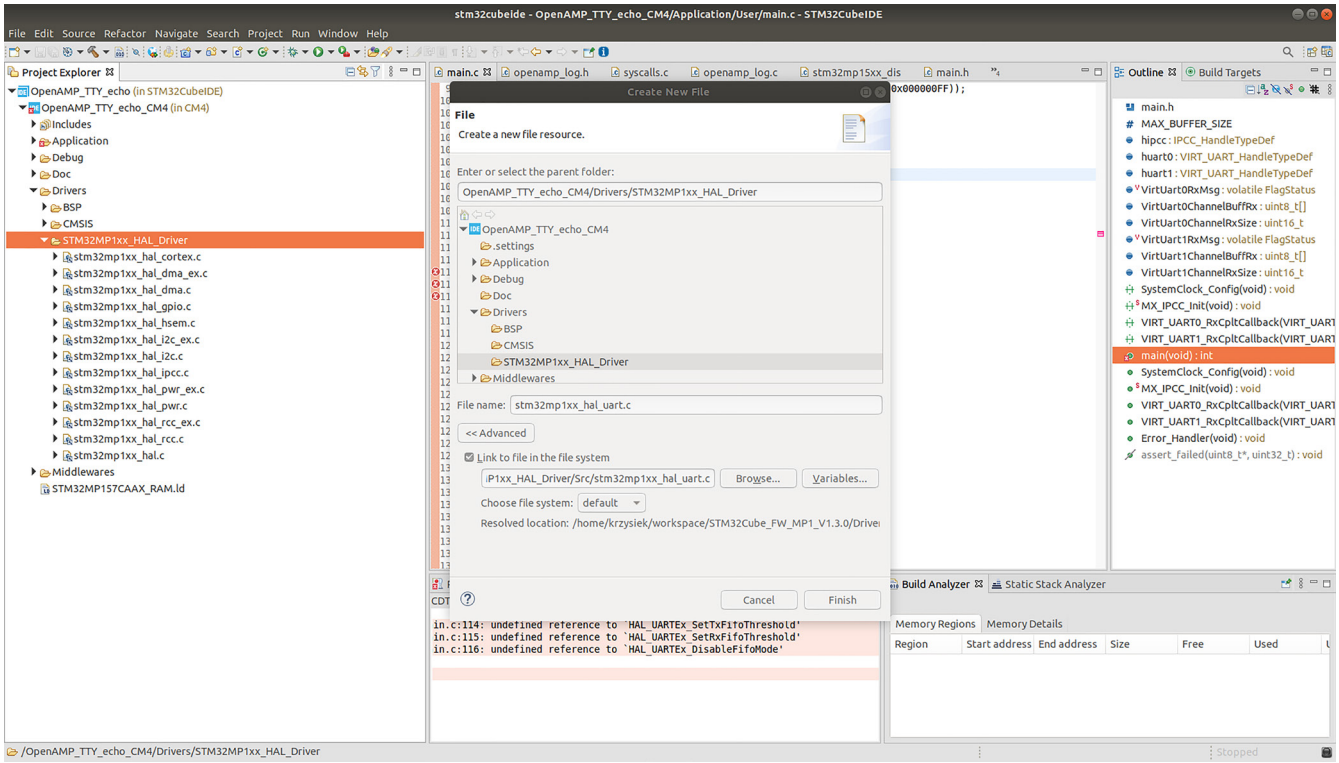
```
wget https://wiki.somlabs.com/images/9/94/OpenAMP_
TTY_echo.zip
unzip OpenAMP_TTY_echo.zip -d /lib/firmware
cd /sys/class/remoteproc/remoteproc0
echo "OpenAMP_TTY_echo.elf" > firmware
echo start > state
```

System Linux domyślnie szuka plików z programem dla Cortex-M4 w katalogu */lib/firmware*, dlatego to właśnie tam należy rozpakować plik *OpenAMP_TTY_echo.elf*. Ostatnie z podanych powyżej poleceń uruchamia rdzeń, który po załadowaniu systemu jest domyślnie wyłączony.

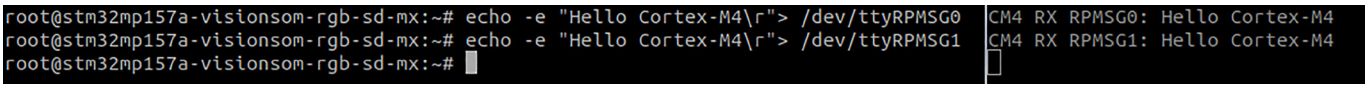
Po uruchomieniu przykładu, w terminalu zobaczymy kilka komunikatów świadczących o włączeniu rdzenia Cortex-M4 i utworzeniu dwóch kanałów komunikacji RPMSG (**rysunek 2**). Są to specjalnie

```
[   51.697447] remoteproc remoteproc0: powering up m4
[   51.794724] remoteproc remoteproc0: Booting fw image OpenAMP_TTY_echo.elf, size 2001996
[   51.820287] rproc-srm-core mlahb:m4@10000000:m4_system_resources: bound mlahb:m4@10000000:m4_system_resources:serial@4000f000 (ops 0xc0bf84d4)
[   51.855694] rproc-srm-core mlahb:m4@10000000:m4_system_resources: bound mlahb:m4@10000000:m4_system_resources:dma@48001000 (ops 0xc0bf84d4)
[   51.886007]  remoteproc0#vdev0buffer: assigned reserved memory node vdev0buffer@10042000
[   51.895294] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x0
[   51.901540] virtio_rpmsg_bus virtio0: rpmsg host is online
[   51.907625] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 -> 0x0 : ttyRPMSG0
[   51.915185] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel addr 0x1
[   51.926968]  remoteproc0#vdev0buffer: registered virtio0 (type 7)
[   51.931628] remoteproc remoteproc0: remote processor m4 is now up
```

Rysunek 2. Komunikaty po włączeniu rdzenia Cortex-M4 i uruchomieniu przykładowej aplikacji



Rysunek 4. Dodanie źródeł sterownika UART z biblioteki STM32CubeMP1



Rysunek 5. Działanie przykładowej aplikacji – po lewej stronie terminal systemu Linux, po prawej wyjście portu szeregowego USART3

zdefiniowane obszary pamięci, które służą do wymiany danych pomiędzy rdzeniami aplikacyjnymi pod kontrolą systemu Linux a rdzeniem czasu rzeczywistego. Z punktu widzenia użytkownika systemu Linux można traktować je jako zwykłe porty szeregowe. Przekonamy się o tym, używając poleceń *echo* i *cat*, którymi możemy odpowiednio wysyłać i odbierać dane z każdego z tych portów:

```
cat /dev/ttyRPMSG0 &
echo "Hello Cortex-M4" >/dev/
ttyRPMSG0
```

Dodatkowo, komunikacja pomiędzy rdzeniami w przykładzie jest logowana w systemie i możemy ją odczytać z pliku */sys/kernel/debug/remoteproc/remoteproc0/trace0*.

Kompilacja aplikacji dla Cortex-M4

W poprzednim rozdziale uruchomiliśmy gotową aplikację, przygotowaną dla rdzenia Cortex-M4. Teraz zobaczymy, w jaki sposób skompilować jej zmodyfikowaną wersję, która dodatkowo będzie używała skonfigurowanego przez nasz wcześniej portu USART3.

Zaczynamy od pobrania pakietu STM32CubeMP1 w wersji 1.3.0 (https://bit.ly/3uA956m) oraz środowiska STM32CubeIDE (http://bit.ly/2XsPWlH) ze strony producenta. W pakiecie STM32CubeMP1 znajdują się gotowe przykłady, które możemy skompilować i uruchomić na rdzeniu Cortex-M4

Listing 4. Funkcja MX_USART3_UART_Init wygenerowana przez STM32CubeMX

```
void MX_USART3_UART_Init(void) {
    huart3.Instance = USART3;
    huart3.Init.BaudRate = 115200;
    huart3.Init.WordLength = UART_WORDLENGTH_8B;
    huart3.Init.StopBits = UART_STOPBITS_1;
    huart3.Init.Parity = UART_PARITY_NONE;
    huart3.Init.Mode = UART_MODE_TX_RX;
    huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart3.Init.OverSampling = UART_OVERSAMPLING_16;
    huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart3.Init.ClockPrescaler = UART_PRESCALER_DIV1;
    huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart3) != HAL_OK) {
        Error_Handler();
    }
    if (HAL_UARTEx_SetTxFifoThreshold(&huart3, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK) {
        Error_Handler();
    }
    if (HAL_UARTEx_SetRxFifoThreshold(&huart3, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK) {
        Error_Handler();
    }
    if (HAL_UARTEx_DisableFifoMode(&huart3) != HAL_OK) {
        Error_Handler();
    }
}
```

Listing 5. Funkcja HAL_UART_MspInit wygenerowana przez STM32CubeMX

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) {
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};
    if(huart->Instance==USART3) {
        if(IS_ENGINEERING_BOOT_MODE()) {
            PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_UART35;
            PeriphClkInit.Uart35ClockSelection = RCC_UART35CLKSOURCE_PCLK1;
            if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK) {
                Error_Handler();
            }
        }
        __HAL_RCC_USART3_CLK_ENABLE();
        __HAL_RCC_GPIOD_CLK_ENABLE();
        GPIO_InitStruct.Pin = GPIO_PIN_8;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
        GPIO_InitStruct.Pin = GPIO_PIN_9;
        GPIO_InitStruct.Mode = GPIO_MODE_AF;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
    }
}
```

procesora STM32MP1. My użyjemy projektu o nazwie OpenAMP_TTY_echo, który służył nam do testów w poprzednim punkcie.

Po zainstalowaniu środowiska STM32CubeIDE importujemy projekt za pomocą opcji File → Import, a następnie wybierając w oknie dialogowym General → Existing Projects into Workspace. W kolejnym oknie musimy wskazać katalog z projektem (*STM32Cube_FW_MP1_V1.3.0/Projects/STM32MP157C-DK2/Applications/OpenAMP/OpenAMP_TTY_echo/STM32CubeIDE*) tak jak na **rysunku 3**, a następnie przyciskiem Finish kończymy cały proces.

W kodzie aplikacji są inicjalizowane dwa kanały RPMSG, a następnie rozpoczyna się pętla *while*, w której otrzymane wiadomości są wysyłane z powrotem na odpowiedni port. Zmodyfikujemy kod, aby dodatkowo wysyłał dane na port USART3. Na początku będziemy potrzebować funkcji konfigurujących port szeregowy oraz przypisane do niego wyprowadzenia. Odpowiednie funkcje możemy napisać własnoręcznie lub skorzystać z kodu wygenerowanego przez narzędzie STM32CubeMX, które oprócz plików device tree, tworzy także kod dla Cortex-M4, który znajdziemy w katalogu CM4 projektu *openst-cube-mx/VisionSOM-rgb-sd*. Skopiujmy więc funkcje *MX_USART3_UART_Init* z pliku *CM4/Src/main.c* oraz *HAL_UART_MspInit* z pliku *CM4/Src/stm32mp1xx_hal_msp.c* do pliku *main.c* w otwartym projekcie w katalogu Application/User. Obie funkcje zostały pokazane na **listingach 4 i 5**. Na początku pliku *main.c* musimy także umieścić deklaracje dodanych funkcji i zmienną *huart3*, co zostało zawarte na **listingu 6**. Możemy teraz zmodyfikować funkcję *main()*, dodając wywołanie funkcji konfigurującej port szeregowy oraz przekierowując na niego otrzymane pakiety danych za pomocą funkcji bibliotecznej *HAL_UART_Transmit*. Wspomniane zmiany są widoczne na **listingu 7**.

Do poprawnej kompilacji powinniśmy dodać do projektu nagłówki z definicjami i deklaracjami dla portu UART przez odkomentowanie linii *#define HAL_UART_MODULE_ENABLED* w pliku *Inc/stm32mp1xx_hal_conf.h*. Musimy także uwzględnić w projekcie niezbędne pliki sterowników z biblioteki STM32Cube – klikamy prawym przyciskiem myszy na katalog *STM32MP1xx_HAL_Driver* w drzewie projektu po lewej stronie i wybieramy opcję New → File. W oknie dialogowym, które się pojawiło, wybieramy opcję Advanced i podajemy ścieżkę do pliku sterownika UART: *${PARENT-7-PROJECT_LOC}/Drivers/STM32MP1xx_HAL_Driver/Src/stm32mp1xx_hal_uart.c*, tak jak na **rysunku 4**. W podobny sposób musimy dodać także plik *${PARENT-7-PROJECT_LOC}/Drivers/STM32MP1xx_HAL_Driver/Src/stm32mp1xx_hal_uart_ex.c*.

Teraz możemy przebudować projekt i skopiować plik *OpenAMP_TTY_echo_CM4.elf* na płytkę VisionSOM. Plik *elf* znajduje się w katalogu z przykładem: *OpenAMP_TTY_echo/STM32CubeIDE/CM4/Debug*. Po uruchomieniu nowej wersji programu na rdzeniu Cortex-M4, wszystkie wiadomości odebrane z kanałów RPMSG0 i RPMSG1 będą również wysyłane na port USART3, którego wyprowadzenia znajdziemy na pinach 8 (TX) i 10 (RX) złącza Raspberry Pi. Efekt działania nowej wersji programu jest widoczny na **rysunku 5**.

Krzysztof Chojnowski

Listing 6. Deklaracje nowych funkcji i zmienna huart3

```
UART_HandleTypeDef huart3;
static void MX_USART3_UART_Init(void);
void HAL_UART_MspInit(UART_HandleTypeDef* huart);
```

Listing 7. Konfiguracja portu szeregowego i główna pętla aplikacji

```
MX_USART3_UART_Init();
while (1) {
    OPENAMP_check_for_message();
    if (VirtUart0RxMsg) {
        HAL_UART_Transmit(&huart3, (uint8_t*)"CM4 RX RPMSG0: ", 8, 1000);
        HAL_UART_Transmit(&huart3, VirtUart0ChannelBuffRx, VirtUart0ChannelRxSize, 1000);
        VirtUart0RxMsg = RESET;
        VIRT_UART_Transmit(&huart0, VirtUart0ChannelBuffRx, VirtUart0ChannelRxSize);
    }
    if (VirtUart1RxMsg) {
        HAL_UART_Transmit(&huart3, (uint8_t*)"CM4 RX RPMSG1: ", 8, 1000);
        HAL_UART_Transmit(&huart3, VirtUart1ChannelBuffRx, VirtUart1ChannelRxSize, 1000);
        VirtUart1RxMsg = RESET;
        VIRT_UART_Transmit(&huart1, VirtUart1ChannelBuffRx, VirtUart1ChannelRxSize);
    }
}
```