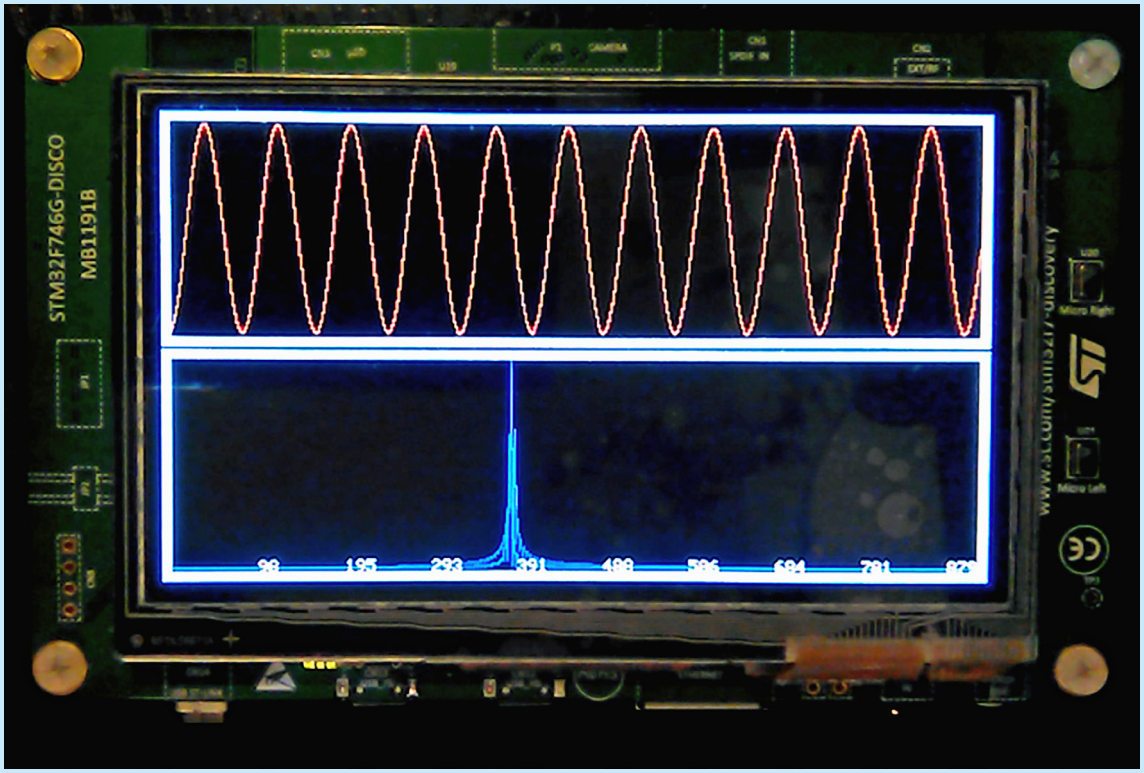

Podstawy programowania STM32F746G-DISCO (2)

Jak zbudować oscyloskop z FFT z użyciem STM32F746G-DISCO

W drugiej części cyklu zostaną przedstawione podstawy działania systemu operacyjnego FreeRTOS jako głównego elementu struktury aplikacji. W artykule będą pokazane jedynie funkcje systemu wykorzystane w projekcie, co stanowi niewielką część jego możliwości. Informacje na temat wszystkich możliwości systemu FreeRTOS można znaleźć na oficjalnej stronie projektu *www.freertos.org*.

Zastosowany w projekcie system operacyjny zawsze decyduje o strukturze aplikacji i sposobie implementacji poszczególnych funkcji. W mniejszych projektach zastosowanie systemu operacyjnego nie niesie ze sobą wielu korzyści, dlatego najczęściej spotykaną strukturą kodu jest inicjalizacja peryferiów w funkcji main i reakcja na zdarzenia za pośrednictwem funkcji obsługi przerwań. Struktura ta sprawdza się w większości prostych systemów sterowania, jednak okazuje się niewystarczająca w przypadku większych projektów. Wraz ze wzrostem ilości kodu następuje potrzeba jego organizacji, tak aby ułatwić jego rozwój i wprowadzanie zmian oraz umożliwić jednoczesną pracę nad projektem większemu zespołowi programistów. Systemy operacyjne spełniają te warunki, ale nie tylko. Pozwalają na lepszą separację warstwy aplikacji od wykorzystywanego sprzętu i BSP (*Board Support Package*), a także kontrolę zależności czasowych pomiędzy poszczególnymi częściami programu, co bywa szczególnie istotne w wielu systemach wbudowanych.

FreeRTOS jest systemem darmowym, dystrybuowanym na licencji OpenSource. Oznacza to, że można go wykorzystać także w projektach komercyjnych na warunkach przedstawionych na stronie: *http://goo.gl/pW2WIa*. Jego zaletą jest niewątpliwie małe zużycie pamięci. Zgodnie z danymi przedstawionymi na oficjalnej stronie planista (*scheduler*), czyli podstawa działania systemu operacyjnego, potrzebuje jedynie 236 bajtów

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **86118**, pass: **8655327a**

Tabela 1. Wymagana pamięć RAM	
Komponent	**Liczba bajtów**
Planista	236
Każda utworzona kolejka	76 + pamięć zarezerwowana na przesyłane dane
Każde utworzone zadanie	64 + rozmiar stosu

RAM-u. Całkowite zużycie pamięci zależy od liczby utworzonych komponentów systemu, takich jak zadania, lub kolejki (**tabela 1**). Ilość potrzebnej pamięci Flash jest zależna od kompilatora i konfiguracji systemu i wynosi od 5 do 10 kB.

Swoje niskie wymagania odnośnie do pamięci FreeRTOS zawdzięcza brakowi jakichkolwiek sterowników peryferiów, czy BSP. Kod źródłowy systemu zawiera m. in. mechanizmy tworzenia i zarządzania zadaniami, liczniki programowe oraz implementację metod komunikacji i synchronizacji zadań. W efekcie kod FreeRTOSa jest prosty do zrozumienia oraz połączenia z istniejącymi bibliotekami, w tym BSP.

Konfiguracja systemu

System FreeRTOS jest konfigurowany na etapie kompilacji za pośrednictwem definicji znajdujących się w pliku FreeRTOSConfig.h. Plik konfiguracyjny w projekcie włącza tylko te opcje systemu które są potrzebne do poprawnego działania, aby zmniejszyć ilość potrzebnej pamięci. Część z nich przyjmuje wartości 0 lub 1 (wyłączone, lub włączone), podczas gdy inne wymagają wartości liczbowych. Pełną listę opcji oraz wartości domyślnych niezdefiniowanych parametrów można znaleźć w pliku FreeRTOS.h, w źródłach systemu. Warto także zwrócić uwagę, że część opcji nie ma wartości domyślnych – muszą być one zdefiniowane jawnie, w przeciwnym razie zostanie zgłoszony błąd kompilacji. **Tabela 2** zawiera opis niektórych z używanych opcji.

Planista

Sercem systemu operacyjnego jest planista, umożliwiający istnienie wielu zadań w tym samym czasie. Jest on odpowiedzialny za wywłaszczenie obecnie wykonywanego zadania oraz wybór i uruchomienie kolejnego, z kolejki zadań gotowych. Zmiana wykonywanego zadania wiąże się z tzw. przełączaniem kontekstu, czyli m. in. zapisem stanu rejestrów procesora wywłaszczanego zadania do pamięci, oraz odtworzenia z pamięci stanu jego następcy. FreeRTOS implementuje dwa spośród znanych algorytmów kolejkowania. Pierwszy z nich opiera się na wykonywaniu zadań o wyższym priorytecie w pierwszej kolejności. Algorytm ten gwarantuje że najważniejsze zadania w systemie zostaną wykonane w odpowiednim czasie, kosztem tych mniej istotnych. Należy przy tym zwrócić uwagę na fakt, że przy dużym obciążeniu procesora niektóre z nich mogą nie być w ogóle wykonane. Drugim algorytmem kolejkowania jest algorytm karuzelowy (Round Robin), który jest stosowany do zadań o tym samym priorytecie. W tym przypadku planista przydziela zasoby jednemu z zadań, po czym wywłaszcza je po upływie określonego kwantu czasu, przenosząc je na koniec kolejki zadań gotowych w obrębie tego samego priorytetu.

W przykładzie jedyną operacją przeprowadzaną na planiście systemu jest jego uruchomienie za pomocą funkcji *vTaskStartScheduler();*. Funkcja ta (wywoływana najczęściej na końcu funkcji main) kończy się wyłącznie w przypadku błędu braku pamięci.

Tworzenie zadań

W systemie FreeRTOS aplikacja składa się z komunikujących się ze sobą zadań oraz przerwań. Każde z zadań należy najpierw utworzyć – służy do tego funkcja xTaskCreate(). W projekcie są wykorzystywane trzy zadania:

```
xTaskCreate((TaskFunction_t)GUI_
Task, „GUI_Task”, 1024, NULL,
1, &appGlobals.guiTaskId);
xTaskCreate((TaskFunction_t)Signal_
Task, „Signal_Task”, 1024, NULL,
1, &appGlobals.signalTaskId);
xTaskCreate((TaskFunction_t)FFT_
Task, „FFT_Task”, 1024, NULL,
1, &appGlobals.fftTaskId);
```

Argumenty przekazywane do funkcji xTaskCreate() to odpowiednio:

- wskaźnik do funkcji realizującej zadanie,
- nazwa zadania,
- rozmiar stosu w bajtach,

Tabela 2. Wybrane parametry konfiguracji systemu	
Parametr	**Opis**
configUSE_PREEMPTION	Włączenie wywłaszczania zadań (algorytm karuzelowy)
configUSE_TICK_HOOK	Dodanie funkcji, która będzie wywoływana przy każdym przerwaniu zegara systemowego (w projekcie jest używana do aktualizacji licznika wewnętrznego BSP)
configCPU_CLOCK_HZ	Częstotliwość taktowania rdzenia procesora
configTICK_RATE_HZ	Częstotliwość zegara systemu operacyjnego
configMAX_PRIORITIES	Maksymalna liczba priorytetów
configMINIMAL_STACK_SIZE	Minimalny rozmiar stosu zadania w bajtach
configTOTAL_HEAP_SIZE	Całkowita pamięć RAM przydzielona systemowi
configUSE_MUTEXES	Umożliwia użycie muteksów do synchronizacji zadań (wymagane przez bibliotekę graficzną)
configUSE_TIMERS	Włączenie liczników programowych
configTIMER_TASK_PRIORITY	Priorytet przerwania licznika programowego (funkcja obsługi licznika jest traktowana przez planistę jak zwykłe zadanie)
configTIMER_QUEUE_LENGTH	Maksymalna liczba przerwań liczników oczekujących na obsługę
configTIMER_TASK_STACK_DEPTH	Rozmiar stosu zadania obsługi przerwania licznika
INCLUDE_vTaskDelay	Dodanie funkcji umożliwiającej implementację opóźnienia wewnątrz zadania

- wskaźnik na strukturę danych zawierającą opcjonalne argumenty zadania – może być to dowolny typ danych prosty lub złożony,
- priorytet,
- wskaźnik do zmiennej typu TaskHandle_t, do której zostanie zapisany unikalny identyfikator zadania, lub 0 w przypadku błędu.

Wszystkie zadania są tworzone w funkcji main po inicjalizacji mikrokontrolera i przed uruchomieniem planisty. Uruchomienie tego ostatniego jest jednoznaczne ze startem wszystkich utworzonych zadań.

Komentarza wymagają funkcje realizujące zadania i przekazywane przez wskaźnik do funkcji xTaskCreate. Funkcje te nie powinny się nigdy kończyć inaczej niż przez wywołanie funkcji vTaskDelete() na ich końcu. Jest to spowodowane tym, ze system operacyjny musi dowiedzieć się o ukończeniu zadania i zwolnić zajmowane przez niego zasoby. W przykładzie funkcje zadań nie kończą się nigdy – po fazie inicjalizacji, każde z nich wchodzi w nieskończoną pętlę zawierającą kod aplikacji. Zadania zostały krótko opisane poniżej:

- **GUI_Task** w fazie inicjalizacji (przed główną pętlą zadania) przygotowuje komponenty interfejsu graficznego biblioteki StemWin: wykresy oraz obiekty danych oraz informuje zadanie Signal_Task o gotowości. Jest to podyktowane względami bezpieczeństwa: komponenty graficzne nie powinny być modyfikowane przez ich utworzeniem. W pętli głównej wywoływana jest funkcja GUI_Delay() realizująca opóźnienie podczas którego odrysowywany jest interfejs graficzny oraz sprawdzana jest kolejka danych zawierająca gesty wykryte na panelu dotykowym. Jeżeli zawiera ona dane powiadamiane jest jedno z zadań: Signal_Task, lub FFT_Task o konieczności przeskalowania lub przesunięcia wykresu.
- **Signal_Task** zadanie to rozpoczyna się oczekiwaniem na powiadomienie o gotowości interfejsu graficznego. Po jego otrzymaniu uruchamiany jest pomiar sygnału, a w pętli głównej aktualizowany jest wykres. Dodatkowo zadanie odpowiada za skalowanie i przesunięcie wykresu, odpowiednio do otrzymanych powiadomień o wykrytych gestach.
- **FFT_Task** inicjalizacja sprowadza się do przygotowania zmiennych potrzebnych później w obliczaniu FFT, natomiast pętla główna jest bardzo podobna do pętli zadania Signal_Task. Jedyną różnicą są dodatkowe operacje matematyczne służące do obliczenia widma amplitudowego.

Prototypy funkcji realizujących powyższe zadania wyglądają następująco:

```
static void GUI_Task(void
const *argument);
static void Signal_Task(void
const *argument);
static void FFT_Task(void
const *argument);
```

Przyjmują one pojedynczy argument, poprzez który można przekazać parametry do wykonywanego zadania podczas jego tworzenia. W przykładzie argumenty wszystkich trzech funkcji mają wartość NULL.

Liczniki programowe

Liczniki programowe (*Software Timers*) są realizowane w całości przez system operacyjny i nie wykorzystują żadnych dodatkowych modułów sprzętowych. Ponadto funkcje wywoływane cyklicznie po upływie zadanego czasu traktowane są przez planistę jak zwykłe zadania o priorytecie ustawianym w parametrze *configTIMER_TASK_PRIORITY*. Przez to ich wywołanie może być opóźnione przez aktualnie wykonywane zadania o wyższych priorytetach.

W projekcie używany jest wyłącznie jeden licznik programowy służący do odczytu danych z kontrolera panelu dotykowego i wpisywania zdetekowanych gestów do odpowiedniej kolejki danych. Jest on tworzony razem z zadaniami w funkcji main za pomocą wywołania

```
appGlobals.touchPanelTimer = xTimerCreate
(„Timer”, pdMS_TO_TICKS(100), pdTRUE,
&appGlobals.touchPanelTimerId,
TouchPanel_TimerCallback);
```

Funkcja xTimerCreate() przyjmuje następujące argumenty:

- nazwa licznika,
- okres licznika wyrażony w wielokrotności okresu zegara systemu (ustawianego parametrem *configTICK_RATE_HZ*),
- flaga oznaczająca, czy funkcja licznika ma być wywoływana okresowo (*pdTRUE*), czy jednokrotnie (*pdFALSE*),
- wskaźnik do zmiennej, która może być użyta do przechowywania dowolnej wartości pomiędzy kolejnymi wywołaniami funkcji obsługi licznika,
- wskaźnik do funkcji obsługi licznika.

Wartością zwracaną jest globalny identyfikator licznika, za którego pomocą można się później do niego odwołać z dowolnego miejsca w aplikacji. W przeciwieństwie do zadań, liczniki nie startują automatycznie po uruchomieniu planisty, lecz są wyzwalane za pomocą funkcji *xTimerStart(appGlobals.touchPanelTimer, 0);*. Funkcja ta przyjmuje globalny identyfikator licznika oraz czas, przez który wywołujące ją zadanie ma oczekiwać na wpisanie polecenia do kolejki komend licznika. W przykładzie licznik jest uruchamiany w zadaniu *GUI_Task*, od razu po inicjalizacji panelu dotykowego.

Ostatnim elementem użytego licznika jest jego funkcja obsługi. Jej prototyp wygląda następująco: *static void TouchPanelTimerCallback(TimerHandle_t pxTimer);*. Argumentem funkcji jest globalny identyfikator licznika. W przeciwieństwie do funkcji zadań, funkcje obsługi liczników muszą się zakończyć w zwykły sposób i nie mogą korzystać z operacji blokujących, jak np. *vTaskDelay()*.

Komunikacja między zadaniami

Działające w systemie procesy muszą mieć możliwość wzajemnego powiadamiania i przesyłania danych. FreeRTOS udostępnia w tym celu kilka łatwych do zastosowania mechanizmów. Poniżej omówione zostały dwa z nich – użyte w projekcie: powiadamianie i kolejki danych.

Powiadamianie (Task Notifications). Każde z zadań w systemie FreeRTOS ma własną 32-bitową wartość służącą do powiadamiania go z poziomu innych zadań lub przerwań. Może ona być użyta zarówno do prostego obudzenia zadania oczekującego na jakieś zdarzenie (jak ma to miejsce na początku wykonania Signal_Task) – wówczas mechanizm ten imituje znane z większości systemów operacyjnych semafory, ale może służyć także

do przesyłania krótkich, 32-bitowych komunikatów. Mechanizm ten jest najszybszą oraz wymagającą najmniej pamięci metodą komunikacji, a przy okazji niezwykle prostą w użyciu, gdyż nie wymaga żadnych dodatkowych struktur danych. Jego wadą jest ograniczenie wynikające z maksymalnego rozmiaru przesyłanej wiadomości do 32 bitów.

W celu wysłania do zadania notyfikacji bez wartości wystarczy wywołać funkcję (tak jak w zadaniu GUI_Task) *xTaskNotifyGive(appGlobals.signalTaskId);*. Jako argument podawany jest globalny identyfikator zadania-adresata przydzielony podczas jego tworzenia. W rzeczywistości wartość notyfikacji jest inkrementowana po każdym wywołaniu tej funkcji, dzięki czemu można uzyskać taki sam efekt jak przy użyciu semaforów.

Zadanie, które oczekuje na powiadomienie (w tym wypadku jest to Signal_Task) wywołuje funkcję *ulTaskNotifyTake(pdTRUE, portMAX_DELAY);*. Pierwszy argument informuje o tym, czy wartość notyfikacji powinna zostać wyczyszczona (pdTRUE), czy zdekrementowana (pdFALSE). Dzięki temu powiadamianie może działać odpowiednio jako semafor binarny, lub zliczający. Drugim argumentem jest maksymalny czas, jaki zadanie będzie oczekiwało na powiadomienie. Funkcja zwraca wartość powiadomienia przed modyfikacją, zależną od pierwszego argumentu.

Drugim sposobem użycia powiadomień jest przesyłanie wiadomości. Aby to zrobić wystarczy z poziomu innego zadania (GUI_Task w przykładzie) wywołać funkcję *xTaskNotify(appGlobals.fftTaskId, TASK_EVENT_CHANGE_VIEW_MOVE_LEFT, eSetBits);*. Jeżeli wiadomość ma zostać wysłana z przerwania, należy wywołać funkcję *xTaskNotifyFromISR(appGlobals.fftTaskId, TASK_EVENT_DMA_HALF_DONE, eSetBits, &xHigherPriorityTaskWoken);*. Pierwsze trzy argumenty obu funkcji oznaczają to samo:

- identyfikator zadania-adresata,
- wartość wiadomości,
- sposób ustawienia wartości.

Trzeci z argumentów mówi o tym w jaki sposób wiadomość ma być wpisana do wartości powiadomienia zadania. Dostępne opcje przedstawia **tabela 3**.

Jeżeli funkcja powiadamiania jest wywoływana z przerwania to ma jeszcze jeden argument. Jest to wskaźnik do zmiennej, w której zostanie zapisana flaga mówiąca o tym, czy na skutek wysłania powiadomienia nie zostało obudzone zadanie o wyższym priorytecie niż to wywłaszczone przez przerwanie. Jeżeli tak się stanie można rozkazać planiście jego natychmiastowe wykonanie po powrocie z funkcji obsługi przerwania, wywołując funkcję (będąc jeszcze w przerwaniu) *portYIELD_FROM_ISR(xHigherPriorityTaskWoken);*. Powyższy scenariusz powiadamiania znajduję się w obsłudze przerwania *void BSP_AUDIO_IN_HalfTransfer_CallBack(void);*. Obie funkcje powiadomień zwracają *pdPASS*, gdy wiadomość została poprawnie przekazania do zadania lub *pdFAIL* w razie błędu (może mieć on miejsce tylko w trybie *eSetValueWithoutOverwrite*, jeżeli poprzednia wiadomość nie została jeszcze odebrana).

Do odebrania wiadomości przez adresata służy funkcja *xTaskNotifyWait(0, UINT32_MAX, &notificationValue, portMAX_DELAY)*. Jej argumenty to odpowiednio:

- maska bitów do wyczyszczenia przed odebraniem wiadomości (0 pozostawia wiadomość bez zmian),
- maska bitów do wyczyszczenia po odebraniu wiadomości (UINT32_MAX czyści wszystkie bity),
- wskaźnik do zmiennej typu uint32_t, w której zostanie zapisana wiadomość,
- maksymalny czas oczekiwania na powiadomienie.

Tabela 3. Możliwe sposoby przekazywania wiadomości do powiadomienia zadania

eSetBits	Suma logiczna wartości wiadomości i powiadomienia
eIncrement	wartość powiadomienia jest inkrementowana, wartość wiadomości nie ma znaczenia
eSetValueWithOverwrite	wartość powiadomienia jest bezwzględnie nadpisywana przez wiadomość
eSetValueWithoutOverwrite	wiadomość jest wpisywana do wartości powiadomienia, jeżeli żadna inna wiadomość nie oczekuje na odbiór
eNoAction	Wartość powiadomienia nie jest modyfikowana, wartość wiadomości nie ma znaczenia

Funkcja zwraca *pdPASS*, jeżeli zostało odebrane powiadomienie, pdFAIL w przeciwnym razie. Przykład użycia opisanego sposobu komunikacji znajduje się w zadaniach *Signal_Task* oraz *FFT_Task*.

Kolejki. Kolejki są dobrze znanym sposobem komunikacji między zadaniami w systemach operacyjnych. FreeRTOS ma również implementację kolejek, którą można wykorzystać do przesyłania danych. Przed skorzystaniem z kolejki należy są utworzyć za pomocą funkcji *appGlobals.gestureQueue = xQueueCreate(1, sizeof(MTOUCH_GestureData_s));*. Funkcja ta przyjmuje liczbę oraz rozmiar elementów przesyłanych za pośrednictwem kolejki. W przykładzie, kolejka jest używana do przekazywania informacji o wykrytych gestach na panelu dotykowym. Gesty są przechowywane w strukturach typu *MTOUCH_GestureData_s* (więcej na ten temat w kolejnej części artykułu). Kolejka ma długość tylko jednego elementu, ponieważ każda nowa wartość powinna nadpisać starą, w przypadku, kiedy zadanie nie zdąży jej odebrać. Funkcja zwraca globalny identyfikator kolejki, dzięki któremu można się do niej odwołać w celu wysyłania i odbierania danych.

Istnieje kilka funkcji umożliwiających zapis danych do kolejki. Ze względu na nadpisywanie nieodebranych danych, wspomniane powyżej, w przykładzie użyta została funkcja *xQueueOverwrite(appGlobals.gestureQueue, &gestureData);*. Funkcja ta jest wywołana w funkcji obsługi licznika programowego i kopiuje dane ze struktury gestureData do kolejki o identyfikatorze przekazywanym w pierwszym argumencie. Ze względu na to, że funkcja zawsze wpisuje dane do kolejki, nawet jeżeli jest ona pełna, wartością zwracaną może być tylko pdPASS. Dane z kolejki odbierane są wewnątrz głównej pętli zadania GUI_Task za pomocą funkcji *xQueueReceive(appGlobals.gestureQueue, &gestureData, 0)*. Pierwszym argumentem jest ponownie identyfikator kolejki. Drugi argument to wskaźnik do miejsca, w którym zostanie zapisany element znajdujący się w kolejce. Na końcu listy argumentów znajduje się czas, na który wywołanie funkcji zablokuje zadanie w oczekiwaniu na odbiór danych. W przypadku wartości 0, funkcja kończy się od razu, bez względu na to czy zostały odebrane dane, czy nie – o tym informuje wartość zwracana, odpowiednio *pdTRUE* i *pdFALSE*.

Krzysztof Chojnowski