**ZACHODNIOPOMORSKI UNIWERSYTET**

**TECHNOLOGICZNY W SZCZECINIE**

WYDZIAŁ ELEKTRYCZNY



**Daniel Grzegorz Szot**

nr albumu: 16118

**Projekt i realizacja odtwarzacza audio formatów bezstratnych do współpracy z układem graficznym FPGA**

Praca dyplomowa inżynierska

kierunek: Automatyka i Robotyka

specjalność: Automatyzacja Procesów Przemysłowych

Opiekun pracy:

**dr inż. Krzysztof Penkala**  
Katedra Inżynierii Systemów, Sygnałów i Elektroniki  
Wydział Elektryczny

Szczecin 2013

**Oświadczenie autora**

Oświadczam, że przedkładaną pracę dyplomową pt. „Projekt i realizacja odtwarzacza audio formatów bezstratnych do współpracy z układem graficznym FPGA” napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zleciłem opracowania pracy lub jej części innym osobom oraz nie przypisałem sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego.

Załączona wersja elektroniczna pracy dyplomowej jest w pełni zgodna z wersją drukowaną.

Data:

Podpis autora

**Streszczenie pracy**

Streszczenie po Polsku.

**Słowa kluczowe**

RTOS, PIC32 itd.

**Streszczenie pracy**

Streszczenie po Polsku.

**Słowa kluczowe**

RTOS, PIC32 itd.

[Wprowadzenie 7](#_Toc347594539)

[1 Zastosowane technologie 9](#_Toc347594540)

[1.1 Hardware 9](#_Toc347594541)

[1.1.1 Procesor 10](#_Toc347594542)

[1.1.2 Płyta ewaluacyjna 12](#_Toc347594543)

[1.1.3 Układ graficzny 14](#_Toc347594544)

[1.2 Software 15](#_Toc347594545)

[1.2.1 Język programowania 15](#_Toc347594546)

[1.2.2 System operacyjny 15](#_Toc347594547)

[1.2.2.1 System operacyjny jako menadżer dostępu do zasobów 16](#_Toc347594548)

[1.2.2.2 System operacyjny jako maszyna wirtualna 16](#_Toc347594549)

[1.2.3 System plików 17](#_Toc347594550)

[1.2.4 Biblioteka graficzna 18](#_Toc347594551)

[2 Architektura systemu 19](#_Toc347594552)

[2.1 Struktura katalogów 19](#_Toc347594553)

[2.2 FreeRTOS 20](#_Toc347594554)

[2.3 Warstwa abstrakcji sprzętu 22](#_Toc347594555)

[2.3.1 Board init 24](#_Toc347594556)

[2.3.2 HLD 25](#_Toc347594557)

[2.3.3 LLD 28](#_Toc347594558)

[2.3.3.1 UART 28](#_Toc347594559)

[2.3.3.2 ADC 30](#_Toc347594560)

[2.3.3.3 IR 32](#_Toc347594561)

[2.3.3.4 AUDIO 35](#_Toc347594562)

[2.3.3.5 DISK 37](#_Toc347594563)

[2.3.3.6 LCD 39](#_Toc347594564)

[2.3.3.7 TOUCH 43](#_Toc347594565)

[2.4 Biblioteki 45](#_Toc347594566)

[2.4.1 Biblioteka standardowa 45](#_Toc347594567)

[2.4.1.1 Filtr cyfrowy 46](#_Toc347594568)

[2.4.1.2 LOG 47](#_Toc347594569)

[2.4.2 Konsola 48](#_Toc347594570)

[2.4.3 System plików 49](#_Toc347594571)

[2.4.4 Menadżer wejść 49](#_Toc347594572)

[2.4.5 Audio 49](#_Toc347594573)

[2.4.6 User 49](#_Toc347594574)

[2.4.7 Graficzny interfejs użytkownika 49](#_Toc347594575)

[2.4.7.1 Okno 49](#_Toc347594576)

[2.4.7.2 Tekst statyczny 49](#_Toc347594577)

[2.4.7.3 Przycisk 50](#_Toc347594578)

[2.4.7.4 Pasek postępu 50](#_Toc347594579)

[3 Aplikacja demonstracyjna 51](#_Toc347594580)

[3.1 Założenia 51](#_Toc347594581)

[3.2 Interfejs użytkownika 51](#_Toc347594582)

[Zawartość dodatkowej płyty CD 52](#_Toc347594583)

[Bibliografia 53](#_Toc347594584)

[Spis tabel 54](#_Toc347594585)

[Spis rysunków 55](#_Toc347594586)

[Spis kodów źródłowych 56](#_Toc347594587)

# Wprowadzenie

Tematem pracy jest opracowanie projektu jak i realizacja odtwarzacza audio przez co rozumie się napisanie oprogramowania oraz dobór platformy sprzętowej w postaci procesora i płyty ewaluacyjnej posiadającej odpowiednie peryferia na której będzie można przeprowadzić pierwsze testy funkcjonalne.

Do przygotowania oprogramowania spełniającego założone zadanie można podejść w dwojaki sposób. Pierwszy ze sposobów jest znacznie prostszy. Ogranicza się on do napisania kodu źródłowego działającego tylko i wyłącznie na wybranej platformie sprzętowej, używającego jedynie przerwań jako namiastkę programowania wielowątkowego i spełniającego jedynie jedną założoną funkcję – w tym przypadku odtwarzanie plików audio. W ten sposób omijamy wiele pracy potrzebnej aby napisane oprogramowanie było uniwersalne oraz wygodne do rozszerzenia lub całkowitej zmiany funkcjonalności urządzenia. Rozwiązanie to nie jest złe w przypadku gdy projekt jest stosunkowo prosty a zapotrzebowanie na przyszły rozwój znikome. W przypadku gdy chcemy stworzyć uniwersalną platformę, na której odtwarzacz audio będzie jedynie aplikacją demonstracyjną a możliwości rozszerzania oraz zmiany funkcjonalności będą w zasadzie nieograniczone – będziemy musieli zastosować sposób drugi. Polega on na stworzeniu oprogramowania bazującego na jednym z wybranych, systemów operacyjnych czasu rzeczywistego, przygotowaniu warstwy abstrakcji sprzętu w celu umożliwienia łatwego przeniesienia oprogramowania na zupełnie inną platformę sprzętową, oraz napisaniu odpowiednich bibliotek ułatwiających przygotowywanie konkretnych aplikacji. Wymienione czynności prowadzą w efekcie do stworzenia prostego uniwersalnego systemu operacyjnego wyłączając z tego niskopoziomowe zarządzanie przestrzenią adresową, zarządzanie wątkami oraz komunikację międzyprocesową (IPC[[1]](#footnote-1)) – funkcje te zostaną dostarczone przez wybrany system operacyjny czasu rzeczywistego (RTOS[[2]](#footnote-2)).

Niniejsza praca traktować będzie o procesie przygotowywania takiego oprogramowania od stworzenia pierwszego sterownika urządzenia, przez napisanie biblioteki do obsługi odtwarzania plików audio, aż do przygotowania aplikacji widocznej dla użytkownika.

?????????/MOŻE COS O ANGIELSKICH KOMENTARZACH???????

?I ZE KOD JEST NA GOOGLE CODE WRZUCONY?

ZE KOD NA PLYCIE BO NIE DALO SIĘ TUTAJ ZMIESCIC 30k LINI

GŁÓWNA DOKUMENTACJA PROJEKTU JEST KOD ZRODŁOWY

PRACA MA JEDYNIE ZGRUBNIE PREZENTOWAC KOD

CZYTAZ Z KODEM

**Cel pracy**

Celem pracy było Napisac o FPGA koniecznie.

**Zakres pracy**

asdasd

ROZDZIAŁ 1

# Zastosowane technologie

Proces budowania uniwersalnego oprogramowania pełniącego funkcję prostego systemu operacyjnego wymaga przygotowania zarówno od strony sprzętowej jak i programowej. Rozdział ten zostanie poświęcony opisowi procesu przygotowań do rozpoczęcia prac nad projektem. Zostaną w nim przedstawione dylematy na które natrafiłem zarówno przy wyborze pierwszej platformy sprzętowej na której oprogramowanie będzie rozwijane i testowane jak i przy podejmowaniu decyzji jaki język programowania wykorzystać oraz które biblioteki użyć gotowe, napisane przez ruch otwartego oprogramowania [1] a które przygotować samodzielnie.

## Hardware

Pomysł stworzenia niniejszej pracy dyplomowej powstał w momencie gdy kończyłem jeden z projektów w pracy zawodowej. Był to prosty układ oparty na procesorze Microchip PIC32MX460F512L, posiadający kolorowy wyświetlacz o przekątnej 1,8 cala i rozdzielczości 128x160 oraz prosty przetwornik cyfrowo analogowy wykorzystujący modulację szerokości impulsów (PWM[[3]](#footnote-3)). Układ ten miał zostać zastosowany w zabawce więc jakość odtwarzanego dźwięku nie musiała być wysoka. Urządzenie to było bazą dla kilku innych modeli podobnych zabawek. Oprogramowanie jednak wymagało modyfikacji pomiędzy każdą z nich a ponieważ zostało ono przygotowane pod konkretne rozwiązanie (czyli w sposób pierwszy opisany we wprowadzeniu) modyfikacje te były pracochłonne. Wtedy powstał pomysł stworzenia oprogramowania bazowego dzięki któremu będzie można bardzo szybko tworzyć projekty tego typu. Dodatkowym wymaganiem jakie zostało postawione dla nowego oprogramowania była wysoka wydajność w aplikacjach multimedialnych. Założenia te pozwoliły ustalić, iż platforma sprzętowa na której będzie rozwijane tworzone oprogramowanie musi posiadać:

1. Wydajny procesor
2. Wyświetlacz graficzny
3. Przetwornik cyfrowo analogowy
4. Wejście na kartę pamięci SD
5. Port RS232 (pomocny przy debugowaniu)
6. Odbiornik podczerwieni (sterowanie przy pomocy pilota)

Podrozdział ten przedstawiał będzie zastosowaną w pracy platformę oraz uzasadniał wykorzystanie wybranych elementów.

### Procesor

Wybór sprzętu na którym będziemy pracować warto rozpocząć od serca układu czyli procesora. W dzisiejszych czasach różnica między procesorem a mikrokontrolerem dąży do zatarcia ponieważ właściwie każdy układ scalony który nazywamy procesorem jest w istocie mikrokontrolerem. Popularny staje się termin SoC[[4]](#footnote-4) zarezerwowany dla procesorów dedykowanych do konkretnych rozwiązań np. odbiorników telewizji satelitarnej. Każdy z procesorów posiada przynajmniej kilka urządzeń peryferyjnych co pozwala sklasyfikować je jako mikrokontrolery. Dlatego w pracy terminy te będą używane zamiennie.

Wymagania które postawiliśmy sobie dla całej platformy sprzętowej implikują kryteria jakie powinniśmy wziąć pod uwagę przy wyborze procesora. Założyliśmy, że układ ma posiadać wysoką wydajność w aplikacjach multimedialnych. Nie chodzi tutaj oczywiście o stworzenie możliwości odtwarzania filmów w wysokiej rozdzielczości ani o możliwość uruchamiania trójwymiarowych gier. Pod pojęciem tym mam na myśli:

1. Przystosowanie układu do wyświetlania graficznego interfejsu użytkownika (GUI[[5]](#footnote-5)) oraz możliwie szybkiej reakcji tego interfejsu na polecenia ze strony użytkownika.
2. Możliwość odtwarzania plików audio co najmniej w tak zwanej jakości „CD”[[6]](#footnote-6)

Aby możliwe było spełnienie pierwszego założenia konieczne jest zapewnienie możliwości szybkiej komunikacji bezpośrednio z wyświetlaczem lub z układem graficznym. Obecnie na rynku możemy nabyć zarówno wyświetlacze bez wbudowanego układu graficznego, sterowane sygnałami RGB, oraz sygnałami synchronizacji pionowej i poziomej jak i wyświetlacze z wbudowanymi kontrolerami do których polecenia wysyłane są przez interfejs komunikacji równoległej. Jako że tematem pracy była realizacja odtwarzacza audio do współpracy z układem graficznym będziemy musieli zastosować procesor posiadający możliwość szybkiej komunikacji równoległej z innym układem scalonym. Oczywiście każdy z dostępnych na rynku mikrokontrolerów ma taką możliwość. Dla niektórych z nich będzie konieczna symulacja programowa sygnałów zapisu i odczytu oraz zegara a niektóre z nich mają wbudowane sprzętowe urządzenia peryferyjne które wspierają komunikację równoległą. Przykładem może być moduł transmisji równoległej (PMP[[7]](#footnote-7)) instalowany w procesorach firmy „Microchip”. Zastosowanie procesora posiadającego taki moduł wpłynie korzystnie zarówno na wydajność jak i prostotę implementacji protokołu komunikacji.

Spełnienie drugiego założenia będzie wymagało wybrania mikrokontrolera który posiada moduły komunikacji szeregowej I2C[[8]](#footnote-8) oraz SPI[[9]](#footnote-9) – są one wymagane do komunikacji z przetwornikiem cyfrowo analogowym. Podobnie jak poprzednio ich funkcjonalność można emulować programowo jednak wpływa to niekorzystnie na wydajność.

Istnieje wiele procesorów spełniających przedstawione założenia. Dzięki pracy zawodowej udało mi się poznać dość szczegółowo architekturę procesorów firmy „Microchip” i na wybór procesora tego producenta się zdecydowałem. Znajomość architektury nie była jedynym argumentem przemawiającym za tym wyborem. Procesory firmy „Microchip” posiadają bardzo dobrą, szczegółową i pozbawioną błędów dokumentację techniczną oraz są bardzo łatwo dostępne. Podczas realizacji projektu zabawki (patrz rozdział 1.1 na stronie 5) okazało się, że procesory 8-bitowe są zbyt wolne do zastosowań multimedialnych. Pierwszy prototyp tego urządzenia został zaprojektowany w oparciu o procesor PIC18. Jednak w trakcie tworzenia oprogramowania, po pierwszych testach uznaliśmy, że zastosowany został zbyt wolny procesor – wymusiło to zmianę na 32-bitową rodzinę procesorów PIC32. Został zastosowany model PIC32MX460F512L mimo tego, że okazał się wystarczający miał jeszcze pewne mankamenty. Kontroler DMA w nim zainstalowany pozwalał na zlecenie kopiowania jednorazowo maksymalnie 256 bajtów pamięci co przy rozmiarze jednostki alokacji dla karty SD o wielkości 512 bajtów wymaga zlecenia kopiowania dwa razy na jednostkę. Kolejnym mankamentem była znacznie ograniczona ilość pamięci operacyjnej, jest to 32 kB w przypadku tego modelu.

Aby uniknąć wymienionych problemów zdecydowałem się na procesor **PIC32MX795F512L** posiadający 128 kB pamięci operacyjnej oraz możliwość kopiowania przez kontroler DMA maksymalnie 64 kB danych. Procesor ten posiada zarówno moduł PMP jak i moduły I2C i SPI.

### Płyta ewaluacyjna

Pierwszą czynnością po wyborze procesora było znalezienie odpowiedniej płyty ewaluacyjnej spełniającej założenia dotyczące zainstalowanych na niej peryferii. Oczywiście można było wykonać własny prototyp płyty drukowanej idealnie dopasowany do potrzeb pracy, jednak nie było to głównym jej celem. Dlatego zdecydowałem się użyć gotowego rozwiązania.

Mimo tego że firma „Microchip” udostępnia wiele narzędzi deweloperskich dla swoich procesorów to jednak nie udało mi się znaleźć jednej płyty spełniającej postawione wymagania. Korzystając z produktów producenta procesora konieczne byłoby zakupienie trzech płyt i połączenie ich ze sobą (posiadają kompatybilne złącza). Wszystkie trzy płyty kosztują razem $210.

Pomocna okazała się być oferta firmy „MikroElektronika” która jest polecanym przez „Microchip” dostawcą narzędzi deweloperskich. Zainteresowała mnie płyta o nazwie „multimedia for PIC32MX7” posiadająca wszystkie wymagane przeze mnie peryferia oprócz odbiornika podczerwieni. Po jego dołożeniu płyta spełniała założenia w stu procentach dlatego zdecydowałem się ją użyć. Cena płyty wynosi $149 i została ona zasponsorowana na potrzeby tej pracy inżynierskiej przez firmę „MB Turnkey Design” za co serdecznie dziękuję.



**Rysunek 1.1.** Zdjęcie płyty "multimedia for PIC32MX7" z obu stron

Źródło: Na podstawie [3]

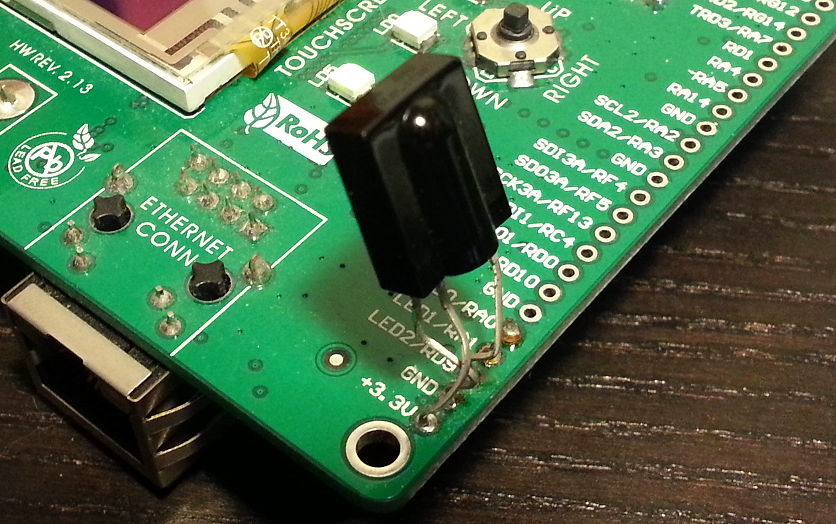
Na płycie zastosowany został kodek audio WM8731 wyprodukowany przez firmę „Wolfson”. Posiada on wbudowane dwa 24-bitowe przetworniki cyfrowo analogowe umożliwiające pracę z maksymalną częstotliwością próbkowania 96kHz [4]. To znacznie więcej niż założono. Układ ten posiada również wbudowany wzmacniacz słuchawkowy oraz również dwa przetworniki analogowo cyfrowe pozwalający na obsługę nie tylko wyjścia ale również wejścia stereo. Na wybranej płycie ewaluacyjnej wyprowadzone zostało stereofoniczne gniazdo słuchawkowe oraz monofoniczne gniazdo mikrofonowe. W pracy zostanie wykorzystane jedynie stereofoniczne wyjście audio.

Zastosowany na płycie wyświetlacz o rozdzielczości QVGA[[10]](#footnote-10) posiada wbudowany układ graficzny HX8347 obsługujący tryb RGB666 co daje możliwość wyświetlenia 256 tysięcy kolorów [5]. Docelowo projektowane urządzenie ma pracować z układem graficznym zrealizowanym na strukturze FPGA którego zaprojektowanie jest tematem innej pracy inżynierskiej (patrz rozdział Układ graficzny na stronie 5). Biorąc pod uwagę, iż prace nad oby dwoma projektami będą prowadzone równolegle, warto zapewnić sobie możliwość wcześniejszego uruchamiania i testowania aplikacji na rozwiązaniu tymczasowym w postaci gotowego układu graficznego.

Zgodnie z założeniami na płycie ewaluacyjnej znajdują się również takie peryferia jak gniazdo na kartę SD i port RS232 oraz dodatkowe, nie wymagane paryferia takie jak złącze Ethernet, czujnik temperatury, gniazdo USB (slave i host), cztery diody LED i joystick.

Wybrana płyta posiada możliwość zainstalowania dowolnych dodatkowych urządzeń dzięki wyprowadzeniu przez producenta płyty wszystkich portów zastosowanego mikrokontrolera na złącza znajdujące się na krawędziach laminatu. Tego rodzaju rozwiązanie pozwoli połączyć układ z osobą płytą ewaluacyjną z układem FPGA oraz umożliwi dołączenie jedynego brakującego urządzenia peryferyjnego którym jest odbiornik podczerwieni.

Zastosowany został odbiornik TSOP1138 [6] pracujący na częstotliwości nośnej 38kHz oraz kompatybilny z protokołem transmisji NEC [7]. Został on przylutowany do płyty wykorzystując złącza z wyprowadzeniami portów mikrokontrolera (patrz Rysunek 1.2 na stronie 5). Wybrany został NEC ponieważ jest to obecnie jeden z najczęściej stosowanych przez producentów sprzętu elektronicznego protokół.



**Rysunek 1.2.** Montaż odbiornika podczerwieni TSOP1138

Źródło: Opracowanie własne

### Układ graficzny

Chciałbym teraz powrócić do projektu opisanego we wstępie tego rozdziału. Na procesorze PIC32MX460L512 zastosowanym w tym układzie przy pomocy modułu PMP możliwe było wysłanie do sterownika wyświetlacza 160 ramek na sekundę. Wielkość ta jest imponująca jednak rozmywa się nieco z prawdą. Po pierwsze cała klatka obrazu musi być trzymana w pamięci operacyjnej procesora. W przypadku wyświetlacza o tak małej rozdzielczości oraz ograniczeniu formatu koloru piksela do RGB565 można sobie pozwolić na przetrzymywanie bufora ramki (FB[[11]](#footnote-11)) w pamięci operacyjnej. W przypadku większego wyświetlacza, zakładając dostępność jedynie 128 kB pamięci operacyjnej, staje się to niemożliwe. Po drugie obraz ten musi być najpierw w jakiś sposób wygenerowany – to właśnie ta operacja zajmuje najwięcej czasu procesora.

W obliczu tej sytuacji konieczne było zastosowanie dodatkowego układu graficznego odciążającego procesor. Możliwe było zastosowanie gotowego układu takiego jak na przykład „Solomon Systech SSD1926 Graphics Controller” lub stworzenie własnego układu graficznego opartego na programowalnym układzie logicznym (FPGA[[12]](#footnote-12)). Jako że Bartosz Zamolski zaproponował zaprojektowanie takiego układu graficznego, jako temat jego pracy inżynierskiej, stało się jasne, że moje urządzenie musi współpracować z tym właśnie układem.

## Software

Po wybraniu platformy sprzętowej dla projektowanego urządzenia, należało zastanowić się nad wyborem języka programowania oraz zdecydować, które części systemu zostaną oparte o gotowe rozwiązania. Podczas projektowania, stosunkowo rozbudowanej jak na system wbudowany aplikacji, zawsze należy zastanowić się, które fragmenty kodu będziemy tworzyć od zera a które wykorzystamy gotowe napisane przez kogoś innego. Tworzenie wszystkich elementów samodzielnie często prowadzi do niepowodzenia całego projektu. Podejście takie zajmuje znacznie więcej czasu a stworzone w ten sposób fragmenty oprogramowania często posiadają wiele błędów. Zawsze należy skupić się na fragmentach na których najbardziej nam zależy i dla których dostępność gotowych rozwiązań jest ograniczona.

W rozdziale tym został opisany proces podejmowania decyzji projektowych dotyczących programowej części pracy.

### Język programowania

W chwili rozpoczynania prac nad projektem, jedynym dostępnym językiem programowania dla procesorów z serii PIC32 był język „C”. Oczywiście znając architekturę procesora można zastosować dowolny kompilator wspierający daną architekturę. Jednak brak plików nagłówkowych określających rejestry procesora, środowiska programistycznego oraz możliwości debugowania aplikacji w sposób krokowy, skutecznie odstrasza od stosowania niewspieranych kompilatorów.

W trakcie trwania prac nad projektem firma „Microchip” wypuściła nową rodzinę kompilatorów dla swoich produktów. Kompilatory te wspierają zarówno język „C” jak i „C++”. W takiej sytuacji możliwe było zastosowanie języka „C” dla niskopoziomowych funkcji systemu, natomiast dla wysokopoziomowych „C++”. Zastosowanie języka obiektowego dla najwyższych warstw systemu spowodowałoby ułatwienie procesu tworzenia nowych aplikacji. Niestety stopień zaawansowania prac nad projektem wykluczył możliwość łatwej zmiany kompilatora.

### System operacyjny

Większość użytkowników komputerów oraz systemów wbudowanych posiada pewne doświadczenie związane z obsługą i zarządzaniem systemem operacyjnym, jednak trudno jest im określić czym właściwie jest system operacyjny. Dzieje się tak dlatego, że system operacyjny ma dwie niezwiązane ze sobą funkcje [8]:

1. Funkcja menadżera dostępu do zasobów
2. Funkcja maszyny wirtualnej

#### System operacyjny jako menadżer dostępu do zasobów

Obecnie komputery a nawet proste systemy wbudowane posiadają możliwość uruchamiania kilku aplikacji lub realizowania kilku zadań jednocześnie. Aby każde z uruchomionych zadań mogło korzystać z tych samych zasobów, np. sprzętowych, wymagane jest aby synchronizowały one między sobą kiedy które z nich będzie korzystało z danego zasobu. Zarówno za możliwość uruchamiania kilku wątków jednocześnie jak i komunikacji między nimi odpowiada moduł systemu operacyjnego zwany planistą (z ang. scheduler). Moduł ten jest absolutną podstawą działania każdego wielowątkowego systemu operacyjnego. Jeden błąd popełniony podczas jego pisania może implikować wiele trudnych do wychwycenia błędów występujących w aplikacjach. Dlatego właśnie tworzenie tego modułu systemu operacyjnego od podstaw jest ekstremalnie trudne oraz czasochłonne i mogłoby być tematem osobnej pracy inżynierskiej.

Jako że, każdy system operacyjny czasu rzeczywistego posiada planistę, postanowiłem skorzystać z jednego z nich. Ze względu na ilość wspieranych architektur procesorów, powszechność oraz licencję został wybrany FreeRTOS. Ciekawym aspektem tej pracy inżynierskiej będzie stworzenie w pełni funkcjonalnego systemu operacyjnego (z pewnymi wyjątkami) bazując na gotowym rozwiązaniu systemu czasu rzeczywistego.

#### System operacyjny jako maszyna wirtualna

Jak wiadomo każdy procesor jak i każde urządzenie wejścia/wyjścia posiada pewien zbiór instrukcji na podstawie których wykonuje operacje. Należałoby zastanowić się, czy programista piszący aplikację, powinien zastanawiać się jakich rozkazów procesora użyć oraz jakie polecenia wysłać do urządzeń peryferyjnych aby wykonać zadaną czynność. Oczywiście jeżeli istniałaby taka konieczność, to przez stopień skomplikowania tworzenia oprogramowania, większość obecnie dostępnych programów komputerowych nigdy by nie powstało. Aby lepiej zobrazować problem wykorzystajmy przykład rezystancyjnego panelu dotykowego. Gdyby system operacyjny nie pełnił opisywanej funkcji, programista piszący prostą aplikację okienkową (np. kalkulator) aby odczytać pozycję rysika na panelu musiałby wystawić stan wysoki na wejście X+ panelu oraz stan niski na wejście X- po czym odczytać stan z wyjścia Y+, operację powtórzyć, odfiltrować najbardziej odstające próbki po czym uśrednić pozostałe wyniki. Zamiast niego, może to zrobić system i udostępnić programiście jedną funkcję za pomocą której pobierze dokładną odfiltrowaną pozycję rysika. Idąc dalej programista nie musi nawet znać pozycji rysika – wystarczy mu informacja, że użytkownik wcisnął przycisk „+”. Te informacje przez różne warstwy abstrakcji udostępnia programiście system operacyjny.

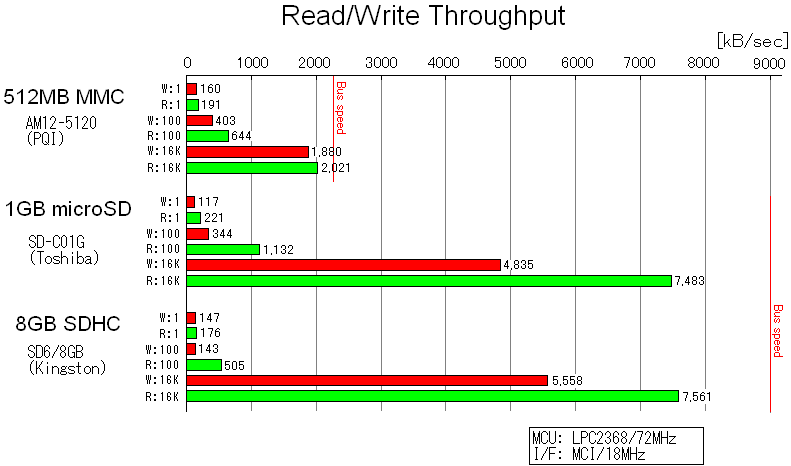
To właśnie nad tą funkcją systemu operacyjnego chciałbym skupić się w pracy najbardziej. Samodzielne przejście przez wszystkie etapy tworzenia tej funkcji pozwoliło mi dogłębnie zrozumieć zasady działania dzisiejszych systemów operacyjnych oraz zgłębić wiedzę dotyczącą projektowania tego typu systemów. Głównym źródłem informacji którym kierowałem się podczas tworzenia poszczególnych elementów projektu były kody źródłowe innych systemów operacyjnych tj. Linux, Haiku, ReactOS oraz ChibiOS.

### System plików

Głównym założeniem wynikającym z tematu pracy ma być możliwość odtwarzania plików audio. Warunkuje to konieczność implementacji wybranego systemu plików. Jako jeden z najczęściej stosowanych na pamięciach przenośnych systemów plików został wybrany system FAT32. Jest on powszechnie stosowany przez twórców różnego rodzaju systemów wbudowanych dzięki czemu istnieje wiele gotowych implementacji. Proces tworzenia własnej implementacji systemu plików, podobnie jak w przypadku planisty, jest procesem skomplikowanym oraz wymagającym specjalistycznej wiedzy i doświadczenia. Dlatego też zdecydowałem się użyć jednej z gotowych implementacji.

Projekt zabawki (patrz rozdział 1.1 na stronie 5) był oparty o implementację przygotowaną przez firmę „Microchip” do współpracy z ich procesorami. Okazało się, że jest ona bardzo wolna i w oryginalnej formie nie pozwoli na spełnienie założonych prędkości odczytu[[13]](#footnote-13). Konieczna była modyfikacja polegająca na zastosowaniu kontrolera DMA[[14]](#footnote-14) do kopiowania bloków pamięci z karty SD do pamięci procesora.

Po zapoznaniu się z wynikami testów wydajności (patrz rysunek 1.3 na stronie 17) oraz aby uniknąć problemów z koniecznością modyfikacji istniejącego rozwiązania zdecydowałem się na użycie implementacji o nazwie FatFS [9]. Testy zostały przeprowadzone na procesorze o podobnej częstotliwości taktowania więc spodziewamy się zbliżonych wyników.



**Rysunek 1.3.** Wyniki testów biblioteki FatFS

Źródło: Na podstawie [9]

### Biblioteka graficzna

Z tematu niniejszej pracy inżynierskiej wynika, iż projektowane urządzenie ma współpracować z układem graficznym zbudowanym na układzie FPGA. Biblioteka graficzna powinna być zaprojektowana tak, aby jak najlepiej pasowała do możliwości układu graficznego. Jest to główny argument dla którego zdecydowałem się na napisanie własnej biblioteki graficznej. Kolejnym argumentem „za”, była chęć stworzenia takiego graficznego interfejsu użytkownika jaki sobie wyobrażałem. Posiadanie możliwości wpływu na to jakie efekty graficzne są wspierane sprzętowo pobudza wyobraźnię.

Niestety, z powodu opóźnionego dostępu do płyt ewaluacyjnych, koledze nie udało się skończyć projektu układu graficznego na czas. Prace są nadal prowadzone, na bieżąco dodawane są nowe funkcje do sterownika układu graficznego.

W zastępstwie wykorzystany został układ graficzny wbudowany w wyświetlacz znajdujący się na wybranej w rozdziale 1.1.2 na stronie 5 płycie ewaluacyjnej. Kod źródłowy przystosowany jest do łatwego przełączania między tymi dwoma układami. Jeżeli tylko uda się zakończyć prace nad układem graficznym FPGA do czasu obrony – zaprezentowana zostanie wersja z tym właśnie układem.

Więcej szczegółów technicznych dotyczących biblioteki graficznej zostanie omówione w dalszej części pracy (patrz rozdział 2.4.7 na stronie 49).

ROZDZIAŁ 2

# Architektura systemu

Rozdział ten będzie prezentował architekturę oprogramowania które powstało na potrzeby pracy. Została przyjęta kolejność omawiania modułów od tych najniżej poziomowych do coraz wyższej warstwy abstrakcji. Na początku przedstawiona zostanie struktura katalogów w projekcie aby w razie ewentualnego czytania kodu równolegle z dokumentacją ułatwić czytelnikowi nawigację. Następnie przejdziemy do sposobu przystosowywania źródła do konkretnej płyty drukowanej. Kolejnym etapem będzie prezentacja warstwy abstrakcji sprzętu (HAL[[15]](#footnote-15)) oraz omówienie najważniejszych z obecnie dostępnych sterowników. Po omówieniu sterowników przejdziemy do prezentacji bibliotek.

## Struktura katalogów

Aktualnie oprogramowanie urządzenia składa się z kilkudziesięciu plików. Aby łatwo nimi zarządzać konieczne było przygotowanie drzewa katalogów podzielonego w taki sposób, aby każdy programista który po raz pierwszy widzi projekt, był w stanie bez trudu znaleźć poszukiwany sterownik czy bibliotekę. Kiedy projekt był rozpoczynany, taka struktura katalogów wydawała się być zbyt rozbudowana. Wraz z rozwojem projektu okazało się, iż pozwoliła ona w łatwy sposób zapanować nad porządkiem. Tabela 2.1 przedstawia obecne drzewo najważniejszych katalogów (niektóre, nieznaczące zostały pominięte).

**Tabela 2.1.** Struktura katalogów projektu

|  |  |
| --- | --- |
| Ścieżka katalogu | Zawartość |
| *./app* | Katalog zarezerwowany dla plików dodatkowych aplikacji. Aktualnie znajduje się w nim jedynie kod źródłowy odtwarzacza audio. |
| *./boards* | Katalog przeznaczony na pliki konfiguracyjne projektu dla konkretnych płyt PCB. Dokładny opis znajduje się w rozdziale 2.3.1 na stronie 5. |
| *./hal* | Katalog zarezerwowany na pliki warstwy abstrakcji sprzętu |
| *./hal/hld* | W katalogu tym znajdują się pliki nagłówkowe definiujące modele struktur opisujących sterowniki różnych typów (patrz rozdział 2.3.2 na stronie 5). |
| *./hal/lld* | Pliki źródłowe oraz nagłówkowe konkretnych sterowników sprzętu. |
| *./hal/lld/platforms* | Katalog na sterowniki zależne od wybranej platformy sprzętowej czyli procesora lub projektu płyty PCB. |
| *./lib* | Katalog zawiera źródła wszystkich bibliotek systemowych. Każda z nich posiada swój podkatalog. Pliki bibliotek wspólnych nie posiadają podkatalogu. |
| *./os* | Pliki źródłowe systemu FreeRTOS. |

## FreeRTOS

Jak zostało wcześniej wspomniane w projekcie za wielowątkowość oraz zarządzanie dostępem do zasobów odpowiedzialny będzie system operacyjny czasu rzeczywistego FreeRTOS. Podrozdział ten nie ma na celu prezentacji możliwości tego systemu ani opisywania jego poszczególnych funkcji. Zostanie tutaj przedstawiona jedynie konfiguracja tego systemu do pracy w naszym środowisku. Plik konfiguracyjny znajduje się pod ścieżką: „*./os/FreeRTOSConfig.h*”.

Jednym z parametrów konfiguracyjnych systemu FreeRTOS jest stała configUSE\_PREEMPTION, odpowiedzialna jest ona za włączenie wywłaszczania. Oczywiście w naszym przypadku została ustawiona na wartość „1” co oznacza włączenie danej funkcji. Włączenie wywłaszczanie oznacza, że wykonywany aktualnie wątek zostać w dowolnej chwili przerwany a czas procesora zostanie przekazany do innego wątku który aktualnie ma najwyższy priorytet a jego flaga odpowiadająca za gotowość jest ustawiona. W systemach bez wywłaszczania wykonywanie wątku zostanie przerwane jedynie w sytuacji gdy ten wyrazi na to zgodę (wykona tak zwaną procedurę „yeld” – nazwa zależna od systemu).

Kolejnym interesującym nas parametrem konfiguracyjnym jest configTICK\_RATE\_HZ. Stała ta definiuje z jaką częstotliwością mają być wykonywane kroki planisty. Dokładniej oznacza to czas na jaki zostanie ustawiony zegar który po przepełnieniu zgłasza przerwanie w którym wykonywany jest jeden krok planisty. Planista podczas wykonywania kroku może zmienić kontekst procesora, czyli zmienić wykonywany wątek. W naszym przypadku wartość to została ustawiona na 1000 Hz. Jest to stosunkowo wysoka wartość. Nie zostały jeszcze przeprowadzone testy wpływu tej wartości na ogólną wydajność systemu. Można uznać, że wartość ta jest tymczasowa.

Następne dwa parametry configCPU\_CLOCK\_HZ i configPERIPHERAL\_CLOCK\_HZ określają częstotliwości zegarów dla procesora oraz urządzeń peryferyjnych. W naszym przypadku dla obu wartości jest to 80Mhz i jest to maksymalna wartość z jaką może pracować zastosowany procesor.

Nieco niżej w pliku pojawiają się parametry odpowiedzialne za wielkość stosu przerwania (configISR\_STACK\_SIZE) oraz wielkość sterty (configTOTAL\_HEAP\_SIZE). Wielkość stosu dla przerwania została ustawiona na 1024. Jest to wysoka wartość, niestety przerwanie odpowiedzialne za pobieranie z pamięci oraz wysyłanie do przetwornika kolejnej próbki pliku audio wymaga tak dużego stosu. Sterta jest to miejsce w pamięci operacyjnej na które trafiają wszystkie zmienne zadeklarowane dynamicznie. Wielkość ta została ustawiona na 64kB czyli dokładnie połowę dostępnej pamięci.

Przy okazji omawiania wielkości sterty warto wspomnieć o wybranym modeli pamięci. FreeRTOS oferuje aż cztery modele do wyboru. Model pierwszy pozwala jedynie alokować pamięć. Raz zadeklarowanej pamięci nie można już zwolnić. Model drugi pozwala zarówno alokować jak i zwalniać pamięć, jednak po zwolnieniu pamięci nie grupuje zwolnionych bloków w jeden ciągły obszar pamięci. Trzeci model jest jedynie zapewnieniem synchronizacji między wątkami dla poleceń malloc i free używanego kompilatora. Model ten nie jest zalecany ponieważ powoduje wzrost objętości kodu. Czwarty model działa podobnie do modelu drugiego z tą różnicą, że powoduje grupowanie zwolnionych bloków pamięci. Jako że model czwarty jest zalecany jedynie w przypadku alokowania bloków pamięci o losowych rozmiarach, a my tego nie robimy, w projekcie zastosowany został model drugi.

Następne istotne parametry definiują jedynie włączenie lub wyłączenie poszczególnych funkcji systemu. W omawianym projekcie wszystkie funkcje poza współprogramami (ang. co-routines) są włączone.

FreeRTOS udostępnia dwa bardzo pomocne podczas tworzenia oprogramowania wywołania zwrotne. Jedno z nich odpowiada za wystąpienie wyjątku procesora (np. dzielenie przez zero) drugie natomiast wywoływane jest w przypadku wystąpienia przepełnienia stosu.

W przypadku wystąpienia wyjątku procesora odczytanie wartości kilku rejestrów może być niezwykle pomocne podczas debugowania aplikacji. Dla architektury MIPS są to rejestry EPC, STATUS oraz CAUSE. Rejestr EPC[[16]](#footnote-16) przechowuje licznik programu w chwili wystąpienia wyjątku. Na podstawie tej wartości możemy dokładnie określić linię kodu w której wystąpił problem. Rejestry STATUS oraz CASUE odpowiadają kolejno za status procesora oraz powód wystąpienia wyjątku. Na podstawie wartości zapisanych w tych trzech rejestrach możemy szczegółowo określić problem jaki wystąpił. Aby ułatwić uzyskiwanie wartości tych rejestrów w wywołaniu zwrotnym została zapisana procedura która drukuje je na ekranie. Jest to typowe zachowanie systemu operacyjnego znane z systemu „Windows” pod niechlubną nazwą „Blue Screen Of Death”. Rysunek 2.1 przedstawia efekt przykładowego, wywołanego sztucznie, wyjątku procesora. Dokładne informacje na temat rejestrów procesorów MIPS oraz ich szczegółowe opisy znajdują się w pozycji [10] bibliografii.

!!!!RYSUNEK!!!!!

**Rysunek 2.1.** Wyjątek procesora. Wartości rejestrów EPC, STATUS oraz CAUSE wydrukowane na ekran

Źródło: Opracowanie własne

Podobnie w przypadku wystąpienia przepełnienia stosu, pomocne podczas debugowania jest wyciągnięcie z systemu informacji o tym, który wątek spowodował przepełnienie oraz o jaką ilość danych został on przepełniony. Ta sytuacja również została obsłużona wyświetleniem pomocnych informacji na ekranie. W celu ułatwienia identyfikacji czy wystąpił wyjątek czy przepełnienie, przypadek ten został wyświetlony na czerwonym tle. Zdjęcie ekranu po wystąpieniu zasymulowanego przepełnienia stosu przedstawia Rysunek 2**.2**.

!!!!!!!!!RYSUNEK!!!!!!!

**Rysunek 2.2.** Przepełnienie stosu. Zwizualizowana nazwa wątku oraz adresy początku i końca stosu

Źródło: Opracowanie własne

Szczegółowe informacje na temat obsługi systemu FreeRTOS dostępne są w: [11]. Wewnętrzna budowa systemu opisana została w [12].

## Warstwa abstrakcji sprzętu

W ujęciu systemu operacyjnego warstwa abstrakcji sprzętu jest zaimplementowaną w oprogramowaniu warstwą pomiędzy platformą sprzętową a oprogramowaniem. Jej funkcją jest ukrycie różnic w sprzęcie tak aby programista piszący aplikację, na dany system operacyjny nie musiał używać poleceń specyficznych dla konkretnego urządzenia a zamiast tego stosował uniwersalne odwołania specyficzne dla tego typu urządzeń. Kolejną funkcją jaką zawdzięczamy warstwie abstrakcji sprzętu jest możliwość uruchamianie systemu operacyjnego na różnym sprzęcie bez konieczności ponownej kompilacji. Funkcja ta jest zarezerwowana dla rozbudowanych systemów operacyjnych uruchamianych na różnych, aczkolwiek podobnych pod względem architektury platformach sprzętowych. W ujęciu systemów wbudowanych dynamiczne wykrywanie podłączonego sprzętu ma sens, ale tylko w ściśle zdefiniowanym zakresie np. wykrycie typu głowicy w odbiorniku telewizyjnym.

Nawet jeżeli zrezygnujemy z dynamicznej detekcji platformy sprzętowej dzięki warstwie abstrakcji sprzętu znacznie ułatwimy sobie przenoszenie na inną platformę. Wystarczy przygotować odpowiedni zestaw sterowników, aby bez modyfikacji wysokopoziomowej części oprogramowania uruchomić system na zupełnie innym sprzęcie. Oczywiście będzie to wymagało kompilacji, ale jak już zostało powiedziane w przypadku systemów wbudowanych nie jest to przeszkodą.

Projekt realizowany na potrzeby tej pracy miał się wykazywać uniwersalnością oraz łatwością zastosowania do przyszłych rozwiązań – właśnie dlatego została w nim zaimplementowana prosta, aczkolwiek w pełni funkcjonalna warstwa abstrakcji sprzętu. Pierwszym momentem w którym okazała się pomocna była wymiana wyświetlacza z zainstalowanego na płycie ewaluacyjnej na zewnętrzy wyświetlacz z układem graficznym FPGA. Jeżeli chcemy skompilować program do współpracy z układem graficznym FPGA dodajemy do pliku konfiguracyjnego platformy sprzętowej definicję LCD\_FPGA. Dzięki temu kompilator wie który sterownik wyświetlacza wybrać (patrz Kod źródłowy 2.1).

**Kod źródłowy 2.1.** Wybór sterownika wyświetlacza w czasie kompilacji

#ifdef LCD\_FPGA

ret = lldFpgaGpuAttach(); // Select FPGA based GPU

#else

ret = lldHx8347Attach(); // Select HX8347 graphics controller

#endif

Znaczenie funkcji zostanie dokładnie lldXXXAttach() zostanie opisane w dalszej części pracy (patrz rozdział 2.3 na stronie 22).

### Board init

Pierwszym bardzo prostym elementem warstwy abstrakcji sprzętu jest moduł inicjalizacji systemu do współpracy z płytą PCB, na potrzeby której kompilowane jest oprogramowanie.

W strukturze katalogów zawarty został katalog „boards”. W jego wnętrzu znajdują się podkatalogi przypisane do konkretnych płyt. Poprzez ustawienia projektu wybieramy który z tych katalogów ma zostać dołączony do kompilacji. W ten sposób wybieramy płytę do współpracy z którą będzie kompilowane oprogramowanie. Każdy z tych katalogów posiada dwa pliki „board.c” oraz „board.h”.

W pliku nagłówkowym konieczne jest zdefiniowanie pewnych stałych konfiguracyjnych, wymaganych przez system bądź niektóre sterowniki, oraz zdefiniowanie porów procesora do których podłączone są pewne urządzenia. Na przykład sterownik wyświetlacza, wymusza konieczność zdefiniowania orientacji ekranu, oraz portu, pod który podłączone jest sterowanie podświetleniem (patrz Kod źródłowy 2.1). Natomiast sterownik panelu dotykowego, fabrycznych wartości kalibracji, oraz kanałów przetwornika analogowo cyfrowego, do których jest on podłączony. Mogą się tutaj znaleźć również definicje określone i wykorzystywane później przez użytkownika. Taką definicją jest na przykład opisywana wcześniej LCD\_FPGA.

**Kod źródłowy 2.2.** Konfiguracja sprzętowa sterownika wyświetlacza

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// BOARD Configuration

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define LCD\_ORIENTATION 90

#define LCD\_FPGA

#define LCD\_FPGA\_TEST

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// GPIO Definitions

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#ifdef LCD\_FPGA

#define LCD\_BACKLIGHT\_GPIO LATDbits.LATD2 // not used

#define LCD\_RESET\_GPIO LATCbits.LATC1 // not used

#define LCD\_RS\_GPIO LATBbits.LATB5

#define LCD\_CS\_GPIO LATAbits.LATA10

#else

#define LCD\_BACKLIGHT\_GPIO LATDbits.LATD2

#define LCD\_RESET\_GPIO LATCbits.LATC1

#define LCD\_RS\_GPIO LATBbits.LATB15

#define LCD\_CS\_GPIO LATBbits.LATB14

#endif

Plik źródłowy „board.c” posiada definicję jednej funkcji o nazwie boardInit (patrz Kod źródłowy 2.3). W funkcji tej należy wykonać wszystkie czynności konieczne do uruchomienia płyty. Są to na przykład: ustawienie częstotliwości taktowania procesora, włączenie kontrolera przerwań, zdefiniowanie kierunków portów itp. Czynności te, są zależne od konkretnej platformy sprzętowej.

**Kod źródłowy 2.3.** Prototyp funkcji boardInit

void boardInit()

{

//...

}

### HLD

Moduł HLD[[17]](#footnote-17) warstwy abstrakcji sprzętu odpowiedzialny jest za zarządzanie sterownikami zainstalowanymi w systemie oraz definicję typów sterowników które mogą zostać dodane niego dodane. Została tutaj zdefiniowana struktura bazowa opisująca każdy sterownik (patrz Kod źródłowy 2.4 na stronie 25). Struktury opisujące konkretne typy sterowników będą z niej „dziedziczyć”. Ze względu na zastosowanie języka „C”, który nie wspiera obiektowości, konieczna była symulacja dziedziczenia.

**Kod źródłowy 2.4.** Struktura bazowa dla sterowników

struct hldDevice

{

struct hldDevice \*next;

hldDeviceType type;

UINT16 id;

INT8 name[HLD\_DEVICE\_MAX\_NAME\_LEN];

hldDeviceState state;

// Private data of each device

// If you would like to use it in your lld you need to

// allocate memmory for this using pvPortMalloc

void \*priv;

};

W systemie musi istnieć lista zainstalowanych (niekoniecznie używanych) sterowników. Pierwszy element struktury odpowiedzialny jest właśnie za obecność na liście zainstalowanych sterowników. Jest to wskaźnik na następny element listy. Wskaźnik na pierwszy element listy został zadeklarowany w pliku „*/hal/hld/hld.c*” a wartość do niego przypisywana jest podczas dodawania pierwszego sterownika. Sterowniki dodajemy wywołując polecenie hldDeviceAdd.

Kolejnym elementem jest typ urządzenia. Dzięki temu elementowi możemy korzystając z funkcji hldDeviceGetByType pobrać pierwsze urządzenie danego typu z listy. Często jest tak, że urządzenie danego typu występuje tylko jedno w systemie. Wtedy polecenie to jest przydatne w celu pobrania wskaźnika np. na urządzenie wyświetlacza z dowolnego miejsca programu. Kod źródłowy 2.5 przedstawia przykład wykorzystania omawianej funkcji.

**Kod źródłowy 2.5.** Przykład wykorzystania funkcji hldDeviceGetByType

struct hldLcdDevice \*lcd = hldDeviceGetByType(NULL,

HLD\_DEVICE\_TYPE\_LCD);

graphSetDrawingColor(0xff, 0xff, 0xff, 0xff);

graphDrawText(10, 10, 110, 30, "IntenseOS", &g\_FontHaveltica26AA,

FS\_VALIGN\_CENTER | FS\_ALIGN\_CENTER);

Przy okazji omawiania typów urządzeń warto wspomnieć, iż w systemie został przewidziany jeden specyficzny typ urządzeń. Są to urządzenia znakowe. Charakteryzują się one dwoma udostępnianymi poleceniami tj. wysłanie znaku do urządzenia oraz odczyt znaku z urządzenia. Z tego typu sterowników korzystają urządzenia takie jak UART czy I2C.

Pole o nazwie „id” jest identyfikatorem urządzenia. Jest on przypisywany każdemu urządzeniu automatycznie, podczas alokowania pamięci na sterownik (funkcja hldDeviceAlloc). Identyfikator jest kolejnym numerem począwszy od „1” dla danego typu sterownika. Czyli np. pierwszy dodany do systemu sterownik wyświetlacza, będzie miał zawsze identyfikator równy „1”, niezależnie od sterowników innego typu istniejących w systemie. Zakładając, że znamy kolejność dodawania sterowników na listę, używając polecenia hldDeviceGetById pobrać wskaźnik do np. drugiego portu transmisji szeregowej.

Kolejne pole odpowiada za nazwę urządzenia. Nazwa ta musi być unikalna wewnątrz całego systemu. Dlatego w przypadku chęci podłączenia kilku takich samych urządzeń korzystających z tego samego sterownika należy do nazwy dodać numer urządzenia. Rozwiązanie to zostało zastosowane np. dla sterownika portu transmisji szeregowej. Korzystając z nazwy urządzenia, również możemy pobrać wskaźnik na urządzenie. W tym celu wykorzystujemy funkcję hldDeviceGetByName.

Przedostatni element struktury hldDevice określa stan urządzenia. W chwili obecnej dostępne są jedynie dwa stany urządzenia. Jeden z nich określa, że urządzenie zostało włączone. Drugi natomiast czy urządzenie pracuje poprawnie (patrz Kod źródłowy 2.6 na stronie 26). Znajomość możliwych stanów urządzenia jest ważna, ponieważ podczas pisania sterownika niskiego poziomu, będziemy musieli je odpowiednio przełączać.

**Kod źródłowy 2.6.** Możliwe stany pracy urządzenia

typedef enum

{

HLD\_DEVICE\_STATE\_TURNED\_ON = 0x00010001, // Device have supply

HLD\_DEVICE\_STATE\_RUNNING = 0x00010002, // Device is running

} hldDeviceState;

Ostatnie pole o nazwie „priv” jest wskaźnikiem na dowolny obszar pamięci zadeklarowany przez sterownik niskiego poziomu. Czyli jeżeli sterownik który będziemy pisać, potrzebuje tak zwanych danych prywatnych zarezerwowanych tylko dla niego, może zaalokować pamięć i przetrzymywać wskaźnik do zaalokowanego obszaru w tym właśnie polu (patrz Kod źródłowy 2.7 na stronie 26). Posiadanie takiej możliwości jest szczególnie istotne podczas tworzenia sterownika który może być wykorzystany wielokrotnie do obsługi kilku takich samych urządzeń. Wtedy niemożliwe jest zadeklarowanie zmiennych statycznych w pliku sterownika, ponieważ będą one wspólne dla wszystkich urządzeń obsługiwanych przez ten sterownik. Funkcjonalność ta została wykorzystana w sterowniku portu komunikacji szeregowej.

**Kod źródłowy 2.7.** Wykorzystanie pola priv struktury hldDevice

struct hldUartDevice \*dev;

struct lldPic32UARTPrivateData \*priv;

//...

priv = pvPortMalloc(sizeof(struct lldPic32UARTPrivateData));

if (priv == NULL)

{

hldDeviceFree(dev);

return ERR\_NO\_MEMMORY;

}

//...

dev->charHead.head.priv = priv;

Struktury opisujące poszczególne typy sterowników, zostaną opisane przy okazji omawiania implementacji sterowników dla konkretnych urządzeń (patrz rozdział 2.3.3 na stronie 27).

### LLD

Moduł LLD[[18]](#footnote-18) jest właściwie jedynie katalogiem przygotowanym do przechowywania sterowników niskiego poziomu. Definiujmy je jako sterowniki do konkretnych urządzeń, np. kodeka audio WM8731. W niniejszym podrozdziale opisane zostaną najważniejsze sterowniki zaimplementowane na potrzeby powstałego urządzenia. Kolejne podrozdziały przyjęły nazwy typów sterowników które będziemy implementować. W każdym z nich znajdzie się również opis struktury „dziedzicznej” określającej dany typ sterownika z modułu HLD.

#### UART

Sterownik urządzenia portu transmisji szeregowej (UART[[19]](#footnote-19)) był pierwszym sterownikiem który został napisany na potrzeby projektu. Jest to jeden ze sterowników implementujących typ urządzenia znakowego. Został napisany dla modułu UART mikrokontrolerów z rodziny PIC32.

Kod źródłowy 2.8. Warstwa abstrakcji dla sterowników portu transmisji szeregowej

struct hldUartDevice

{

struct hldCharDevice charHead;

struct hldUartConfig config;

xQueueHandle rxedCharsQueue;

xQueueHandle charsForTxQueue;

retcode (\*attach)();

retcode (\*open)(struct hldUartDevice \*pUartDev,

struct hldUartConfig \*pCfg);

retcode (\*close)(struct hldUartDevice \*pUartDev);

};

Pole charHead jest strukturą opisującą urządzenie znakowe (patrz Kod źródłowy 2.9 na stronie 28). Jak zostało wcześniej wspomniane sterownik urządzenia znakowego musi implementować dwie funkcje:

1. Zapis jednego znaku (funkcja write)
2. Odczyt jednego znaku (funkcja read)

**Kod źródłowy 2.9.** Warstwa abstrakcji dla sterowników urządzeń znakowych

struct hldCharDevice

{

struct hldDevice head;

retcode (\*write)(struct hldCharDevice \*pCharDev,

UINT8 pByte, UINT32 pTimeout);

retcode (\*read)(struct hldCharDevice \*pCharDev,

UINT8 \*pByte, UINT32 pTimeout);

};

Kolejnym polem jest pole config typu struct hldUartConfig. Struktura ta posiada pola służące do konfiguracji urządzenia. W przypadku urządzenia portu transmisji szeregowej są to takie wartości jak prędkość transmisji, ilość bitów stopu, typ kontroli parzystości, priorytet przerwania urządzenia itd. Wskaźnik na taką strukturę z wpisanymi oczekiwanymi wartościami przekazujmy do funkcji uruchamiającej urządzenie (open).

Kolejne dwa pola są kolejkami FIFO[[20]](#footnote-20) służącymi do buforowania danych wejściowych oraz wyjściowych. Programista piszący sterownik nie jest zobligowany do wykorzystywania tych kolejek, jest to jedynie opcja. Niektóre procesory posiadają sprzętowe kolejki, jeżeli ich rozmiar okaże się wystarczający nie jest potrzebna dodatkowa implementacja buforowania programowego.

Kolejnym polem jest wskaźnik na funkcję dodającą sterownik urządzenia do listy sterowników systemu o nazwie attach. Funkcja ta jest odpowiedzialna za alokację wymaganej dla sterownika pamięci, sprawdzenie unikalności nazwy oraz dodania go do listy zainstalowanych sterowników. Funkcja attach przypisuje również adresy funkcji konkretnego sterownika do struktury go opisującej. Dzięki temu możliwe są później odwołania w stylu uart->open(…) które po tym przypisaniu odnoszą się do konkretnego sterownika. Rozwiązanie to jest główną zasadą działania całej warstwy abstrakcji sprzętu.

Pola o nazwach open i close są wskaźnikami na funkcje uruchamiającą oraz zatrzymującą pracę urządzenia. Funkcje te powinny obsługiwać zmianę stanów urządzenia. W przypadku większości typów sterowników funkcja open przyjmuje jako dodatkowy parametr wskaźnik na strukturę konfiguracyjną urządzenia.

Urządzenie portu szeregowego jest traktowane jako urządzenie wejściowe. Oznacza to, iż po odebraniu znaku, sterownik urządzenia powinien powiadamiać menadżera wejść (patrz rozdział 2.4.4 na stronie 49) o wystąpieniu zdarzenia.

Funkcje takie jak attach, open oraz close w przypadku pełnienia takiej samej funkcji dla innych urządzeń będą pomijane w dalszej części pracy.

Przygotowany na potrzeby rodziny procesorów PIC32 sterownik portu szeregowego został zaprojektowany w taki sposób aby wysyłanie znaków było możliwie jak najszybsze. Głównym założeniem była jednak możliwość swobodnego korzystania z urządzenia z poziomu różnych wątków. W tym celu wykorzystane zostały udostępniane przez warstwę abstrakcji sprzętu programowe kolejki FIFO. Dzięki temu, że są to kolejki udostępniane przez system FreeRTOS, mają zapewnioną synchronizację między-wątkową. Sterownik portu transmisji szeregowej dla PIC32 używa przerwań od modułu UART zarówno do wysyłania jak i odbierania danych.

Zaimplementowana została możliwość dołączenia kilku takich samych sterowników dla kolejnych modułów UART procesora PIC32. Wystarczy w strukturze konfiguracyjnej ustawić pole portNumber na numer portu który chcemy używać a sterownik sam zajmie się przydzielaniem znaków odebranych w przerwaniu do odpowiednich kolejek wejścia/wyjścia.

Po ustawieniu pola enableLoopback struktury konfiguracyjnej na wartość „1” sterownik w chwili odebrania znaku, będzie automatycznie odpowiadał takim samym znakiem.

#### ADC

Sterownikiem koniecznym do zapewnienia obsługi panelu dotykowego, która była jednym z założeń projektu, jest sterownik przetwornika analogowo cyfrowego.

Budowa warstwy abstrakcji sprzętu wymusza pewien schemat budowy sterownika (patrz Kod źródłowy 2.10 na stronie 29).

**Kod źródłowy 2.10.** Warstwa abstrakcji dla sterowników przetworników ADC

struct hldAdcDevice

{

struct hldDevice head;

struct hldAdcConfig config;

retcode (\*attach)(struct hldAdcConfig \*pCfg);

retcode (\*open)(struct hldAdcDevice \*pAdcDev, UINT8 channel);

retcode (\*close)(struct hldAdcDevice \*pAdcDev, UINT8 channel);

retcode (\*read)(struct hldAdcDevice \*pAdcDev, UINT8 pChannel,

UINT32 \*pValue);

BOOL (\*isChannelOpened)(struct hldAdcDevice \*pAdcDev,

UINT8 channel);

};

Specyficzna jest tutaj funkcja open. Przyjmuje ona parametr channel określający który kanał przetwornika chcemy uruchomić. W przeciwieństwie do sterownika portu transmisji szeregowej nie musimy dołączać osobnych sterowników dla każdego kanału. Wystarczy jeden zbiorczy sterownik który może obsłużyć dowolną ilość kanałów (zależnie od ilości wspieranej przez przetwornik). Do odczytu aktualnej wartości z przetwornika używamy funkcji read do której jako drugi parametr przekazujemy numer interesującego nas kanału.

Procesory z rodziny PIC32 mają wbudowany 16-kanałowy, 10-bitowy przetwornik analogowo cyfrowy. Budowa wewnętrzna przetwornika znacznie ułatwiła prace nad uniwersalnym sterownikiem. Posiada on bowiem 16 elementowy bufor odczytów, oraz możliwość konfiguracji, w jakiś sposób te odczyty mają być do niego wpisywane. Istnieje możliwość skonfigurowania przetwornika w taki sposób, aby wartość odczytu dla każdego kanału została wpisana do innego elementu bufora. Dzięki możliwości ustalenia co ile odczytów ma być zgłaszane przerwanie, możemy zażądać jego zgłoszenia w momencie, kiedy będą dostępne nowe dane dla wszystkich kanałów, czyli po 16 odczytach [13]. Dzięki takiemu rozwiązaniu, w procedurze obsługi przerwania, jedynie przepisujemy dane ze sprzętowych buforów mikrokontrolera do tablicy która będzie je przechowywała do czasu zgłoszenia kolejnego przerwania (patrz Kod źródłowy 2.11 na stronie 30). Rozwiązanie to uprościło funkcję odczytującą wartość danego kanału do minimum (patrz Kod źródłowy 2.12 na stronie 31).

**Kod źródłowy 2.11.** Funkcja obsługi przerwania przetwornika

void lldPic32ADCIntHandler()

{

// Zadeklarowane jako static aby zminimalizować użycie stosu

static portBASE\_TYPE higherPriorityTaskWoken;

higherPriorityTaskWoken = FALSE;

adcInfo[0].lastReading = ADC1BUF0;

adcInfo[1].lastReading = ADC1BUF1;

//...

adcInfo[15].lastReading = ADC1BUFF;

IFS1CLR = 0x0002;

/\* If sending or receiving necessitates a context switch,

\* then switch now. \*/

portEND\_SWITCHING\_ISR( higherPriorityTaskWoken );

}

Parametry konfiguracyjne sterownika, zostały ograniczone jedynie do czasu akwizycji, okresu zegara przetwornika, oraz priorytetu przerwania.

**Kod źródłowy 2.12.** Funkcja odczytująca wartość podanego kanału przetwornika

static inline retcode \_\_attribute\_\_ ((always\_inline))

lldPic32ADCRead(struct hldAdcDevice \*pAdcDev, UINT8 pChannel,

UINT32 \*pValue)

{

assert(pAdcDev != NULL);

assert(pChannel <= LLD\_PIC32\_ADC\_MAX);

\*pValue = adcInfo[pChannel].lastReading;

assert(pValue != NULL);

return SUCCESS;

}

Kod źródłowy sterownika przetwornika ADC znajduje się na dołączonej płycie CD pod ścieżką: „*./source/hal/lld/platforms/microchip/pic32ADC.c*”.

#### IR

W warstwie abstrakcji sprzętu, zgodnie z założeniami projektu, znalazło się również miejsce dla sterownika odbiornika podczerwieni. Sterownik ten, ma za zadanie mierzyć czasy pomiędzy zboczami, które wystąpią w sygnale od odbiornika podczerwieni a następnie przekazywać je do dekodera określonego systemu kodowania.

Struktura bazowa dla tego typu sterownika (patrz Kod źródłowy 2.13 na stronie 31) wygląda podobnie do struktury urządzenia znakowego.

**Kod źródłowy 2.13.** Warstwa abstrakcji dla sterownika odbiornika podczerwieni

struct hldIrDevice

{

struct hldDevice head;

struct hldIrConfig config;

retcode (\*attach)(struct hldIrConfig \*pCfg);

retcode (\*open)(struct hldIrDevice \*pIrDev);

retcode (\*close)(struct hldIrDevice \*pIrDev);

retcode (\*read)(struct hldIrDevice \*pIrDev,

UINT32 \*pLastCode);

};

Specyficzna jest tutaj jedynie funkcja read. W odróżnieniu od urządzenia znakowego odczytuje ona 32-bitową wartość zamiast 8-bitowej. Jest to kod odczytany z odbiornika podczerwieni po analizie wykonanej przez dekoder. Jako jedyny parametr konfiguracyjny podajemy wskaźnik do funkcji dekodera (patrz Kod źródłowy 2.14 na stronie 32). Funkcja ta na podstawie czasu od ostatniego zbocza, flag odbiornika oraz czasu od włączenia urządzenia w milisekundach, ma za zadanie określić jaki kod został wysłany. Flagi odbiornika określają czy ostatnie zbocze było zboczem narastającym czy opadającym, czy było to pierwsze zbocze oraz czy czas od ostatniego zbocza nie jest podejrzanie długi (flaga timeout). To sterownik urządzenia musi kontrolować stan tych flag. Czas w milisekundach wymagany jest do wykrycia powtórzeń spowodowanych długim przytrzymaniem przycisku pilota. Czasy te są stosunkowo długie w stosunku do odstępów między zboczami, dlatego musiał zostać zastosowany dodatkowy licznik czasu o zmniejszonej rozdzielczości.

**Kod źródłowy 2.14.** Parametry konfiguracyjne sterownika odbiornika podczerwieni

struct hldIrConfig

{

UINT32 (\*decode)(UINT32 pLastPulseTime,

union hldIrFlags \*pFlags, UINT32 pTicks);

};

Obecnie każde urządzenie może mieć przypisany tylko jeden dekoder. W planach jest rozszerzenie tej funkcjonalności do nieograniczonej liczby dekoderów. Ma to na celu umożliwienie urządzeniu jednoczesnego wykrywania kodów dowolnego pilota i na przykład uczenia się ich.

W rodzinie procesorów PIC32 zastosowane zostało urządzenie peryferyjne o nazwie „Input Capture” [14] służące między innymi do wykrywania zboczy w sygnale oraz mierzenia czasu między nimi. Nadaję się ono idealnie do zbudowania na jego podstawie odbiornika podczerwieni i do tego właśnie zostało wykorzystane.

Aby urządzenie było zdolne do pomiaru czasu pomiędzy zboczami, musimy zainicjalizować zegar sprzętowy z którym będzie ono współpracować. Zegar ten musi zgłaszać przerwanie o przepełnieniu aby umożliwić wykrycie niekompletnego kodu. Następnie inicjalizujemy urządzenie peryferyjne „Input Capture” do współpracy z przygotowanym wcześniej zegarem. Musi ono zostać skonfigurowane tak aby zgłaszało przerwanie w przypadku wystąpienia zbocza opadającego (patrz Kod źródłowy 2.15 na stronie 34). Wynika to z budowy zastosowanego układu TSOP1138 [6] – jego wyjście OUT przyjmuje stan niski jeżeli obierany jest sygnał. W procedurze obsługi przerwania przełączane jest zbocze na które zostanie zgłoszone kolejne przerwanie (patrz XX na stronie XX). Każde zgłoszenie przerwania powoduje wywołanie funkcji dekodera z nowymi parametrami. Po otrzymaniu z dekodera informacji, że kod został pomyślnie odebrany, wysyłane jest powiadomienie do biblioteki obsługi urządzeń wejściowych (patrz XX na stronie XX).

**Kod źródłowy 2.15.** Inicjalizacja urządzenia „Input Capture” do współpracy z zegarem sprzętowym „Timer2”.

// Configure Timer2 to use with IC2

OpenTimer2(T2\_OFF | T2\_SOURCE\_INT | T2\_PS\_1\_16, 0xffff);

// TODO: dodac priorytet do configa

ConfigIntTimer2(T2\_INT\_ON | T2\_INT\_PRIOR\_3 | T2\_INT\_SUB\_PRIOR\_0);

WriteTimer2(T2\_RELOAD);

// Configure IC2

OpenCapture2(IC\_ON | IC\_TIMER2\_SRC | IC\_EVERY\_FALL\_EDGE);

ConfigIntCapture2(IC\_INT\_OFF | IC\_INT\_PRIOR\_3 |

T2\_INT\_SUB\_PRIOR\_0);

**Kod źródłowy 2.16.** Obsługa przerwania wykrycia zbocza

static void lldPic32IRInterrupt(struct hldIrDevice \*pIrDev,

UINT16 pPulseLen, INT32 \*pHiPriorTaskWoken)

{

WriteTimer2(T2\_RELOAD);

if (ir\_flags.edge)

{

OpenCapture2(IC\_ON | IC\_TIMER2\_SRC | IC\_EVERY\_FALL\_EDGE);

ir\_flags.edge = 0;

}

else

{

OpenCapture2(IC\_ON | IC\_TIMER2\_SRC | IC\_EVERY\_RISE\_EDGE);

ir\_flags.edge = 1;

}

// If this is the first edge. Start timer

if (ir\_flags.first)

IRTimerEnable();

lastCode = pIrDev->config.decode(pPulseLen, &ir\_flags,

xTaskGetTickCountFromISR());

if (ir\_flags.first)

ir\_flags.first = 0;

if (lastCode != 0)

inputRcuEventNotifyISR(EVENT\_RCU\_CODE\_RECEIVED, lastCode,

pHiPriorTaskWoken);

}

Kod źródłowy sterownika odbiornika podczerwieni znajduje się na dołączonej płycie CD pod ścieżką: „*./source/hal/lld/platforms/microchip/pic32IR.c*”.

#### AUDIO

Istotą aplikacji demonstracyjnej przygotowanej na potrzeby pracy jest odtwarzanie plików audio. Implikuje to konieczność obsługi kodera/dekodera audio. Warstwa abstrakcji sprzętu udostępnia strukturę bazową dla tego typu sterowników (patrz Kod źródłowy 2.17 na stronie 34).

**Kod źródłowy 2.17.** Warstwa abstrakcji dla sterowników audio

struct hldAudioDevice

{

struct hldDevice head;

struct hldAudioConfig config;

retcode (\*attach)(struct hldAudioConfig \*pCfg);

retcode (\*open)(struct hldAudioDevice \*pAudioDev);

retcode (\*close)(struct hldAudioDevice \*pAudioDev);

retcode (\*ioctl)(struct hldAudioDevice \*pAudioDev,

UINT32 pCmd, UINT32 pParam);

};

Charakterystycznym elementem tej struktury jest wskaźnik na funkcję ioctl. Nazwa funkcji zaczerpnięta została z warstwy abstrakcji sprzętu systemu Linux. Pełni ona funkcje wydawania urządzeniom poleceń specyficznych dla danego ich typu. W opisywanych wcześniej sterownikach nie było potrzeby implementacji takiej funkcji. W przypadku sterownika audio było to konieczne aby umożliwić wydawanie poleceń takich jak na przykład ustawianie głośności, nie tworząc jednocześnie wielu osobnych specyficznych funkcji do ich obsługi. Sterownik audio powinien implementować następujące polecenia:

1. AC\_SET\_VOLUME – ustawienie poziomu głośności
2. AC\_SET\_VOLUME\_SOFT – ustawienie poziomu głośności z synchronizacją przejścia przez zero
3. AC\_GET\_VOLUME – pobranie aktualnej głośności
4. AC\_DISABLE – wyłączenie wzmacniacza wyjściowego
5. AC\_ENABLE – włączenie wzmacniacza wyjściowego
6. AC\_SET\_SAMPLE – ustawienie częstotliwości próbkowania
7. AC\_SET\_BITS – ustawienie ilości bitów na kanał
8. AC\_SET\_CHANNELS – ustawienie ilości kanałów

Przekazywane są one do funkcji ioctl jako drugi parametr. Trzecim parametrem jest ustawiana wartość lub wskaźnik na miejsce w pamięci gdzie zostanie zapisana wartość pobrana z urządzenia (np. odczyt aktualnej głośności).

Każdy sterownik audio powinien implementować możliwość współpracy ze wszystkimi trybami pracy biblioteki audio. Dostępne są następujące tryby:

1. AM\_NONE – tryb testowy (zawsze wysyłana jest zerowa próbka)
2. AM\_KMIXER – niezaimplementowany tryb miksera audio
3. AM\_SINE – tryb testowy (generator fali sinusoidalnej o określonej częstotliwości)
4. AM\_ONECHANNEL – odgrywany jest dźwięk z jednego pliku muzycznego

Domyślnym trybem jest AM\_ONECHANNEL, pozwala od odtwarzać jednocześnie jeden dźwięk. Nie ma możliwości miksowania kilku dźwięków razem. Do tego celu został przygotowany tryb AM\_KMIXER który niestety nie został jeszcze zaimplementowany po stronie biblioteki audio. Ma on na celu, umożliwienie odtwarzania kilku dźwięków na raz oraz dynamiczne dopasowanie częstotliwości próbkowania danego pliku muzycznego do ustawień urządzenia audio, poprzez decymację lub aproksymację próbek.

Struktura konfiguracyjna dla tego typu sterowników zawiera priorytet przerwania urządzenia, tryb pracy, informację o tym które moduły urządzenia mają zostać włączone oraz początkowe ustawienia częstotliwości próbkowania, ilości kanałów i ilości bitów na próbkę. Jeżeli wybrany jest tryb AM\_ONECHANNEL wartości początkowe ulegają zmianie po odtworzeniu pierwszego dźwięku. Urządzenie jest wtedy ustawiane na takie wartości jakie zostaną odczytane z nagłówka pliku dźwiękowego.

Układ scalony WM8731 zastosowany w projekcie jest konfigurowany przez interfejs I2C. Do komunikacji wykorzystany został sprzętowy moduł I2C procesora PIC32. Kolejne próbki wysyłane są do urządzenia przez interfejs SPI który może pracować zarówno w trybie master jak i slave. Jeżeli układ WM8731 pracuje jako master, zgłasza on potrzebę uzyskania wartości kolejnej próbki w odpowiednich odstępach czasowych. W trybie tym wystarczy w obsłudze przerwania od SPI w procesorze odesłać kolejną próbkę odczytaną z bufora audio. W przypadku pracy układu WM8731 w trybie slave, będziemy musieli pilnować interwałów czasowych w których podawane są kolejne próbki. Zastosowany został tryb pierwszy. Jako, że przy częstotliwości próbkowania równej 48kHz, przerwanie żądania kolejnej próbki występuje 48 tysięcy razy na sekundę, ważna jest maksymalna optymalizacja procedury jego obsługi. Po stronie sterownika leży jedynie sprawdzenie w jakim trybie pracuje i wykonanie odpowiedniej, zależnej od tego trybu, funkcji bibliotecznej, pobierającej kolejną próbkę dźwięku. Dzięki temu, że funkcje biblioteczne są gotowe i zostały wcześniej zoptymalizowane, programista sterownika nie musi się o to martwić.

Jako że, źródła sterownika obejmują ponad 500 linijek kodu, nie ma możliwości zamieszczenia pełnego kodu źródłowego w pracy. W przypadku sterownika audio zamieszczanie fragmentów nie pozwoli zrozumieć istoty jego działania. Kod źródłowy dostępny jest na załączonej w dodatku A płycie CD pod ścieżką: „*./source/hal/lld/platforms/microchip/wm8731.c*”.

#### DISK

Aby sterownik audio posiadał dane do odtwarzania, musimy mieć źródło tych danych. W naszym projekcie jest to karta SD. Warstwa abstrakcji sprzętu została przygotowana pod dowolne urządzenie pamięci masowej. Nowa funkcja (w stosunku do omówionych do tej pory sterowników) która pojawiła się w strukturze bazowej dla tego typu sterowników to funkcja status (patrz XX na stronie XX). Pobiera ona status urządzenia, możliwe są trzy stany:

1. STA\_NOINIT - sterownik nie zainicjalizowany
2. STA\_NODISK – brak nośnika
3. STA\_PROTECT – zabezpieczenie przed zapisem włączone

**Kod źródłowy 2.18.** Warstwa abstrakcji dla sterowników dysków

struct hldDiskDevice

{

struct hldDevice head;

retcode (\*attach)();

diskstatus (\*open)(struct hldDiskDevice \*pDiskDev);

retcode (\*close)(struct hldDiskDevice \*pDiskDev);

retcode (\*read)(struct hldDiskDevice \*pDiskDev, UINT8 \*pBuf,

UINT32 pSector, UINT8 pCount);

retcode (\*write)(struct hldDiskDevice \*pDiskDev,

const UINT8 \*pBuf, UINT32 pSector, UINT8 pCount);

retcode (\*ioctl)(struct hldDiskDevice \*pDiskDev,

UINT32 pCmd, UINT32 \*pParam);

diskstatus (\*status)(struct hldDiskDevice \*pDiskDev);

};

Funkcje read oraz write przyjmują parametry specyficzne dla pamięci masowych. Poza wskaźnikiem na bufor danych, funkcje przyjmują parametry pSector oraz pCount. Pierwszy z nich określa sektor z którego będziemy czytać lub do którego będziemy pisać. Drugi natomiast mówi ile bajtów chcemy odczytać lub zapisać.

W przypadku sterownika dysku istnieje, podobnie jak w przypadku sterownika audio, funkcja ioctl. Każdy sterownik dysku musi wspierać wymienione poniżej komendy:

1. CTRL\_SYNC – sprawdzenie czy zlecony zapis został zakończony
2. GET\_SECTOR\_SIZE – pobranie wielkości sektora danego dysku. Dla kart SD jest to najczęściej wartość 512.
3. GET\_SECTOR\_COUNT – pobranie ilości sektorów dysku
4. GET\_BLOCK\_SIZE – pobranie wielkości bloku czyszczenia karty (sektory jako jednostka)
5. CTRL\_ERASE\_SECTOR – czyszczenie części pamięci, sektor początkowy oraz końcowy podane są kolejno jako starsze i młodsze słowo (16 bitów) parametru.

Sterowniki określonych typów dysków, jak na przykład karta SD, mogą wspierać dodatkowe polecenia. W przypadku karty SD może to być na przykład pobranie numeru identyfikatora karty (CID).

W projekcie zastosowany został gotowy sterownik do karty SD wykonany przez Aidena Morrisona a przystosowany do pracy na procesorze PIC32MX795L512 przez Riccardo Leonardiego. Komunikuje się on z kartą pamięci przy pomocy interfejsu SPI. Niestety sterownik ten nie wykorzystuje kontrolera DMA do kopiowania danych pomiędzy kartą SD a pamięcią operacyjną. W planach rozwojowych projektu jest dodanie wsparcia dla kontrolera DMA.

Kod źródłowy sterownika znajduje się na dołączonej w dodatku A płycie SD pod ścieżką: „*./source/lld/platforms/microchip/pic32SDMMC.c*”.

#### LCD

Warstwa abstrakcji sprzętu dla sterowników wyświetlaczy oraz układów graficznych posiada znacznie więcej funkcji, aniżeli omawiane do tej pory sterowniki. Znajdziemy tutaj funkcje, odpowiadające specyficznym możliwością układów graficznych. W obecnym stopniu zaawansowania projektu, są to funkcje służące do rysowania podstawowych elementów takich jak prostokąt czy piksel (patrz Kod źródłowy 2.18 na stronie 37). Funkcje te będą przybywały razem z rozwojem, projektowanego równolegle, układu graficznego FPGA. Wszystkie funkcje których nie wspiera układ graficzny a które znajdują się w strukturze bazowej powinny być emulowane programowo przez sterownik.

**Kod źródłowy 2.19.** Warstwa abstrakcji dla sterowników układów graficznych

struct hldLcdDevice

{

struct hldDevice head;

UINT32 drawingColor; // Drawing color in color format of LCD

retcode (\*attach)();

retcode (\*open)(struct hldLcdDevice \*pLcdDev);

retcode (\*close)(struct hldLcdDevice \*pLcdDev);

retcode (\*setColor)(struct hldLcdDevice \*pLcdDev, UINT8 pA,

UINT8 pR, UINT8 pG, UINT8 pB);

retcode (\*drawPixel)(struct hldLcdDevice \*pLcdDev,

UINT16 pX, UINT16 pY);

UINT16 (\*getPixel)(struct hldLcdDevice \*pLcdDev,

UINT16 pX, UINT16 pY);

retcode (\*fill)(struct hldLcdDevice \*pLcdDev,

UINT16 pX1, UINT16 pY1, UINT16 pX2, UINT16 pY2);

UINT32 (\*getMaxX)();

UINT32 (\*getMaxY)();

};

Funkcja setColor ustawia aktualny kolor którym będą rysowane elementy wysłane do układu graficznego po jej wywołaniu. Przyjmuje ona cztery parametry, są to składowe koloru w formacie ARGB888. Sterownik musi zadbać o to aby kolor został zmieniony na format używany przez układ graficzny.

Kolejna funkcja nosi nazwę drawPixel i odpowiedzialna jest za narysowanie na ekranie pojedynczego piksela o ustalonym przez funkcję setColor kolorze. Jako parametry przyjmuje współrzędne pX, pY piksela.

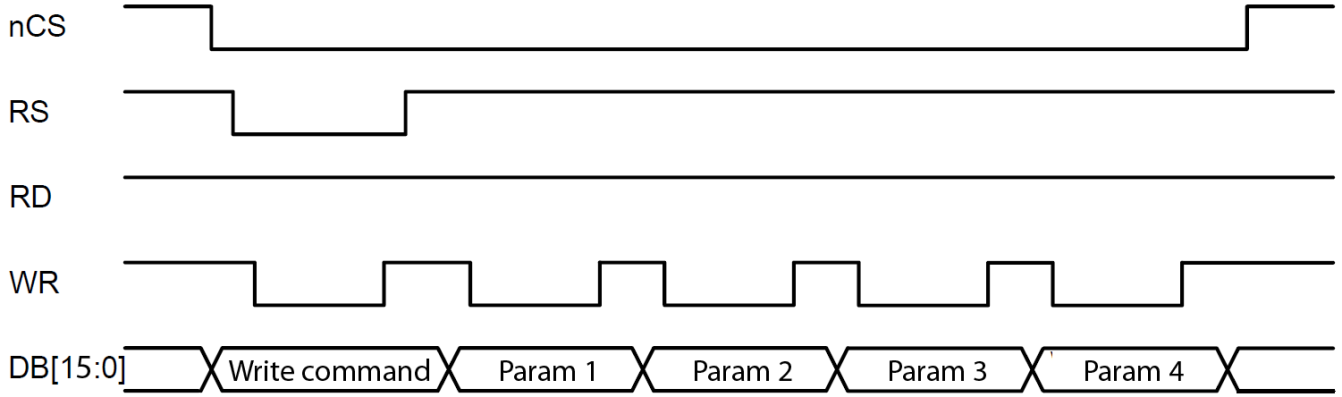
Funkcja getPixel pobiera kolor piksela o współrzędnych pX, pY.

Funkcja fill wypełnia zadany prostokąt aktualnym kolorem. Prostokąt ten definiujemy poprzez podanie współrzędnych lewego górnego (pX1, pY1) oraz prawego dolnego (pX2, pY2) wierzchołka.

Ostatnimi elementami są wskaźniki na funkcje getMaxX oraz getMaxY. Pobierają one maksymalne pozycje pikseli dla współrzędnych x oraz y. Jeżeli sterownik wspiera różne orientacje wyświetlacza (pionowa/pozioma) funkcje te powinny to uwzględniać.

Na potrzeby projektu omawianego w tej pracy zaimplementowane zostały dwa sterowniki układów graficznych. Pierwszym z nich jest sterownik do układu HX8347 [5] który został wbudowany w standardowy wyświetlacz zamontowany na płycie ewaluacyjnej. Drugim z nich jest, ciągle rozwijany, sterownik do układu graficznego FPGA z którym nasze urządzenie ma współpracować.

Komunikacja z oby dwoma układami nawiązywana jest przez 16-bitowy interfejs i80 (patrz Rysunek 2.3 na stornie 40). Transmisja inicjowana jest poprzez wymuszenie przez procesor stanu niskiego na jednym z portów WR lub RD (zależnie od kierunku transmisji). Jeżeli chcemy wysłać polecenie lub dane do układu graficznego, na porcie WR musi zostać wymuszony stan niski, w przeciwnym kierunku wymuszamy stan niski na porcie RD. Za określenie czy przesyłany jest kod komendy czy dane (np. parametry) odpowiada port RS. Każdy z układów dodatkowo wspiera sygnał CS dzięki któremu możemy wybrać do którego z nich wysyłamy polecenie. Oba układy pracują na tej samej 16-bitowej szynie danych.



**Rysunek 2.3.** 16-bitowy interfejs komunikacyjny i80

Źródło: na podstawie [5]

Od strony procesora za komunikację odpowiedzialny będzie wbudowany moduł PMP. Niestety z uwagi na to, że używamy 16-bitowej magistrali sygnał CS będziemy musieli zmieniać programowo. W procesorach PIC32 za obsługę sygnału CS odpowiadają porty DB[15:14] jeżeli nie są używane jako linie danych. Sygnał RS również będziemy przełączać programowo.

Dzięki zastosowaniu sygnałów CS istnieje teoretyczna możliwość obsługiwania obu wyświetlaczy jednocześnie. Wymagało by to przygotowania osobnego sterownika dla urządzenia PMP, zapewniającego synchronizację dostępu z wielu wątków oraz przerobienia sterowników od układów graficznych tak, aby zamiast bezpośrednio odwoływać się do rejestrów procesora, używały właśnie tego sterownika. Zastosowanie dwóch wyświetlaczy pracujących jednocześnie miało by na celu wykorzystanie mniejszego, jako „gładzik” z możliwością wyświetlania na nim dodatkowych informacji. Mogły by to być na przykład niestandardowe przyciski wymagane przez aplikację lub wyświetlanie dodatkowych informacji np. o aktualnie odtwarzanym utworze. W przypadku zastosowania urządzenia w samochodzie dodatkowy wyświetlacz mógłby znaleźć się na przykład na kierownicy.

Sterownik dla układu HX8347 został napisany częściowo na podstawie sterownika z biblioteki graficznej firmy „Microchip”. Zaczerpnięte zostały jedynie polecenia inicjalizujące wyświetlacz. Pozostała część została napisana na podstawie noty katalogowej układu [5]. Podobnie jak w przypadku sterownika audio, kod źródłowy jest zbyt obszerny aby mógł zostać szczegółowo omówiony w pracy. Znajduje się od na dołączonej płycie CD pod ścieżką „*./source/lld/platforms/microchip/hx8347.c*”.

Więcej uwagi chciałbym poświęcić sterownikowi do układu graficznego FPGA. Jako że układ graficzny projektowany był przez mojego kolegę, mieliśmy całkowity wpływ na to jakie polecenia i w jaki sposób będzie on obsługiwał. Kody operacyjne (ang. opcode) obsługiwanych poleceń zostały ustalone na podstawie kodów ASCII pierwszych znaków nazwy komendy. Na przykład polecenie „Set Pixel” posiada kod 0x7370 co odpowiada znakom „sc” w kodzie ASCII. Tabela 2.2 przedstawia pełny zestaw poleceń. Polecenia zaznaczone kolorem czerwonym, nie zostały jeszcze zaimplementowane w układzie graficznym, co uniemożliwia uruchomienie w pełni funkcjonalnej aplikacji korzystającej z tego układu. Zarówno polecenia jak i każdy z parametrów są wartościami 16-bitowymi.

**Tabela 2.2.** Lista poleceń układu graficznego FPGA

Źródło: Opracowanie wspólne: D. Szot, B. Zamolski

|  |  |
| --- | --- |
| Polecenie | Opis |
| **Set Color („sc”) – 0x7363**  Ilość parametrów: 1 | Ustawia aktualny kolor. Kolorem tym będą rysowane kolejne elementy. Odpowiednik funkcji setColor z sterownika.  **Parametr 1:** Kolor w formacie RGB565 |
| **Set Pixel („sp”) – 0x7370**  Ilość parametrów: 2 | Rysuje pojedynczy piksel aktualnie wybranym kolorem. Odpowiednik funkcji drawPixel z sterownika.  **Parametr 1:** Pozycja X piksela  **Parametr 2:** Pozycja Y piksela |
| **Fill Rect („fr”) – 0x6672**  Ilość parametrów: 4 | Wypełnia aktualnym kolorem prostokąt pomiędzy punktami o współrzędnych podanych jako parametry. Odpowiednik funkcji fill z sterownika.    **Parametr 1:** X1  **Parametr 2:** Y1  **Parametr 3:** X2  **Parametr 4:** Y2 |
| **Clear Screen (“cs”) – 0x6373**  Ilość parametrów: 0 | Wypełnia cały ekran aktualnym kolorem. Brak odpowiednika w sterowniku. Może on wykrywać w funkcji fill że polecenie dotyczy obszaru całego ekranu i wysłać polecenie Clear Screen zamiast Fill Rect. |
| **Color Expansion („ce”) – 0x6365**  Ilość parametrów: | Rozszerza podaną 2-bitową mapę monochromatyczną na aktualny kolor. Wartość konkretnego piksela w mapie określa jego przezroczystość po operacji. Funkcja używana do rysowania czcionek bitmapowych z wygładzaniem krawędzi.    **Parametr 1:** Pozycja X  **Parametr 2:** Pozycja Y  **Parametr 3:** Szerokość mapy (**w**)  **Parametr 4:** Wysokość mapy (**h**)  **Parametry od 5 do :** Określają wartości kolejnych pikseli mapy bitowej. Po 8 pikseli na parametr. |
| **Flush („ff”) – 0x6666**  Ilość parametrów: 0 | W przypadku włączonego podwójnego buforowania polecenie zmienia wyświetlany bufor. |

W sterowniku zaimplementowane zostały funkcje ułatwiające obsługę interfejsu komunikacji. Są to funkcje ustawiające stany na portach RS oraz CS. Kod źródłowy 2.20 przedstawia funkcję rysującą piksel. W komentarzach zawarty został opis poszczególnych poleceń.

**Kod źródłowy 2.20.** Rysowanie piksela przez układ graficzny FPGA

static retcode lldFpgaGpuDrawPixel(struct hldLcdDevice \*pLcdDev,

UINT16 pX, UINT16 pY)

{

lldFpgaGpuSetCS(); // CS = 0

lldFpgaGpuSetCommand(); // RS = 0

lldFpgaGpuWrite(0x7370);// Ustawia 0x7370 na DB po czym RW = 0

lldFpgaGpuSetData(); // RS = 1

lldFpgaGpuWrite(pX);

lldFpgaGpuWrite(pY);

lldFpgaGpuRstCS(); // CS = 1

return SUCCESS;

}

Pełny kod źródłowy sterownika układu graficznego FPGA znajduje się na dołączonej płycie CD pod ścieżką „*./source/lld/platforms/microchip/fpgaGPU.c*”.

#### TOUCH

Ostatnim już sterownikiem który został przygotowany na potrzeby pracy jest sterownik panelu dotykowego. Jest to jedyny w projekcie sterownik nieuzależniony od procesora. Zawdzięczamy to posiadaniu osobnego sterownika do przetwornika analogowo cyfrowego który jest wykorzystywany poprzez warstwę abstrakcji przez panel dotykowy.

Warstwa abstrakcji dla panelu dotykowego posiada (poza standardowymi) jedną funkcję służącą do odczytania aktualnej pozycji rysika (patrz XX na stronie XX). Jeżeli rysik nie dotyka ekranu powinna ona zwrócić wartości „-1” zarówno dla współrzędnej X jak i Y. Funkcja ta jest wykorzystywana w systemie jedynie jeżeli chcemy pobrać pozycję rysika omijając warstwę menadżera wejść (patrz 2.4.4 na stronie 49). Ponieważ urządzenie panelu dotykowego jest urządzeniem wejściowym, jego sterownik powinien powiadamiać menadżera wejść o pojawieniu się pewnego zdarzenia (np. wciśnięcia lub przesunięcia rysika). To właśnie w ten sposób zdarzenia są przekazywane do kolejnych warstw systemu a szczególności do biblioteki graficznego interfejsu użytkownika.

**Kod źródłowy 2.21.** Warstwa abstrakcji sterownika panelu dotykowego

struct hldTouchDevice

{

struct hldDevice head;

struct hldTouchConfig config;

retcode (\*attach)(struct hldTouchConfig \*pCfg);

retcode (\*open)(struct hldTouchDevice \*pTouchDev);

retcode (\*close)(struct hldTouchDevice \*pTouchDev);

retcode (\*read)(struct hldTouchDevice \*pTouchDev,

INT32 \*pX, INT32 \*pY);

};

Poprzez strukturę konfiguracyjną podajemy takie wartości jak parametry kalibracji, rozdzielczość ekranu, priorytet wątku sterownika oraz okres czasu co jaki pobierana jest kolejna próbka.

Sterownik zaimplementowany na potrzeby projektu został przygotowany dla 4-pinowych paneli rezystancyjnych. Wymusza on zdefiniowanie w pliku „*./board/XXX/board.h*” kanałów przetwornika analogowo cyfrowego do których zostały podłączone linie X+, Y+, Y- oraz portów procesora dla wszystkich czterech linii panelu.

Dane odczytane z kanałów przetwornika są dodatkowo filtrowane przez prosty cyfrowy filtr uśredniający (ścieżka do źródeł filtru: „*./source/lib/digitalFilter.c*”).

Aby sterownik mógł powiadamiać menadżera wejść o zaistnieniu zdarzeń wygenerowanych przez panel dotykowy musi on sprawdzać co pewien czas czy żadne z nich nie wystąpiło. W przypadku naszego projektu czas ten został doświadczalnie ustawiony na 18 ms. Jako że panel dotykowy wymaga multipleksowania stanów na portach oraz odczytywaniu odpowiedniego kanału analogowego w odpowiedniej chwili, stworzona została prosta maszyna stanów. Możliwe stany oraz wykonywane w nich czynności zostały przedstawione na poniższej liście:

1. SET\_X – Ustaw stan niski na porcie X- oraz stan wysoki na X+.
2. GET\_X – Pobierz wartość z przetwornika ADC z portu Y+. Jeżeli pobrana wartość wskazuje, że rysik został właśnie zabrany z panelu dotykowego, przejdź do kroku GENERATE\_EVENT.
3. SET\_Y – Ustaw stan niski na porcie Y- oraz stan wysoki na Y+.
4. GET\_Y – Pobierz wartość z przetwornika ADC z portu X+. Analogicznie do kroku SET\_X, przejdź do kroku GENERATE\_EVENT jeżeli to konieczne.
5. SET\_VALUES – Dodaj próbkę z pobranymi wartościami do filtru.
6. GENERATE\_EVENT – Jeżeli do filtru zostało dodane tyle próbek ile ma szerokość okna filtru, lub zostało wykryte, że rysik został zabrany – wygeneruj odpowiednie zdarzenie i powiadom o nim menadżera wejść.

Stany zmieniane są co czasu podanego w konfiguracji sterownika.

Pełny kod źródłowy sterownika panelu dotykowego znajduje się na dołączonej płycie CD pod ścieżką „*./source/lld/resistiveTouch.c*”.

## Biblioteki

Ze względu na znaczą objętość oraz stopień skomplikowania kodu bibliotek systemowych, zamiast opisywać w tym rozdziale ich wewnętrzną budowę oraz sposób działania, chciałbym skupić się na sposobie ich wykorzystywania. Rozdział został napisany w formie dokumentacji dla programisty piszącego aplikację użytkową. Najlepszym sposobem na poznanie mechanizmów rządzących bibliotekami jest lektura ich kodu źródłowego. Na tyle na ile było to możliwe, pisząc kod starałem się komentować najważniejsze jego fragmenty. Kody źródłowe bibliotek znajdują się na dołączonej płycie CD w katalogu „*./source/lib/*”.

### Biblioteka standardowa

W rozumieniu biblioteki standardowej jako biblioteka standardowa danego języka programowania, użyta została biblioteka przygotowana przez producenta zastosowanego procesora. Producenci procesorów, zazwyczaj dostarczają zestawy podstawowych funkcji bibliotecznych przystosowanych do ich produktów, więc nawet w przypadku chęci przeniesienia systemu na inny procesor nie stanowiło by to problemu.

Jako bibliotekę standardową możemy również uznać moduły czy funkcje systemowe ogólnego zastosowania. Wykorzystywany w sterowniku od panelu dotykowego filtr cyfrowy jest jednym z takich modułów. Filtr może pracować jedynie na liczbach całkowitych.

#### Filtr cyfrowy

Do stworzenia nowego filtru wykorzystujemy funkcję filterCreate. Zwraca ona wskaźnik na strukturę opisującą filtr. Jedynym parametrem który przyjmuje jest długość okna filtru – czyli ilość próbek na podstawie której liczona będzie wartość średnia. Po zakończeniu korzystania z filtru, możemy go usunąć wywołując funkcję filterDelete jako parametr podając wskaźnik na filtr do usunięcia.

Próbki do filtru dodajemy używając polecenia filterAddSample, pierwszy parametr to wskaźnik na filtr, drugi jest wartością dodawanej próbki. W celu usunięcia wszystkich próbek z filtru wywołujemy funkcję filterResetSamples.

Po dodaniu do filtru pierwszej próbki, pozostałe znajdujące się w oknie próbki mają wartość zerową. Uśredniona wartość będzie więc znacznie mniejsza niż wartość próbki. W niektórych zastosowaniach jest to niedopuszczalne. W przypadku panelu dotykowego, zachowanie takie powodowało by, że każde dotknięcie było by interpretowane, jako przeciągnięcie rysikiem po ekranie, od rogu do rzeczywistej pozycji rysika. Dlatego przygotowana została funkcja umożliwiająca sprawdzenie czy całe okno filtru zostało już zapełnione próbkami. Funkcja ta nosi nazwę filterIsReady i zwraca wartość TRUE jeżeli ilość dodanych próbek od ostatniego wyzerowania filtru jest większa lub równa długości okna.

Istnieje możliwość pobrania maksymalnej oraz minimalnej wartości próbki która znajduje się w oknie filtru. Służą do tego polecenia filterGetMax oraz filterGetMin.

Mamy dwie możliwości pobrania uśrednionej wartości uśrednionej. Jedna z nich jest zwykłą średnią arytmetyczną próbek znajdujących się w oknie filtru (filterGetValue) druga natomiast liczy również średnią arytmetyczną, ale z pominięciem wartości najbardziej odstających od reszty (filterGetRejectMinMax). Przykład użycia filtru prezentuje Kod źródłowy 2.22 na stronie 46.

**Kod źródłowy 2.22**. Przykład użycia filtru cyfrowego

struct digitalFilter \*filter = filterCreate(3);

filterAddSample(filter, 1);

filterAddSample(filter, 2);

filterAddSample(filter, 6);

if (filterIsReady(filter))

{

INT32 a = filterGetValue(filter); // a = 3

INT32 b = filterGetRejectMinMax(filter); // b = 2

filterResetSamples(filter);

}

filterDelete(filter);

Kod źródłowy filtru cyfrowego znajduje się na dołączonej płycie CD pod ścieżką: „*./source/lib/digitalFilter.c*”.

#### LOG

Moduł o nazwie „LOG” służy do umożliwienia śledzenia wykonywania kodu poprzez dowolne urządzenie znakowe. Udostępnia on takie polecania jak LOG, ERROR, DONE, oraz DEBUG. Polecenia te przyjmują składnie klasycznej funkcji „printf”, znanej z biblioteki standardowej języka C. O ile trzy pierwsze polecenia różnią się jedynie dodawanym przed komunikatem przedrostkiem to komunikaty wysłane przy pomocy polecenia DEBUG będą wyświetlane tylko wtedy jeżeli program zostanie skompilowany w konfiguracji do debugowania. Jedynym obowiązkiem programisty koniecznym aby skorzystać z tych poleceń, jest ustawienie urządzenia do którego mają być wysyłane komunikaty (patrz Kod źródłowy 2.23 na stronie 47). Podczas projektowania modułu LOG został popełniony pewien błąd. Jeżeli zlecimy sterownikowi portu szeregowego wysłanie znaku, dodaje on go do kolejki, po czym od razu kontynuuje wykonywanie programu nie czekając aż znak zostanie wysłany. Jeżeli używamy tak działającego urządzenia w celu śledzenia wykonywania programu to w przypadku wystąpienia błędu powodującego ponowne uruchomienie urządzenia, możemy nie zobaczyć ostatniej linii która powinna się pojawić na ekranie przed ponownym uruchomieniem. Sytuacja ta utrudnia wyszukiwanie błędów. Jest to znany problem i zostanie poprawiony w przyszłych wersjach oprogramowania.

**Kod źródłowy 2.23.** Użycie modułu LOG

struct hldUartDevice \*uart;

uart = hldDeviceGetById(HLD\_DEVICE\_TYPE\_UART, 0);

if (uart != NULL)

{

logSetUartDevice(uart);

DONE("LOG device set: %p", uart);

}

UINT32 a = 1024;

LOG("%d is equal 0x%x in hex", a, a);

Kod źródłowy modułu LOG znajduje się na dołączonej płycie CD pod ścieżką: „*./source/lib/log.c*”.

### Konsola

Na samym początku tworzenia projektu, kiedy nie istniał jeszcze graficzny interfejs użytkownika ani nie było możliwości obsługi urządzenia za pomocą pilota, a konieczne było wysyłanie pewnych poleceń do urządzenia, powstała prosta konsola. Działa ona za pośrednictwem portu transmisji szeregowej RS232 i pozwala na zdalne wykonywanie określonych funkcji.

Aby zachować zasady enkapsulacji, konieczne było stworzenie interfejsu przez który poszczególne moduły będą rejestrowały niektóre ze swoich funkcji jako możliwe do wykonania z poziomu konsoli. Interfejsem tym jest funkcja consoleRegisterCmd. Jako pierwszy parametr przyjmuje ona nazwę polecenia pod którym dana funkcja będzie dostępna z poziomu konsoli. Drugim parametrem jest wskaźnik na funkcję która ma zostać wykonana w przypadku wpisania przez użytkownika tego polecenia. Funkcja ta ma specyficzną listę parametrów (patrz Kod źródłowy 2.24 na stronie 48). Są to dokładnie takie parametry jakie przyjmuje główna funkcja programu (main) w systemie linux.

**Kod źródłowy 2.24.** Przykładowa funkcja konsolowa

void consoleHellowWorld(UINT8 argc, const char \*argv[])

{

LOG("Hellow World! You give me %d params.", argc);

}

Parametr argc przechowuje ilość parametrów z którymi zostało wywołane polecenie z konsoli. Drugi parametr (argv) jest tablicą w której przechowywane są, w postaci tekstowej, parametry z jakimi zostało wywołane polecenie. Kod źródłowy 2.25 przedstawia sposób zarejestrowania tego typu funkcji w konsoli. Rysunek 2.4 przedstawia zrzut ekranu z konsoli po wykonaniu funkcji.

**Kod źródłowy 2.25.** Przykład rejestracji funkcji w konsoli

consoleRegisterCmd("hellowworld", &consoleHellowWorld);

!!!!!!!!!!!!!!OBRAZEK!!!!!!!!!!!!!!!!

**Rysunek 2.4.** Zrzut ekranu konsoli, po wykonaniu polecenia „hellowworld”

Źródło: Opracowanie własne

Kod źródłowy konsoli znajduje się na dołączonej płycie CD w katalogu: „*./source/lib/console/*”.

### System plików

Wcześniej zostało powiedziane że do obsługi systemu plików posłuży nam biblioteka „*FatFS*” [9]. Nie jest ona zgodne ze standardem POSIX. Polecania mają bardzo podobne nazwy, aczkolwiek w „*FatFS*” po znaku „f” w nazwie każdej funkcji występuje podkreślnik np. f\_open, f\_close, f\_read.

Dodatkowo różnią się niektóre parametry funkcji. Jest tak w przypadku funkcji f\_open gdzie zamiast podać jako ostatni parametr ciąg znaków określający tryb otwarcia pliku (wersja wg. POSIX) podajemy flagi bitowe.

Szczegółowy opis funkcji biblioteki „*FatFS*” znajduje się w pozycji [9] bibliografii.

### Audio

Wcześniej w pracy zostało wspomniane, że biblioteka audio posiada kilka trybów pracy. Aktualnie zaimplementowane tryby to generowanie sygnału sinusoidalnego oraz odtwarzanie pojedynczego dźwięku. Uwagę chciałbym poświęcić drugiemu z tych trybów nazywanego dalej 1CH (z ang. 1 channel).

Z punktu widzenia programisty piszącego aplikację, obsługa biblioteki audio jest bardzo prosta. Ogranicza się do wydawania poleceń typu odtwórz, pauza, stop. Po odtworzeniu dźwięku nie musimy pobierać z biblioteki żadnego wskaźnika lub uchwytu dźwięku, z tego względu, iż istnieje możliwość odtwarzania wyłączenie jednego dźwięku w danej chwili. Jeżeli spróbujemy odtworzyć dźwięk, bez zatrzymywania aktualnie odtwarzanego, zostanie on zatrzymany automatycznie.

Do odtworzenia dźwięku wykorzystujemy funkcję audio1chPlaySound. Przyjmuje ona dwa parametry. Pierwszym z nich jest nazwa pliku który ma zostać odtworzony. Kolejnym parametrem jest tryb odtwarzania. Istnieją dwa tryby: synchroniczny (SND\_SYNC) oraz asynchroniczny (SND\_ASYNC). Tryb synchroniczny wykorzystuje wątek, z którego funkcja została wywołana. Co za tym idzie wykonywanie programu zostaje zawieszone, do chwili zakończenia odtwarzania dźwięku. Rozwiązanie takie czasami może być pożądane, jednak w przypadku odtwarzacza audio jest niedopuszczalne. Dlatego istnieje drugi (asynchroniczny) tryb odtwarzania. W chwili zlecenia odtwarzania dźwięku w tym trybie, biblioteka audio automatycznie tworzy nowy wątek, który będzie pobierał kolejne próbki pliku z urządzenia pamięci masowej, po czym umieszczał je w buforze odtwarzania. Po zakończeniu odtwarzania, wątek ten jest automatycznie usuwany.

Tryb „*1CH*” biblioteki udostępnia również funkcje, służące do kontrolowania odtwarzania. Funkcja audio1chPause, odpowiada za tymczasowe zatrzymanie odtwarzania. Przyjmuje ona parametr typu BOOL określający czy odtwarzanie dźwięku ma zostać zatrzymane (TRUE) czy wznowione (FALSE). W celu całkowitego zatrzymania odtwarzania dźwięku, wywołujemy funkcję audio1chStopSound. Zwalnia ona wszystkie wykorzystywane zasoby, włącznie z usunięciem wątku odtwarzającego (tylko w przypadku trybu asynchronicznego).

Przygotowany został również zestaw trzech funkcji służących do zmiany aktualnej pozycji w pliku. Mogą one zostać wykorzystane do implementacji np. przewijania. Są to funkcje:

1. audio1chGetSamplesCount – pobranie ilości próbek
2. audio1chGetCurrentSample – pobranie numeru aktualnej próbki
3. audio1chSetCurrentSample – skok do podanej próbki

Wartości zwracane przez te funkcje, zależne są od dekodera danego formatu muzycznego i wcale nie muszą zwracać konkretnych numerów próbek. Główne założenie jest takie aby polecenie (audio1chGetCurrentSample()/audio1chGetSamplesCount())\*100 zwracało procentowy postęp odtwarzania.

Do pobierania informacji na temat aktualnie odtwarzanego pliku, używane są funkcje:

1. audio1chGetBits – ilość bitów na próbkę
2. audio1chGetChannels – ilość kanałów
3. audio1chGetSampleRate – częstotliwość próbkowania

**Kod źródłowy 2.26.** Przykład wykorzystania biblioteki audio

UINT32 sr, ch, b;

// Odtwórz dźwięk snd.wav (tryb asynchorniczny)

if (audio1chPlaySound("snd.wav", SND\_ASYNC) == SUCCESS)

{

// Pobierz informacje o dźwięku

sr = audio1chGetSampleRate();

ch = audio1chGetChannels();

b = audio1chGetBits();

LOG("Playing: %dkHz, %dch, %dbit", sr/1000, ch, b);

vTaskDelay(1000\*10); // Czekaj 10s

audio1chStopSound(); // Zakończ odtwarzanie

}

Z punktu widzenia aplikacji przedstawione zostały już wszystkie możliwe do wykorzystania funkcje. Istnieje jeszcze jedna bardzo ważna funkcja biblioteki audio, wykorzystywana przez sterowniki audio. Służy ona do pobrania kolejnej próbki sygnału która ma zostać przekazana do przetwornika cyfrowo analogowego. Funkcja ta nosi nazwę audio1chGetSample. Pobiera ona kolejną próbkę z bufora, przygotowanego wcześniej przez wątek odtwarzający. Dzięki zastosowaniu podwójnego buforowania, podczas pobierania próbek z jednego z nich, wątek odtwarzający może ładować nowe próbki do drugiego. Zastosowanie podwójnego buforowania powoduje opóźnienie odegrania pierwszej próbki w stosunku do zlecenia odtwarzania, o czas ładowania kompletu próbek, do jednego nich. W przypadku zastosowania bufora cyklicznego (ang. circular buffer) można to opóźnienie zredukować. Dla projektu odtwarzacza audio do zastosowań ogólnych, czyli przede wszystkim rozrywki – opóźnienie rzędu mikrosekund nie ma najmniejszego znaczenia. Gdyby urządzenie miało być wykorzystywane do zastosowań profesjonalnych, zmiana bufora na cykliczny mogła by okazać się konieczna. Podwójne buforowanie zastosowane zostało ze względu na wydajność. Aby dodać próbkę do tego typu bufora, wymagane jest wykonanie mniejszej ilości operacji niż w przypadku bufora cyklicznego.

Aktualnie zaimplementowane zostało odtwarzanie jedynie plików „*WAV*”. W założeniach, urządzenie miało odtwarzać również pliki w formacie „*FLAC*”. Z powodu braku czasu nie udało się jednak zrealizować tego założenia. Funkcjonalność ta jest oczywiście nadal zapisana w planach rozwojowych. Przygotowanie dekoderów nowych formatów plików wymaga minimalnej ingerencji w kod biblioteki. Został przygotowany pewien schemat funkcji które powinien implementować dekoder aby mógł zostać wykorzystany przez naszą bibliotekę.

Kod źródłowy biblioteki audio znajduje się na dołączonej płycie CD w katalogu: „*./source/lib/audio/*”.

### Menadżer wejść

Biblioteka menadżera wejść (w kodzie nazwana po prostu „*input*”) odpowiedzialna jest za przechwytywanie zdarzeń (ang. event) od urządzeń wejściowych, kolejkowanie ich, konwertowanie na komunikaty (ang. messages) i przesyłanie do doręczyciela (ang. dispatcher) z biblioteki „*user*” (patrz rozdział 2.4.5 na stronie 49). Korzystać z niej powinny tylko sterowniki urządzeń wejściowych lub aplikacje je emulujące np. klawiatura ekranowa.

Menadżer wejść jest więc warstwą pośredniczącą pomiędzy urządzeniami wejściowymi a aplikacjami. Warstwa ta musi istnieć ponieważ zdarzenie wysłane przez urządzenie wejściowe jest wysyłane jednorazowo, tylko i wyłącznie do menadżera wejść. Gdyby pominąć tą warstwę sterownik musiał by przeszukiwać wszystkich słuchaczy (ang. listener), sprawdzać który z nich jest zainteresowany zdarzeniem które ma zostać wysłane i ew. je wysyłać. Podejście takie spowodowało by powtarzanie tego samego kodu, wielokrotnie w każdym sterowniku, oraz skomplikowało proces jego tworzenia. Dodatkowym, choć bardzo istotnym, zadaniem menadżera wejść jest generowanie dodatkowych komunikatów nie wynikających jednoznacznie ze zgłoszonego zdarzenia. Są to na przykład komunikaty MSG\_POINTERHOVER oraz MSG\_POINTERLEAVE wysyłane do okna w chwili najechania na nie rysikiem lub opuszczenia jego powierzchni.

Aby z poziomu sterownika powiadomić menadżer wejść o zaistniałym zdarzeniu należy wywołać funkcję inputEventNotify lub jeżeli powiadamiamy z funkcji obsługi przerwania: inputEventNotifyISR. Zastosowanie dwóch różnych funkcji podyktowane jest budową wewnętrzną systemu FreeRTOS [12]. Jako jej parametr podajemy wskaźnik na strukturę opisującą zdarzenie (patrz Kod źródłowy 2.26 na stronie 49).

**Kod źródłowy 2.27.** Struktura opisująca zdarzenie

struct inputEvent

{

UINT8 type;

UINT32 timestamp;

UINT8 action;

UINT32 param1;

UINT32 param2;

};

Pole type odpowiada za typ zgłaszanego zdarzenia i jest jednoznacznie powiązane z polem action. Dla każdego typu zdarzenia istnieje lista akcji jakie mogą zostać zgłoszone. Poniższa lista przedstawia dostępne typy wraz z akcjami jakie ich dotyczą:

1. EVENT\_TOUCH – zdarzenie od ekranu dotykowego
   1. EVENT\_TOUCH\_DOWN – wciśnięcie rysika
   2. EVENT\_TOUCH\_UP – zabranie rysika z powierzchni panelu dotykowego
   3. EVENT\_TOUCH\_MOVE – przesunięcie rysika
2. EVENT\_KEY – zdarzenie od klawiatury
   1. EVENT\_KEY\_DOWN – wciśnięcie przycisku
   2. EVENT\_KEY\_UP – zwolnienie przycisku
3. EVENT\_CHAR – zdarzenie od dowolnego urządzenia znakowego zgłaszane w chwili odebrania znaku. Nie każde urządzenie znakowe musi zgłaszać to zdarzenie. Jest ono zgłaszane tylko jeżeli odebranie znaku, ma powodować przekazanie go do graficznego interfejsu użytkownika
   1. EVENT\_CHAR\_GET – odebranie znaku od urządzenia
4. EVENT\_RCU – zdarzenie z odbiornika podczerwieni
   1. EVENT\_RCU\_CODE\_RECEIVED – pomyślne odebranie kodu od pilota
5. EVENT\_DISK – zdarzenie zamontowania lub odmontowania dysku
   1. EVENT\_DISK\_INSERTED – zamontowanie dysku
   2. EVENT\_DISK\_REMOVED – odmontowanie dysku

Pole timestamp jest czasem zgłoszenia zdarzenia podanym w milisekundach od chwili uruchomienia urządzenia. Jest ono uzupełniane automatycznie przez menadżer wejść więc można je pomijać.

Pola param1 oraz param2 są parametrami zgłaszanego zdarzenia i są one zależne od typu zgłoszenia.

Menadżer wejść udostępnia funkcje skrótowe służące do zgłaszania zdarzeń konkretnych typów bez konieczności zastanawiania się nad wartościami parametrów oraz bez konieczności uzupełniania poszczególnych pul struktury. Wszystkie potrzebne do zgłoszenia zdarzenia informacje przekazywane są przez parametry funkcji zgłaszającej. Kod źródłowy 2.27 przedstawia przykładową funkcję skrótową dla zgłoszenia zdarzenia od panelu dotykowego.

**Kod źródłowy 2.28.** Prototyp funkcji zgłaszającej zdarzenie od panelu dotykowego

retcode inputTouchEventNotify(UINT8 pAction,

UINT16 pPositionX, UINT16 pPositionY,

INT16 pSpeedX, INT16 pSpeedY);

Parametry pSpeedX oraz pSpeedY oznaczają prędkość przesuwania się rysika obliczoną na podstawie poprzedniej jego pozycji. Obliczanie prędkości leży po stronie sterownika. Przeznaczeniem wartości tej, jest zrealizowanie efektu znanego z dotykowych telefonów komórkowych, w których przewijana lista, kontynuuje przewijanie przez pewien czas po zwolnieniu dotyku. Czas ten obliczany jest na podstawie prędkości przesuwania palca, zaraz przed puszczeniem ekranu.

Pełna lista funkcji skrótowych („*inputEvent.h*”) oraz pełny kod źródłowy menadżera wejść („*inputTask.c*”) znajduje się na dołączonej płycie CD w katalogu: „*./source/lib/input/*”.

### User

Tworząc nową aplikację jesteśmy zobligowani do stworzenia nowego wątku w którym będzie ona pracować. Jeżeli będziemy chcieli odbierać komunikaty przetworzone przez menadżer wejść konieczne będzie również stworzenie i zarejestrowanie tak zwanego słuchacza komunikatów (ang. message listener). Jest to właśnie jeden z elementów biblioteki o nazwie „*user*”. Nazwa wzięła się z faktu, że komunikaty które przechodzą przez tę bibliotekę są komunikatami których efekty widoczne są bezpośrednio dla użytkownika systemu. Drugim elementem tej biblioteki jest tak zwany doręczyciel komunikatów (ang. dispatcher). Jest to zestaw funkcji używanych bezpośrednio przez aplikację w celu wysłania komunikatu np. do danej kontrolki graficznego interfejsu użytkownika.

Aby aplikacja mogła działać w sposób jaki znamy np. z systemu operacyjnego „*Windows*”, w głównym jej wątku musi znajdować się pętla komunikatów (ang. message loop). Jej zadaniem jest oczekiwanie na pojawienie się nowego komunikatu, nie zajmując tym czasu procesora. Po odebraniu komunikatu, pętla może obsłużyć komunikaty które zostały wysłane do wątku w którym pracuje, po czym za pośrednictwem doręczyciela przekazać odpowiednie komunikaty do odpowiednich okien. Kod źródłowy 2.28 przedstawia przykładową pętlę komunikatów.

**Kod źródłowy 2.29.** Przykładowa pętla komunikatów

struct msgListener \*list;

struct msg m;

list = msgListenerCreate(DEF\_MSG\_QUEUE\_SIZE);

while(msgListenerPeek(list, &m, NULL, 0, 0))

{

if (m.wnd == NULL)

{

switch(m.message)

{

case MSG\_DISKINSERTED:

LOG("Disk Inserted!");

break;

}

}

msgDispatch(&m);

}

msgListenerDelete(list);

W pierwszej linii deklarujemy wskaźnik na strukturę typu msgListener. Jak łatwo się domyślić jest to właśnie struktura opisująca słuchacza komunikatów. Następnie deklarujemy strukturę w której przechowywany będzie ostatni odebrany komunikat. Kolejnym krokiem, jest stworzenie instancji struktury słuchacza, przy pomocy funkcji msgListenerCreate. Jako jedyny parametr przyjmuje ona wielkość kolejki komunikatów jaką będzie posiadał nowo stworzony słuchacz. Makrodefinicja DEF\_MSG\_QUEUE\_SIZE określa wartość domyślną. Jest ona równa „*200*” i w większości przypadków rozmiar ten powinien okazać się wystarczający.

Kolejnym elementem jest pętla, w której warunkiem końca jest wynik wykonywania procedury msgListenerGet. Jest to właśnie pętla komunikatów.

Przyjrzyjmy się teraz bliżej funkcji msgListenerGet. Jest ona wywoływana jako warunek końca pętli. Standardowo zwracana wartość musi być więc różna od „*0*”. Wartość równa „*0*” jest zwracana wyłącznie gdy do wątku zostanie wysłany komunikat MSG\_QUIT czyli żądanie zakończenia pracy aplikacji. Ponadto wynik funkcji nie jest zwracany dopóki nie zostanie odebrany dowolny komunikat. Taki schemat działania jest odpowiedni dla aplikacji, która podczas braku dostępnych komunikatów, ma po prostu czekać i nie zajmować niepotrzebnie czasu procesora. Zupełnie inna sytuacja dotyczy gier, powiemy o tym za chwilę. Funkcja msgListenerGet ma na celu pobranie pojedynczego komunikatu z kolejki komunikatów słuchacza do którego wskaźnik przekazujemy jako pierwszy jej parametr. Drugim parametrem jest wskaźnik na strukturę komunikatu (struct msg) do której zostanie wpisany pobrany komunikat. Następny parametr określa czy chcemy pobierać tylko komunikaty skierowane do jednego konkretnego okna, czy wszystkie przeznaczone dla wątku w którym pracujemy. Dwa ostatnie parametry określają zakres komunikatów które chcemy pobierać. Przekazanie do nich wartości MSG\_KEYFIRST oraz MSG\_KEYLAST spowoduje, że pobierane będą tylko komunikaty dotyczące wciskanych przycisków. Funkcja msgListenerGet została zaprojektowana na wzór funkcji GetMessage znanej z API[[21]](#footnote-21) systemu operacyjnego „*Windows*” (WinAPI).

Kolejnym elementem pętli komunikatów jest instrukcja switch w której możemy (jeżeli jest to pożądane) obsłużyć komunikaty wysłane bezpośrednio do wątku w którym pracujemy (m.wnd == NULL).

Ostatnim już koniecznym do prawidłowego działania aplikacji elementem pętli komunikatów jest wywołanie funkcji msgDispatch. Funkcja ta jest częścią wspominanego wcześnie doręczyciela komunikatów. Jako jedyny parametr przyjmuje ona wskaźnik do struktury przechowującej komunikat, który ma zostać doręczony. Struktura ta posiada wskaźnik na okno, do którego kierowany jest komunikat. Zadaniem funkcji msgDispatch jest wywołanie procedury zdarzeniowej okna (patrz rozdział 2.4.7.1 na stronie 53).

Opisany mechanizm pozwala aby aplikacje zachowywały się tak jak życzy sobie tego użytkownik. Procedura zdarzeniowa okna jest docelowym miejscem w które miał dotrzeć komunikat o zaistniałym zdarzeniu. Na przykładzie wciśnięcia przycisku (wyświetlonego na wyświetlaczu) przy pomocy panelu dotykowego, opiszę wszystkie czynności które musiały zostać wykonane aby komunikat dotarł na miejsce docelowe. Użytkownik używając rysika wciska panel dotykowy na obszarze na którym wyświetlony jest przycisk. Sterownik panelu dotykowego odczytuje informację o pojawieniu się dotyku na panelu. Poprzez sterownik przetwornika analogowo cyfrowego, odczytywane są wartości napięć które pojawiły się na portach panelu dotykowego. Na podstawie wartości odfiltrowanych przez filtr cyfrowy, sterownik panelu dotykowego ustala pozycję dotyku na wyświetlaczu i przekazuje ją do menadżera wejść. Menadżer w reakcji na zgłoszone zdarzenie sprawdza, które okno lub kontrolka znajduje się w miejscu dotknięcia, po czym generuje komunikat MSG\_TOUCHDOWN przypisany do niej i rozsyła go, do wszystkich słuchaczy. Słuchacze kolejno, w zależności od priorytetów wątków w których pracują, odbierają tę wiadomość i przekazują ją do pętli głównej aplikacji. W pętli głównej doręczyciel komunikatów wywołuje procedurę zdarzeniową okna lub kontrolki do której kierowana była wiadomość. Zakładając że był to przycisk, zostaje on podświetlony i oczekuje na kolejne komunikaty. Przedstawiona droga jest dosyć skomplikowana, ale konieczna aby uzyskać tak wysoki poziom abstrakcji przy pisaniu aplikacji użytkowych.

Gdybyśmy chcieli, zamiast aplikacji okienkowej napisać grę, która oczywiście też musi reagować na komunikaty, ale jednocześnie pomiędzy obsługą kolejnych komunikatów zapewniać płynność animacji na wyświetlaczu, zastosowanie funkcji msgListenerGet było by niewskazane. Do tego celu przygotowana została funkcja msgListenerPeek która nie zatrzymuje wykonywania programu w przypadku braku komunikatów oczekujących na obsługę. Kod źródłowy 2.29 przedstawia pętlę główną przygotowaną do współpracy z tą funkcją.

**Kod źródłowy 2.30.** Pętla komunikatów dla gier

struct msgListener \*list;

struct msg m;

m.message = MSG\_NULL;

list = msgListenerCreate(DEF\_MSG\_QUEUE\_SIZE);

while(m.message != MSG\_QUIT)

{

if (msgListenerPeek(list, &m, NULL, 0, 0))

{

msgDispatch(&m);

}

else

{

// Kod wykonywany, jeżeli w kolejce nie ma już

// żadnych komunikatów do obsłużenia.

// Na przykład rysowanie kolejnej klatki animacji

}

}

msgListenerDelete(list);

Ostatnie dwie funkcje którym chciałbym poświęcić uwagę są funkcje msgSend oraz msgPost. Obie przyjmują dokładnie takie same parametry. Pierwszym z nich jest wskaźnik na okno do którego jest kierowany komunikat, drugim kod komunikatu, a dwa ostatnie parametry zależne są od wysyłanego komunikatu. Konkretne komunikaty oraz parametry jakie trzeba z nimi przekazywać zostaną dokładniej opisane podczas omawiania biblioteki interfejsu graficznego użytkownika (patrz rozdział 2.4.7 na stronie 55). Funkcja msgPost dodaje komunikat do kolejki komunikatów słuchacza, czyli zleca jego wykonanie w kolejnym przebiegu pętli komunikatów przypisanej do tego słuchacza. Natomiast funkcja msgSend, wywołuje procedurę zdarzeniową okna natychmiast. Dla komunikatów które pobierają informacje zwrotne konieczne jest używanie funkcji msgSend. Kod źródłowy 2.30 przedstawia przykład użycia obu funkcji.

**Kod źródłowy 2.31.** Przykład użycia funkcji msgSend oraz msgPost

msgSend(btnPlay, MSG\_SETTEXT, 0, (UINT32)"PLAY");

msgPost(btnPlay, MSG\_PAINT, 0, 0);

Programista znający API systemu operacyjnego „Windows” szybko zauważy, że panują tutaj bardzo podobne zasady. Taki właśnie był cel, aby aplikacje na proste urządzenia wbudowane, pisało się tak samo łatwo jak na system operacyjny „Windows” – cel udało się osiągnąć.

Kod źródłowy biblioteki „*user*” znajduje się na dołączonej płycie CD w katalogu: „*./source/lib/user/*”.

### Graficzny interfejs użytkownika

Biblioteka graficznego interfejsu użytkownika została zaprojektowana wzorując się na API systemu „*Windows*”. Głównym założeniem było, aby programista obeznany w „*WinAPI*” z łatwością poradził sobie z napisaniem aplikacji opartej na tej bibliotece. Pomimo pewnych różnic w nazewnictwie funkcji i definicji, ogólna zasada działania jest bardzo podobna. Obecnie biblioteka wspiera następujące kontrolki takie jak okno główne, tekst statyczny, przycisk oraz pasek postępu.

#### Okno

Podobnie jak w systemie „Windows” oknem możemy nazwać każdy element „*GUI*”. Podrozdział ten ma jednak na celu przedstawienie okna głównego (ang. main window) Okno to jest kontenerem który umożliwia zgrupowanie innych elementów stanowiących razem funkcjonalną całość. Aktualnie istnieje jedynie możliwość tworzenia okien głównych zajmujących całą powierzchnię ekranu. Przełączania pomiędzy nimi dokonujemy wywołując funkcję guiSetCurrentMainWindow, której jako parametr podajemy wskaźnik na okno które ma zostać aktywowane.

Aby stworzyć okno będziemy musieli najpierw zarejestrować klasę tworzonego okna. Klasa okna określa jego wygląd oraz zachowanie. Wygląd okna jest określany poprzez podanie indeksu stylu z tablicy zdefiniowanej przez aplikację. Tablica ta przyjmuje typ struct guiWinStyle i określa kolor tła, kolor tekstu oraz grubość i kolor obramowania. Kod źródłowy przykładowej tablicy znajduje się na dołączonej płycie CD pod ścieżką: „./source/app/rsc/winstyle/winstyle.h”. Zachowanie okna określane jest poprzez podanie wskaźnika na procedurę zdarzeniową którą będzie używać każda instancja definiowanej klasy okna. XX przedstawia przykładową definicję klasy okna.

**Kod źródłowy 2.32.** Przykład definicji klasy okna

struct guiWndClassInfo wci;

wci.className = "wndplayer";

wci.windowStyle = WS\_NONE;

wci.colorStyle.shIdx = WSTL\_WINDOW\_SH;

wci.colorStyle.hlIdx = WSTL\_WINDOW\_HL;

wci.colorStyle.selIdx = WSTL\_WINDOW\_SH;

wci.colorStyle.gryIdx = WSTL\_WINDOW\_SH;

wci.windowProc = appPlayerWndProc;

guiRegisterWindowClass(&wci);

Aby używać zdefiniowanej w ten sposób klasy, trzeba ją najpierw zarejestrować w systemie. Dokonujemy tego wywołując funkcję guiRegisterWindowClass widoczną w ostatniej linii zaprezentowanego przykładu.

Okno główne posiada specyficzną funkcję je tworzącą (patrz Kod źródłowy 2.32 na stornie 58). Dla wszystkich innych kontrolek istnieje jedna, wspólna funkcja o zbliżonej budowie (patrz XX na stronie XX).

**Kod źródłowy 2.33.** Prototyp funkcji guiCreateMainWindow

struct guiMainWindow \*guiCreateMainWindow (

const char\* pClassName,

const char\* pCaption, UINT32 pStyle,

UINT16 pId, UINT16 pFocusedId,

UINT16 pX, UINT16 pY,

UINT16 pW, UINT16 pH);

**Tabela 2.3.** Lista parametrów funkcji guiCreateMainWindow

Źródło: Opracowanie własne

|  |  |
| --- | --- |
| Parametr | Opis |
| pClassName | Nazwa klasy okna, które chcemy stworzyć. W przeciwieństwie do „*WinAPI*” nie istnieje ograniczenie, że klasa musi zostać zarejestrowana przez tą samą aplikację która próbuj stworzyć jej okno. |
| pCaption | Tytuł okna. W obecnej wersji biblioteki, tytuł nie jest wyświetlany, ponieważ okno nie posiada standardowego obramowania. Wyświetlanie obramowania okna wraz z belką tytułową zostanie zaimplementowana jak tylko będzie możliwe wyświetlanie wielu okien jednocześnie. Wtedy będzie to niezbędne. |
| pStyle | Style określające wygląd tworzonego okna. Dla okien głównych obecnie wspierany jest jedynie styl WS\_VISIBLE. Wywołanie funkcji z tym stylem spowoduje automatyczne ustawienie stworzonego okna jako aktywne oraz narysowanie go na ekranie. |
| pId | Identyfikator okna, używany podczas przełączania aktywnych kontrolek przy pomocy przycisków kierunkowych na pilocie zdalnego sterowania lub klawiaturze. |
| pFocusedId | Identyfikator aktywnej kontrolki wewnątrz tego okna. |
| pX, pY | Pozycja lewego górnego rogu okna na ekranie. |
| pW, pH | Szerokość (pW) oraz wysokość (pH) tworzonego okna. |

Stworzone w ten sposób okno, jest gotowe do odbierania komunikatów z systemu oraz do tworzenia kontrolek które przyjmą je jako rodzica. Wszystkie te kontrolki tworzone będą w bardzo zbliżony sposób. Jest to poziom abstrakcji który był założeniem i który udało się osiągnąć. Wywołując jedno polecenie jesteśmy w stanie stworzyć dowolną kontrolkę np. przycisk, który może reagować na polecenia użytkownika i w zależności od tych poleceń odpowiednio reagować, np. zmieniać kolor po najechaniu na niego rysikiem oraz odsyłać komunikaty zwrotne do okna rodzica. Na podstawie tych komunikatów w procedurze zdarzeniowej okna rodzica możemy reagować na akcje wykonane przez użytkownika aplikacji.

#### Tekst statyczny

Jedną z podstawowych i zarazem najprostszą kontrolką, jest tekst statyczny. Jest to kontrolka wyłącznie informacyjna. Oznacza to, że nie odsyła żadnych komunikatów zwrotnych do okna rodzica. Aby stworzyć kontrolkę zdefiniowaną przez system, jak na przykład tekst statyczny nie jest konieczna rejestracja klasy. Tę czynność wykonał za nas system operacyjny, rejestrując klasy kontrolek podstawowych które możemy używać w aplikacji, nie przywiązując uwagi do ich wewnętrznego sposobu działania. W celu stworzenia kontrolki dowolnego typu wywołujemy funkcję guiCreateWindow. Jej parametry są niemal identyczne, jak w przypadku tworzenia okna głównego. Jedynymi różnicami są konieczność podania identyfikatorów elementów „*GUI*” które znajdują się po każdej stronie od tworzonej kontrolki oraz możliwość podania dodatkowych danych które będą przez nią przechowywane. Kod źródłowy 2.34 przedstawia przykład tworzenia tekstu statycznego. Efektem wykonanie tego polecenia będzie pojawienie się napisu w obszarze okna podanego jako rodzic (patrz Rysunek 2.5 na stronie 60).

**Kod źródłowy 2.34.** Przykład tworzenia tekstu statycznego

stTitle = guiCreateWindow("statictext",

"Przykladowy tekst",

WS\_VISIBLE, // Styl

IDC\_ST\_TITLE, // Id tworzonej kontrolki

IDC\_ST\_TITLE, // Id kontrolki po lewej stronie

IDC\_ST\_TITLE, // Id kontrolki po prawej stronie

IDC\_ST\_TITLE, // Id kontrolki na górze

IDC\_ST\_TITLE, // Id kontrolki na dole

ST\_TITLE\_X, ST\_TITLE\_Y, // Pozycja

ST\_TITLE\_W, ST\_TITLE\_H, // Rozmiar

(struct guiWindow \*) wndPlayer, // Wskaźnik na okno rodzica

0 // Dane dodatkowe

);

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!OBRAZEK!!!!!!!!!!!!!!!!!!!

**Rysunek 2.5.** Kontrolka „*tekst statyczny*”

Źródło: Opracowanie własne

Zdefiniowanie jakie inne kontrolki znajdują się w otoczeniu tworzonego elementu „*GUI*” umożliwia nawigację po interfejsie przy pomocy przycisków kierunkowych pilota lub klawiatury.

Ze można wysylac msg, jak i co robia.

#### Przycisk

Asdasdsad

#### Pasek postępu

Asdasdasd

ROZDZIAŁ 3

# Aplikacja demonstracyjna

Asdasdasd

## Założenia

Asdasdasd

## Interfejs użytkownika

Asdasdasd

DODATEK A

# Zawartość dodatkowej płyty CD

Asdasdasd

# Bibliografia

[1] Wikipedia, Otwarte oprogramowanie: <http://pl.wikipedia.org/wiki/Open_source> [dostęp: 2013-01-29]

[2] Wikipedia, SoC: <http://pl.wikipedia.org/wiki/System-on-a-chip> [dostęp: 2013-01-29]

[3] MikroElektronika: <http://www.mikroe.com/multimedia/pic32mx7> [dostęp: 2013-01-29]

[4] Wolfson Microelectronics plc: WM8731 Datasheet <http://www.wolfsonmicro.com/documents/uploads/data_sheets/en/WM8731.pdf>, 2012

[5] Himax Technologies, Inc: HX8347 Datasheet <http://www.displayfuture.com/Display/datasheet/controller/HX8347.pdf>, 2007

[6] Vishay Semiconductors: IR Receiver Modules for Remote Control Systems, <http://www.vishay.com/docs/82006/tsop11xx.pdf>, 2012

[7] Jason Howie: NEC Infrared Transmission Protocol, <http://wiki.altium.com/display/ADOH/NEC+Infrared+Transmission+Protocol>, 2008

[8] Andrew S. Tanenbaum, Albert S. Woodhull: *Operating Systems Design and Implementation, Third Edition*, Prentice Hall, 2006

[9] FatFS, <http://elm-chan.org/fsw/ff/00index_e.html> [dostęp: 2013-01-31]

[10] Dominic Sweetman: *See MIPS® Run - Second Edition*, Elsevier, San Francisco, 2007

[11] Nicolas Melot: *Study of an operating system: FreeRTOS*, <http://stiff.univ-brest.fr/~boukhobza/images/stories/Documents/Teachings/OSM/expo/FreeRTOS_Melot.pdf>, 2009

[12] Rich Goyette: *An Analysis and Description of the Inner Workings*

*of the FreeRTOS Kernel*,<http://www.mikrocontroller.net/attachment/95930/FreeRTOSPaper.pdf>, 2007

[13] Microchip: *Section 17. 10-bit Analog-to-Digital Converter,* <http://ww1.microchip.com/downloads/en/DeviceDoc/61104E.pdf>, 2011

[14] Microchip: *Section 15. Input Capture,* <http://ww1.microchip.com/downloads/en/DeviceDoc/61122F.pdf>, 2010

# Spis tabel

[Tabela 2.1. Struktura katalogów projektu 19](#_Toc347584395)

[Tabela 2.2. Lista poleceń układu graficznego FPGA 41](#_Toc347584396)

# Spis rysunków

[Rysunek 1.1. Zdjęcie płyty "multimedia for PIC32MX7" z obu stron 12](#_Toc347584277)

[Rysunek 1.2. Montaż odbiornika podczerwieni TSOP1138 14](#_Toc347584278)

[Rysunek 1.3. Wyniki testów biblioteki FatFS 18](#_Toc347584279)

[Rysunek 2.1. Wyjątek procesora. 22](#_Toc347584280)

[Rysunek 2.2. Przepełnienie stosu. 22](#_Toc347584281)

[Rysunek 2.3. 16-bitowy interfejs komunikacyjny i80 40](#_Toc347584282)

# Spis kodów źródłowych

[**Kod źródłowy 2.1.** Wybór sterownika wyświetlacza w czasie kompilacji 23](#_Toc347615523)

[**Kod źródłowy 2.2.** Konfiguracja sprzętowa sterownika wyświetlacza 24](#_Toc347615524)

[**Kod źródłowy 2.3.** Prototyp funkcji boardInit 25](#_Toc347615525)

[**Kod źródłowy 2.4.** Struktura bazowa dla sterowników 25](#_Toc347615526)

[**Kod źródłowy 2.5.** Przykład wykorzystania funkcji hldDeviceGetByType 26](#_Toc347615527)

[**Kod źródłowy 2.6.** Możliwe stany pracy urządzenia 27](#_Toc347615528)

[**Kod źródłowy 2.7.** Wykorzystanie pola priv struktury hldDevice 27](#_Toc347615529)

[Kod źródłowy 2.8. Warstwa abstrakcji dla sterowników portu transmisji szeregowej 28](#_Toc347615530)

[**Kod źródłowy 2.9.** Warstwa abstrakcji dla sterowników urządzeń znakowych 29](#_Toc347615531)

[**Kod źródłowy 2.10.** Warstwa abstrakcji dla sterowników przetworników ADC 30](#_Toc347615532)

[**Kod źródłowy 2.11.** Funkcja obsługi przerwania przetwornika 31](#_Toc347615533)

[**Kod źródłowy 2.12.** Funkcja odczytująca wartość podanego kanału przetwornika 32](#_Toc347615534)

[**Kod źródłowy 2.13.** Warstwa abstrakcji dla sterownika odbiornika podczerwieni 32](#_Toc347615535)

[**Kod źródłowy 2.14.** Parametry konfiguracyjne sterownika odbiornika podczerwieni 33](#_Toc347615536)

[**Kod źródłowy 2.15.** Inicjalizacja urządzenia „Input Capture” do współpracy z zegarem sprzętowym „Timer2”. 34](#_Toc347615537)

[**Kod źródłowy 2.16.** Obsługa przerwania wykrycia zbocza 34](#_Toc347615538)

[**Kod źródłowy 2.17.** Warstwa abstrakcji dla sterowników audio 35](#_Toc347615539)

[**Kod źródłowy 2.18.** Warstwa abstrakcji dla sterowników dysków 37](#_Toc347615540)

[**Kod źródłowy 2.19.** Warstwa abstrakcji dla sterowników układów graficznych 39](#_Toc347615541)

[**Kod źródłowy 2.20.** Rysowanie piksela przez układ graficzny FPGA 43](#_Toc347615542)

[**Kod źródłowy 2.21.** Warstwa abstrakcji sterownika panelu dotykowego 44](#_Toc347615543)

[**Kod źródłowy 2.22**. Przykład użycia filtru cyfrowego 46](#_Toc347615544)

[**Kod źródłowy 2.23.** Użycie modułu LOG 47](#_Toc347615545)

[**Kod źródłowy 2.24.** Przykładowa funkcja konsolowa 48](#_Toc347615546)

[**Kod źródłowy 2.25.** Przykład rejestracji funkcji w konsoli 48](#_Toc347615547)

[**Kod źródłowy 2.26.** Struktura opisująca zdarzenie 49](#_Toc347615548)

[**Kod źródłowy 2.27.** Prototyp funkcji zgłaszającej zdarzenie od panelu dotykowego 51](#_Toc347615549)

[**Kod źródłowy 2.28.** Przykładowa pętla komunikatów 52](#_Toc347615550)

[**Kod źródłowy 2.29.** Pętla komunikatów dla gier 54](#_Toc347615551)

[**Kod źródłowy 2.30.** Przykład użycia funkcji msgSend oraz msgPost 54](#_Toc347615552)

1. (ang. Inter-Process Communication – IPC) [↑](#footnote-ref-1)
2. (ang. Real-Time Operating System – RTOS) [↑](#footnote-ref-2)
3. (ang. Pulse Width Modulation – PWM) [↑](#footnote-ref-3)
4. (ang. System on Chip – SOC) - układ scalony zawierający kompletny system elektroniczny, w tym układy cyfrowe, analogowe (także radiowe) oraz cyfrowo-analogowe [2] [↑](#footnote-ref-4)
5. (ang. Graphical User Interface – GUI) [↑](#footnote-ref-5)
6. 44,1kHz/16bitów [↑](#footnote-ref-6)
7. (ang. Parallel Master Port – PMP) [↑](#footnote-ref-7)
8. (ang. Inter-Integrated Circuit – I2C/IIC) [↑](#footnote-ref-8)
9. (ang. Serial Peripheral Interface) [↑](#footnote-ref-9)
10. 320x240 pikseli [↑](#footnote-ref-10)
11. (ang. Frame Buffer – FB) [↑](#footnote-ref-11)
12. (ang. Field Programmable Gate Array – FPGA) [↑](#footnote-ref-12)
13. Odczyt animacji: 15 fps, 40 kB na jedną klatkę obrazu. Wymagana prędkość odczytu 600 kB/s [↑](#footnote-ref-13)
14. (ang. Direct Memory Access – DMA) [↑](#footnote-ref-14)
15. (ang. Hadrware Abstraction Layer – HAL) [↑](#footnote-ref-15)
16. (ang. Exception Program Counter – EPC) [↑](#footnote-ref-16)
17. (ang. High Level Driver – HLD) [↑](#footnote-ref-17)
18. (ang. Low Level Driver – LLD) [↑](#footnote-ref-18)
19. (ang. Universal Asynchronous Receiver and Transmitter – UART) [↑](#footnote-ref-19)
20. (ang. First In First Out – FIFO) [↑](#footnote-ref-20)
21. (ang. Application Programming Interface – API) [↑](#footnote-ref-21)