

**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

Niniejsza praca ma na celu przedstawienie procesu projektowania oraz budowy wewnętrznej prostego systemu operacyjnego, dedykowanego dla urządzeń wbudowanych. Głównym jego założeniem było osiągnięcie prostoty tworzenia aplikacji użytkowych, znanej z desktopowych systemów operacyjnych. Efektem wykonania pracy będzie aplikacja demonstrująca możliwości systemu, w postaci odtwarzacza plików muzycznych. Stworzony system został zaprojektowany do współpracy z układem graficznym zbudowanym w oparciu o układ FPGA.

Słowa kluczowe

system operacyjny, odtwarzacz audio, RTOS, HAL, GUI, PIC32

Abstract

This thesis aims to present the design process and internal structure of a simple operating system dedicated to use with embedded devices. Main assumption was to achieve simplicity of application development known from desktop operating systems. The result of completed work will be application which will present capabilities of the system. This demo application is an audio player. Created system was designed to work with FPGA based graphics controller.

Keywords

Operating system, audio player, RTOS, HAL, GUI, PIC32

Wprowadzenie.....	7
Zastosowane technologie	9
1.1 Hardware	9
1.1.1 Procesor	10
1.1.2 Płyta ewaluacyjna	12
1.1.3 Układ graficzny.....	14
1.2 Software	14
1.2.1 Język programowania	15
1.2.2 System operacyjny	15
1.2.2.1 System operacyjny jako menadżer dostępu do zasobów.....	15
1.2.2.2 System operacyjny jako maszyna wirtualna.....	16
1.2.3 System plików.....	17
1.2.4 Biblioteka graficzna	18
Architektura systemu	19
2.1 Struktura katalogów	19
2.2 FreeRTOS	20
2.3 Warstwa abstrakcji sprzętu (HAL).....	23
2.3.1 Board init	24
2.3.2 HLD	25
2.3.3 LLD.....	28
2.3.3.1 UART	29
2.3.3.2 ADC.....	31
2.3.3.3 IR	33
2.3.3.4 AUDIO	36
2.3.3.5 DISK.....	38
2.3.3.6 LCD	39
2.3.3.7 TOUCH	44
2.4 Biblioteki.....	46
2.4.1 Biblioteka standardowa.....	46
2.4.1.1 Filtr cyfrowy	46
2.4.1.2 LOG.....	47
2.4.2 Konsola	48

2.4.3	System plików.....	49
2.4.4	Audio.....	50
2.4.5	Menadżer wejść.....	52
2.4.6	User	54
2.4.7	Graficzny interfejs użytkownika (GUI)	58
2.4.7.1	Okno	58
2.4.7.2	Tekst statyczny	60
2.4.7.3	Przycisk.....	62
2.4.7.4	Pasek postępu.....	64
Aplikacja demonstracyjna		66
3.1	Założenia	66
3.2	Interfejs użytkownika	67
Zakończenie		70
Zawartość dodatkowej płyty CD		71
Bibliografia		72
Spis tabel		73
Spis rysunków.....		74
Spis kodów źródłowych		75

Wprowadzenie

Tematem pracy jest opracowanie projektu oraz realizacja odtwarzacza audio. Rozumie się przez to napisanie oprogramowania oraz dobór platformy sprzętowej posiadającej odpowiednie peryferia konieczne do przeprowadzenia pierwszych testów funkcjonalnych.

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 tylko 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 (IPC¹) – funkcje te zostaną dostarczone przez wybrany system operacyjny czasu rzeczywistego (RTOS²).

Niniejsza praca traktuje będzie o procesie przygotowywania takiego oprogramowania, od stworzenia pierwszego sterownika urządzenia, przez napisanie bibliotek, aż do przygotowania aplikacji widocznej dla użytkownika.

Projekt zrealizowany na potrzeby pracy, przygotowany został do współpracy z procesorem graficznym FPGA, zaprojektowanym przez Bartosza Zamolskiego, w ramach osobnej pracy inżynierskiej.

Ponieważ niemożliwe było umieszczenie w pracy całości kodu źródłowego, został on zamieszczony na dołączonej płycie CD. W celu dokładniejszego zapoznania się z poruszonymi tematami, zalecane jest czytanie pracy z możliwością wglądu w kod. Komentarze w nim zawarte napisane zostały w języku angielskim w celu rozszerzenia grupy potencjalnych odbiorców (kod źródłowy dostępny jest na licencji GNU GPL v3 [15] i jest możliwy do pobrania pod adresem <http://code.google.com/p/intensesx/>).

¹ (ang. Inter-Process Communication – IPC)

² (ang. Real-Time Operating System – RTOS)

Cel pracy

Tematem pracy jest „projekt i realizacja odtwarzacza audio formatów bezstratnych do współpracy z układem graficznym FPGA”. Temat ten został znacznie rozszerzony. Celem pracy stało się stworzenie prostego systemu operacyjnego dedykowanego dla urządzeń wbudowanych, a następnie przygotowanie sterownika graficznego, współpracującego z układem graficznym FPGA, zaprojektowanym w ramach osobnej pracy. System ten powinien być uniwersalny i umożliwiać wykorzystanie w przyszłych projektach. Głównym jego założeniem jest osiągnięcie prostoty tworzenia aplikacji, znanej z desktopowych systemów operacyjnych. Odtwarzacz audio formatów bezstratnych został zaprojektowany jako jedna z takich aplikacji. Demonstruje on jednocześnie zakładaną prostotę ich tworzenia.

Zakres pracy

Analiza rozwiązań odtwarzaczy audio formatów bezstratnych. Zaprojektowanie oraz napisanie API dla układu zbudowanego na wybranym mikrokontrolerze, umożliwiającego wyświetlanie wybranych efektów graficznych za pomocą układu FPGA. Praktyczna realizacja urządzenia demonstrującego możliwości układu w postaci odtwarzacza formatów bezstratnych audio. Badania laboratoryjne modelu odtwarzacza, we współpracy z zewnętrznym układem wyświetlającym zbudowanym na module FPGA w ramach osobnej pracy.

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 procesowi przygotowania do rozpoczęcia prac nad projektem. Zostaną w nim przedstawione dylematy które pojawiły się, 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.

1.1 Hardware

Pomysł stworzenia niniejszej pracy dyplomowej powstał podczas kończenia jednego 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 (PWM³). 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

³ (ang. Pulse Width Modulation – PWM)

multimedialnych. Założenia te pozwoliły ustalić wymagania dla platformy sprzętowej, na której będzie rozwijane tworzone oprogramowanie:

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 przedstawia będzie zastosowaną platformę oraz uzasadnia wykorzystanie wybranych elementów.

1.1.1 Procesor

Wybór sprzętu, na którym projekt będzie rozwijany warto rozpocząć od serca układu czyli procesora. W dzisiejszych czasach różnica między procesorem a mikrokontrolerem zaciera się, ponieważ właściwie każdy układ scalony, który nazywamy procesorem, jest w istocie mikrokontrolerem. Popularny staje się termin SoC⁴ zarezerwowany dla procesorów dedykowanych do konkretnych rozwiązań, np. odbiorników telewizji satelitarnej. Każdy z procesorów posiada przynajmniej kilka urządzeń peryferyjnych, co pozwala sklasyfikować go jako mikrokontroler. Dlatego w pracy terminy te będą używane zamiennie.

Wymagania, które zostały postawione dla całej platformy sprzętowej, implikują kryteria jakie powinniśmy wziąć pod uwagę przy wyborze procesora. Założono, 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. Pojęcie to określa:

1. Przystosowanie układu do wyświetlania graficznego interfejsu użytkownika (GUI⁵) oraz możliwie szybkiej reakcji tego interfejsu na polecenia ze strony użytkownika.
2. Możliwość odtwarzania plików audio, co najmniej w tak zwanej jakości „CD”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

⁴ (ang. System on Chip – SOC) - układ scalony zawierający kompletny system elektroniczny, w tym układy cyfrowe, analogowe (także radiowe) oraz cyfrowo-analogowe [2]

⁵ (ang. Graphical User Interface – GUI)

⁶ 44,1kHz/16bitów

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, konieczne będzie zastosowanie procesora posiadającego 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 (PMP⁷) 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 I²C⁸ oraz SPI⁹ – 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. Szczegółowa znajomość architektury procesorów firmy Microchip, przesądziła o wyborze tego producenta. Nie był to jedyny argument 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 9) okazało się, że procesory 8-bitowe są zbyt wolne do zastosowań multimedialnych. Po pierwszych próbach z procesorem PIC18 stało się jasne, że należy zastosować 32-bitową rodzinę procesorów. Został zastosowany model PIC32MX460F512L. Mimo 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, w przypadku tego modelu jest to 32 kB.

Aby uniknąć wymienionych problemów wybrany został 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.

⁷ (ang. Parallel Master Port – PMP)

⁸ (ang. Inter-Integrated Circuit – I²C/IIC)

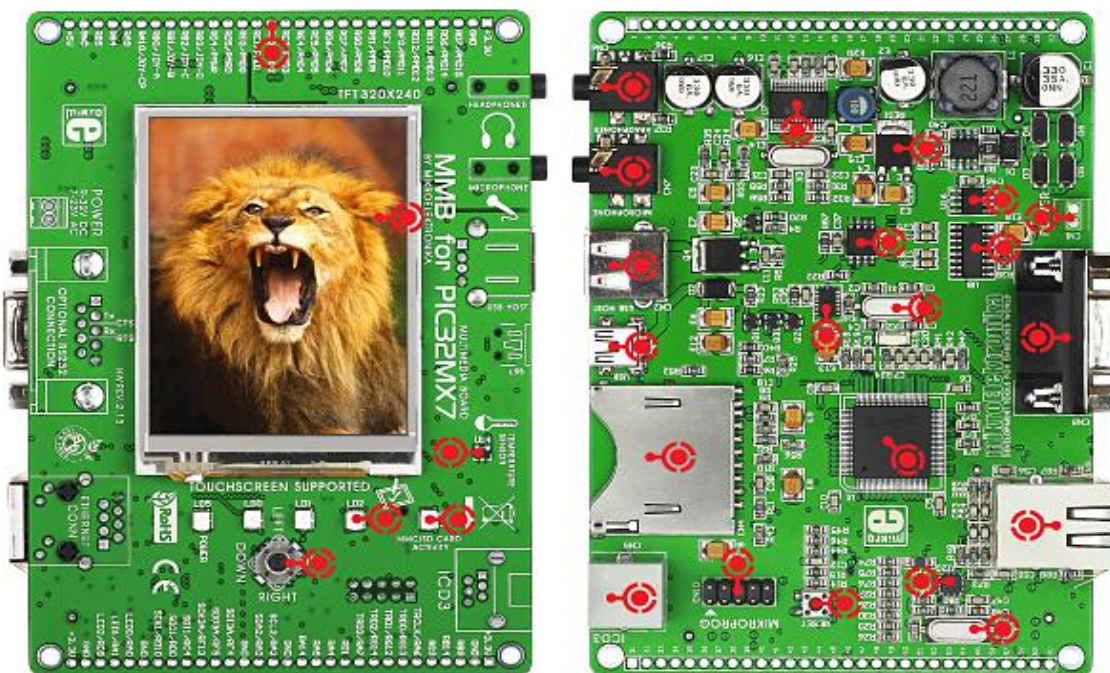
⁹ (ang. Serial Peripheral Interface)

1.1.2 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 jej celem. Projekt i wykonanie płyt, mogłoby być osobnym tematem pracy inżynierskiej. Dlatego wykorzystane zostało gotowe rozwiązanie.

Mimo 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 oferta firmy MikroElektronika, która jest polecanym przez firmę Microchip dostawcą narzędzi deweloperskich. Interesująca okazała się być płyta o nazwie „multimedia for PIC32MX7” posiadająca wszystkie wymagane peryferia oprócz odbiornika podczerwieni. Po jego dołożeniu płyta spełniała założenia. Cena płyty wynosi \$149 i została ona zasponsorowana, na potrzeby tej pracy inżynierskiej, przez firmę „MB Turnkey Design” współpracującą z Wydziałem Elektrycznym ZUT, 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 dwa wbudowane, 24-bitowe przetworniki cyfrowo analogowe

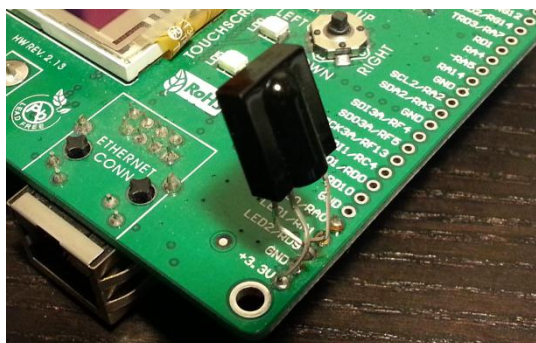
umożliwiają pracę z maksymalną częstotliwością próbkowania 96 kHz [4] (to znacznie więcej niż założono). Układ ten posiada również wbudowany wzmacniacz słuchawkowy oraz dwa przetworniki analogowo cyfrowe, pozwalające na obsługę nie tylko wyjścia, ale również wejścia stereo. Na wybranej płycie ewaluacyjnej wyprowadzone zostało stereofoniczne gniazdo słuchawkowe oraz monofoniczne gniazdo mikrofonowe. W pracy zostanie wykorzystane jedynie stereofoniczne wyjście audio.

Zastosowany na płycie wyświetlacz o rozdzielczości QVGA¹⁰ 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 1.1.3 na stronie 14). Biorąc pod uwagę, iż prace nad obydwojema 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 karty SD, port RS232 oraz dodatkowe, nie wymagane peryferia, takie jak złącze Ethernet, czujnik temperatury, gniazdo USB (slave i host), cztery diody LED i joystick.

Dzięki wyprowadzeniu przez producenta płyty wszystkich portów zastosowanego mikrokontrolera na złącza znajdujące się na krawędziach laminatu, posiada ona możliwość zainstalowania dowolnych, dodatkowych urządzeń. Tego rodzaju rozwiązanie pozwoliło połączyć układ z osobną płytą ewaluacyjną zawierającą strukturę FPGA oraz umożliwiło dołączenie jedynego brakującego urządzenia peryferyjnego – odbiornika podczerwieni.

Zastosowany został odbiornik TSOP1138 [6] pracujący na częstotliwości nośnej 38 kHz oraz kompatybilny z protokołem transmisji NEC [7]. Jest on jednym z najczęściej stosowanych przez producentów sprzętu elektronicznego protokołem. Odbiornik został przylutowany do płyty, wykorzystując złącza z wyprowadzeniami portów mikrokontrolera (patrz Rysunek 1.2 na stronie 13).



1.1.3 Układ graficzny

Warto teraz powrócić do projektu zabawki 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 (FB¹¹) 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 (FPGA¹²). Ponieważ Bartosz Zamolski zaproponował zaprojektowanie takiego układu graficznego, jako temat jego pracy inżynierskiej, stało się jasne, że moje urządzenie musi współpracować z tym właśnie układem.

1.2 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. 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.

¹¹ (ang. Frame Buffer – FB)

¹² (ang. Field Programmable Gate Array – FPGA)

1.2.1

1.2.2 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 realizowanym projektem wykluczył możliwość łatwej zmiany kompilatora.

1.2.3 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.

1.2.3.1 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 dostęp do danego zasobu. Zarówno za możliwość uruchamiania kilku wątków jednocześnie, jak i komunikację 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, postanowiono wykorzystać jeden z nich. Ze względu na ilość wspieranych architektur, procesorów, powszechność oraz licencję, został wybrany system operacyjny czasu rzeczywistego FreeRTOS. Ciekawym aspektem tej pracy inżynierskiej będzie stworzenie prostego systemu operacyjnego, zbliżonego do systemów desktopowych, wykorzystując system czasu rzeczywistego, jako menadżer dostępu do zasobów.

1.2.3.2 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 wykorzystany zostanie 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 „+” wyświetlony na panelu. Te informacje przez różne warstwy abstrakcji udostępnia programiście system operacyjny.

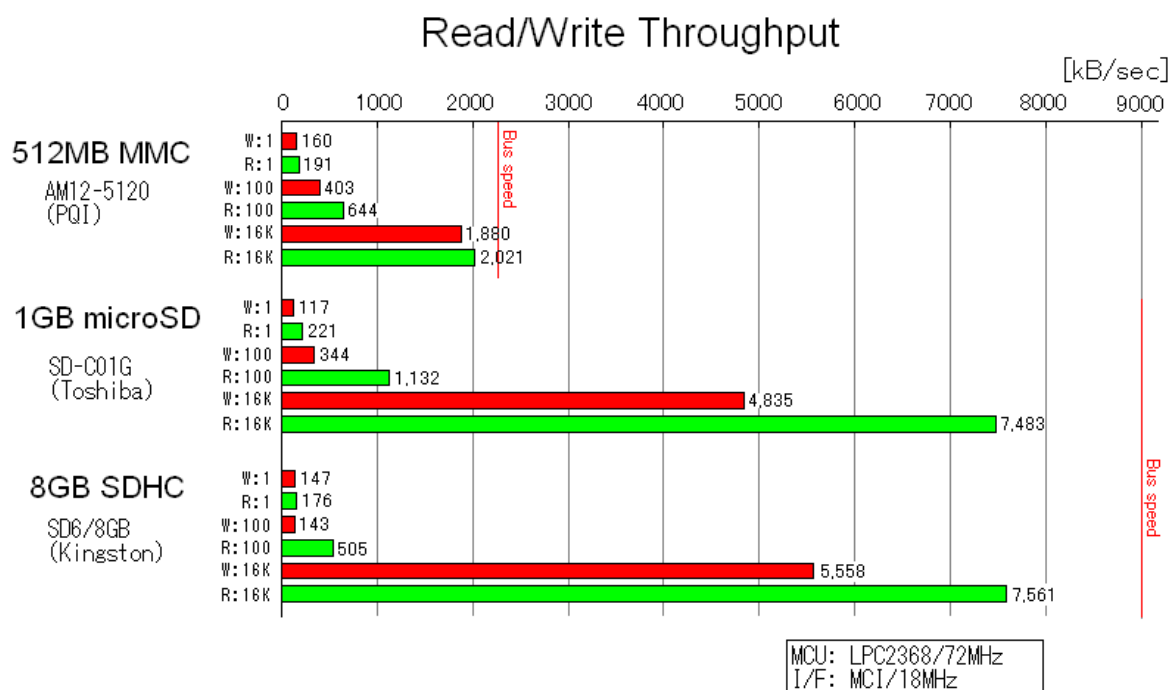
To właśnie nad tą funkcją systemu operacyjnego chciałbym skupić się w pracy najbardziej. Samodzielne przejście przez wszystkie etapy tworzenia tej funkcji, pozwoliło mi dogłębnie zrozumieć zasady działania dzisiejszych systemów operacyjnych oraz zgłębić wiedzę dotyczącą projektowania tego typu systemów.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

1.2.4 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 zapadła decyzja o użyciu jednej z gotowych implementacji.

Projekt zabawki (patrz rozdział 1.1 na stronie 9) 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 odczytu¹³. Konieczna była modyfikacja polegająca na zastosowaniu kontrolera DMA¹⁴ 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, podjęta została decyzja o użyciu implementacji o nazwie FatFS [9]. Testy zostały przeprowadzone na procesorze o podobnej częstotliwości taktowania, więc można spodziewać się zbliżonych wyników (patrz Rysunek 1.3 na stronie 17).



Rysunek 1.3. Wyniki testów biblioteki FatFS

Źródło: Na podstawie [9]

¹³ Odczyt animacji: 15 fps, 40 kB na jedną klatkę obrazu. Wymagana prędkość odczytu 600 kB/s

¹⁴ (ang. Direct Memory Access – DMA)

1.2.5 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 została napisana autorska biblioteka graficzna. 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 12 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 58).

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 omówiony zostanie sposób przystosowywania źródła do konkretnej płyty drukowanej. Kolejnym etapem będzie prezentacja warstwy abstrakcji sprzętu (HAL¹⁵) oraz omówienie najważniejszych z obecnie dostępnych sterowników. Po omówieniu sterowników zaprezentowane zostaną zaimplementowane biblioteki.

2.1 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, mniej znaczące, zostały pominięte).

¹⁵ (ang. Hardware Abstraction Layer – HAL)

Tabela 2.1. Struktura katalogów projektu

Źródło: Opracowanie własne

Ścieżka katalogu	Zawartość
<i>./app</i>	Katalog zarezerwowany dla plików dodatkowych aplikacji. Aktualnie znajduje się w nim jedynie kod źródłowy odtwarzacza audio.
<i>./boards</i>	Katalog przeznaczony na pliki konfiguracyjne projektu dla konkretnych płyt PCB. Dokładny opis znajduje się w rozdziale 2.3.1 na stronie 24.
<i>./hal</i>	Katalog zarezerwowany na pliki warstwy abstrakcji sprzętu
<i>./hal/hld</i>	W katalogu tym znajdują się pliki nagłówkowe definiujące modele struktur opisujących sterowniki różnych typów (patrz rozdział 2.3.22.3 na stronie 25).
<i>./hal/lld</i>	Pliki źródłowe oraz nagłówkowe konkretnych sterowników sprzętu.
<i>./hal/lld/platforms</i>	Katalog na sterowniki zależne od wybranej platformy sprzętowej, czyli procesora lub projektu płyty PCB.
<i>./lib</i>	Katalog zawiera źródła wszystkich bibliotek systemowych. Każda z nich posiada swój podkatalog. Pliki bibliotek wspólnych nie posiadają podkatalogu.
<i>./os</i>	Pliki źródłowe systemu FreeRTOS.

2.2 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`, która odpowiedzialna jest za włączenie wywłaszczania. Oczywiście w naszym przypadku została ustawiona na wartość „1” co oznacza włączenie danej funkcji. Włączenie wywłaszczania oznacza, że wykonywany aktualnie wątek może

zostać w dowolnej chwili przerwany a czas procesora zostanie przekazany do innego wątku. Czas procesora zostanie wykorzystany przez wątek, który aktualnie ma najwyższy priorytet a jego flaga gotowości jest ustawiona. W systemach bez wywłaszczania wykonywanie wątku zostanie przerwane jedynie w sytuacji, gdy ten wyrazi na to zgodę (wykona tak zwaną procedurę „yield” – nazwa zależna od systemu).

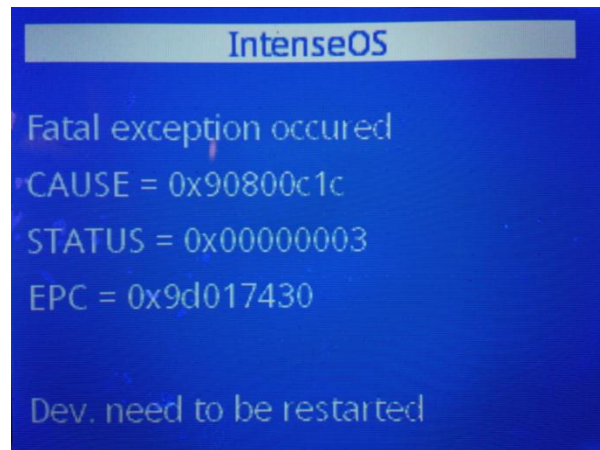
Kolejnym interesującym nas parametrem konfiguracyjnym, jest `configTICK_RATE_HZ`. Stała ta definiuje, z jaką częstotliwością mają być wykonywane kroki planisty. Dokładniej oznacza to czas, na jaki zostanie ustawiony zegar, który po przepełnieniu zgłasza przerwanie, w którym wykonywany jest jeden krok planisty. Planista podczas wykonywania kroku może zmienić kontekst procesora, czyli zmienić wykonywany wątek. W naszym przypadku wartość to została ustawiona na 1000 Hz. 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 80 Mhz 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, ale 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, do którego trafiają wszystkie zmienne zadeklarowane dynamicznie. Wielkość ta została ustawiona na 64 kB, czyli dokładnie połowę dostępnej pamięci.

Przy okazji omawiania wielkości sterty warto wspomnieć o wybranym modelu pamięci. FreeRTOS oferuje aż cztery modele do wyboru. Pierwszy pozwala jedynie alokować pamięć. Raz zadeklarowanej pamięci nie można już zwolnić. 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 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 EPC¹⁶ 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.



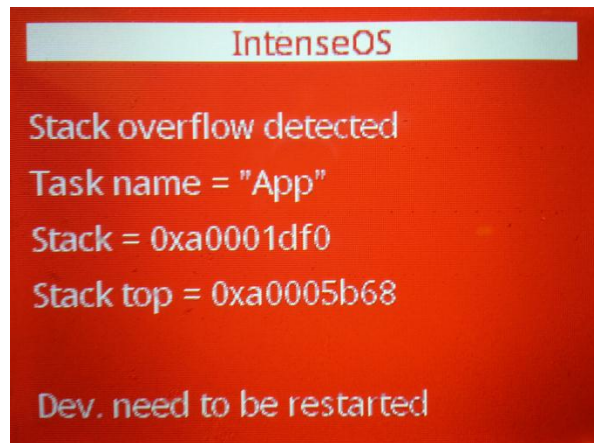
Rysunek 2.1. Wyjątek procesora.

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.

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

¹⁶ (ang. Exception Program Counter – EPC)



Rysunek 2.2. Przepelnienie stosu.

Źródło: Opracowanie własne

2.3 Warstwa abstrakcji sprzętu (HAL)

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 uruchamiania 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 niskopoziomowych 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 wykorzystania na potrzeby 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 zewnętrzny 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 `lldXXXXAttach()` zostanie opisane w dalszej części pracy (patrz rozdział 2.3.3 na stronie 28).

2.3.1 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()
{
    // Inicjalizacja płyty
    // ...
}

```

2.3.2 HLD

Moduł HLD¹⁷ warstwy abstrakcji sprzętu odpowiedzialny jest za zarządzanie sterownikami zainstalowanymi w systemie oraz definicję typów sterowników, które mogą zostać do niego dodane. Została tutaj zdefiniowana struktura bazowa, opisująca każdy sterownik (patrz Kod źródłowy 2.4 na stronie 26). Struktury opisujące konkretne typy

¹⁷ (ang. High Level Driver – HLD)

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 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` możemy 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 dynamicznie dodawać 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 27). 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, dodatkowy 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 28). 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_MEMORY;
}

//...

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 28).

2.3.3 LLD

Moduł LLD¹⁸ jest właściwie jedynie katalogiem przygotowanym do przechowywania sterowników niskiego poziomu, czyli sterowników 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 zostały zaimplementowane. W każdym z nich znajdzie się również opis struktury „dziedzicznej” określającej dany typ sterownika z modułu HLD.

¹⁸ (ang. Low Level Driver – LLD)

2.3.3.1 UART

Sterownik urządzenia portu transmisji szeregowej (UART¹⁹) 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 szeregowego

```
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 29). 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,
```

¹⁹ (ang. Universal Asynchronous Receiver and Transmitter – UART)

```
    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 przerywania urządzenia itd. Wskaźnik na taką strukturę z wpisanymi oczekiwanymi wartościami przekazujemy do funkcji uruchamiającej urządzenie (`open`).

Kolejne dwa pola są kolejkami FIFO²⁰ 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 (`attach`) jest wskaźnik na funkcję dodającą sterownik urządzenia do listy sterowników systemu. 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 50) o wystąpieniu zdarzenia.

Opisy funkcji takich jak `attach`, `open` oraz `close` będą pomijane w dalszej części pracy chyba, że specyficzna konstrukcja sterownika będzie tego wymagać.

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 przerwania od modułu UART zarówno do wysyłania jak i odbierania danych.

²⁰ (ang. First In First Out – FIFO)

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 odbieranych/wysyłanych 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.

2.3.3.2 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 31).

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 należy użyć funkcji `read`, która jako drugi parametr, przyjmuje 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 jaki 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 przepisane zostają dane ze sprzętowych buforów mikrokontrolera do tablicy która będzie je przechowywała do czasu zgłoszenia kolejnego przerwania (patrz Kod źródłowy 2.11 na stronie 32). Rozwiązanie to uprościło funkcję odczytującą wartość danego kanału do minimum (patrz Kod źródłowy 2.12 na stronie 32).

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”.

2.3.3.3 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 33) 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 dekodera. Jako jedyny parametr konfiguracyjny podajemy wskaźnik do funkcji dekodera (patrz Kod źródłowy 2.14 na stronie 34). 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 IR

```
struct hldIrConfig
{
    UINT32 (*decode) (UINT32 pLastPulseTime,
                     union hldIrFlags *pFlags, UINT32 pTicks);
};
```

Obecnie każde urządzenie może mieć przypisany tylko jeden dekodery. W planach jest rozszerzenie tej funkcjonalności do nieograniczonej liczby dekodery. 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. Nadaje 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 odbierany jest sygnał. W procedurze obsługi przerwania przełączane jest zbocze na które zostanie zgłoszone kolejne przerwanie (patrz Kod źródłowy 2.16 na stronie 35). 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 Kod źródłowy 2.16 na stronie 35).

Kod źródłowy 2.15. Inicjalizacja urządzenia „Input Capture”

```
// 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”.

2.3.3.4 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 36).

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, bez tworzenia tym samym 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 jednocześnie, bez możliwości miksowania kilku dźwięków

Domyślnym trybem jest `AM_ONECHANNEL`, pozwala on 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 przerywania 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 przerywania 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, przerywanie 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 linii 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”.

2.3.3.5 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 Kod źródłowy 2.18 na stronie 38). 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`”.

2.3.3.6 LCD

Warstwa abstrakcji sprzętu dla sterowników wyświetlaczy oraz układów graficznych posiada znacznie więcej funkcji aniżeli omawiane do tej pory sterowniki. Znajdują się tutaj funkcje odpowiadające specyficznym możliwościom układów graficznych. W obecnym stopniu zaawansowania projektu, są to funkcje służące do rysowania podstawowych elementów takich jak prostokąt czy piksel (patrz Kod źródłowy 2.19 na stronie 40). Funkcje te będą przybywały razem z rozwojem, projektowanego równoległego, 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`.

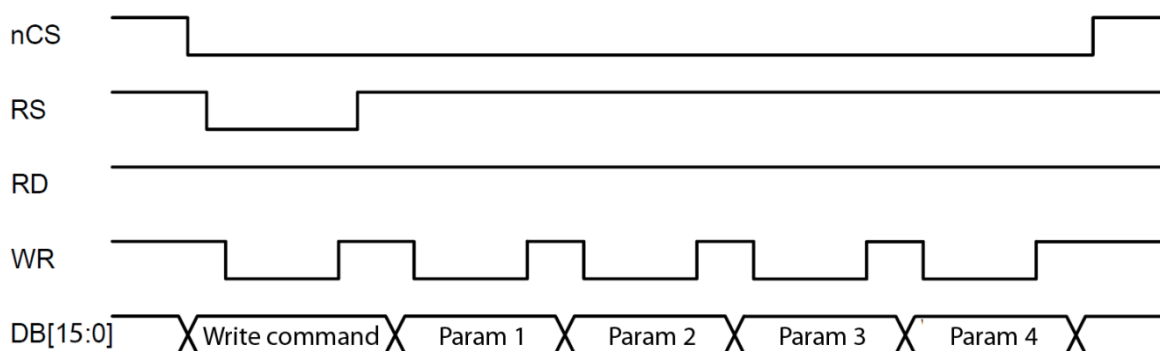
Funkcja `fill` wypełnia zadany prostokąt aktualnym kolorem. Prostokąt ten definiujemy poprzez podanie współrzędnych lewego górnego (`pX1`, `pY1`) oraz prawego dolnego (`pX2`, `pY2`) wierzchołka.

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 urządzenie ma współpracować.

Komunikacja z oby dwoma układami nawiązywana jest przez 16-bitowy interfejs i80 (patrz Rysunek 2.3 na stornie 41). 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 używana jest 16-bitowa magistrala 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 będzie przełączany programowo.

Dzięki zastosowaniu sygnałów CS istnieje teoretyczna możliwość obsługiwanie 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” (ang. touchpad) z możliwością wyświetlania na nim dodatkowych informacji. Mogłyby 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 przy wykorzystaniu 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 od podstaw, możliwy był 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 (aktualnie aplikacja testowa uruchamiana jest na małym wyświetlaczu). 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 <code>setColor</code> z sterownika. Parametr 1: Kolor w formacie RGB565
Set Pixel („sp”) – 0x7370 Ilość parametrów: 2	Rysuje piksel aktualnie wybranym kolorem. Odpowiednik funkcji <code>drawPixel</code> 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 <code>fill</code> 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 <code>fill</code> że polecenie dotyczy obszaru całego ekranu i wysłać polecenie Clear Screen zamiast Fill Rect.
Color Expansion („ce”) – 0x6365 Ilość parametrów: $(\frac{w \cdot h}{8} + 4)$	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. <div data-bbox="949 672 1236 1041" data-label="Image"> </div> <p>Parametr 1: Pozycja X Parametr 2: Pozycja Y Parametr 3: Szerokość mapy (w) Parametr 4: Wysokość mapy (h) Parametry od 5 do $(\frac{w \cdot h}{8} + 4)$: Określają wartości kolejnych pikseli mapy bitowej. Po 8 pikseli na parametr.</p>
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”.

2.3.3.7 TOUCH

Ostatnim już sterownikiem, który został przygotowany na potrzeby pracy, jest sterownik panelu dotykowego. Jest to jedyny w projekcie sterownik niezależny 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 Kod źródłowy 2.21 na stronie 44). 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 50). 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 hldTouchEvent
{
    struct hldDevice head;
    struct hldTouchConfig config;

    retcode (*attach)(struct hldTouchConfig *pCfg);
    retcode (*open)(struct hldTouchEvent *pTouchDev);
    retcode (*close)(struct hldTouchEvent *pTouchDev);
}

```

```

retcode (*read)(struct hldTouchEvent *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 tego 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 zmieniają się co $\frac{1}{6}$ czasu podanego w konfiguracji sterownika.

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

2.4 Biblioteki

Ze względu na znacząco objętość oraz stopień skomplikowania kodu bibliotek systemowych, zamiast opisywać w tym rozdziale ich wewnętrzną budowę oraz sposób działania, zaprezentowany zostanie sposób 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, najważniejsze jego fragmenty posiadają komentarze. Kody źródłowe bibliotek znajdują się na dołączonej płycie CD w katalogu „./source/lib”.

2.4.1 Biblioteka standardowa

Jako biblioteka standardowa języka C wykorzystana została biblioteka dedykowana dla procesorów PIC32 przygotowana przez firmę Microchip. Producenci procesorów zazwyczaj dostarczają zestawy podstawowych funkcji bibliotecznych przystosowanych do ich produktów, więc nawet w przypadku chęci przeniesienia systemu na inny procesor nie stanowiłoby 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.

2.4.1.1 Filtr cyfrowy

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

Próbki do filtra dodajemy używając polecenia `filterAddSample`. Pierwszy parametr to wskaźnik na strukturę opisującą filtr, drugi jest wartością dodawanej próbki. W celu usunięcia wszystkich próbek z filtra wywołujemy funkcję `filterResetSamples`.

Po dodaniu do filtra 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 powodowałoby, że każde dotknięcie byłoby interpretowane, jako przeciągnięcie rysikiem po ekranie, od rogu do rzeczywistej pozycji rysika. Dlatego

przygotowana została funkcja umożliwiająca sprawdzenie czy całe okno filtru zostało już wypełnione 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`.

Przewidziane zostały dwie możliwości pobrania 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 47.

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”.

2.4.1.2 LOG

Moduł o nazwie „LOG” służy do umożliwienia śledzenia wykonywania kodu poprzez dowolne urządzenie znakowe. Udostępnia on takie polecenia jak `LOG`, `ERROR`, `DONE` oraz `DEBUG`. Polecenia te przyjmują składnię 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 48). 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);
}
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”.

2.4.2 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 49). Są to dokładnie takie parametry, jakie przyjmuje główna funkcja programu (`main`) w systemie Linux.

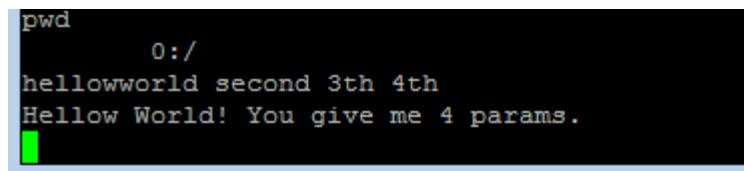
Kod źródłowy 2.24. Przykładowa funkcja konsolowa

```
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 (włącznie z nazwą polecenia). 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("helloworld", &consoleHellowWorld);
```



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

Źródło: Opracowanie własne

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

2.4.3 System plików

Wcześniej stwierdzono, że do obsługi systemu plików posłuży nam biblioteka „*FatFS*” [9]. Nie jest ona zgodna ze standardem POSIX. Polecenia 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.

2.4.4 Audio

Wcześniej w pracy zostało wspomniane, że biblioteka audio posiada kilka trybów pracy. Aktualnie zaimplementowane tryby to generowanie sygnału sinusoidalnego oraz odtwarzanie pojedynczego dźwięku. Opisany został drugi z tych trybów, nazywany 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 `(audiolchGetCurrentSample()/audiolchGetSamplesCount())*100` zwracało procentowy postęp odtwarzania.

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

1. `audiolchGetBits` – ilość bitów na próbkę
2. `audiolchGetChannels` – ilość kanałów
3. `audiolchGetSampleRate` – częstotliwość próbkowania

Kod źródłowy 2.26. Przykład wykorzystania biblioteki audio

```
UINT32 sr, ch, b;

// Odtwórz dźwięk snd.wav (tryb asynchroniczny)
if (audiolchPlaySound("snd.wav", SND_ASYNC) == SUCCESS)
{
    // Pobierz informacje o dźwięku
    sr = audiolchGetSampleRate();
    ch = audiolchGetChannels();
    b = audiolchGetBits();

    LOG("Playing: %dkHz, %dch, %dbit", sr/1000, ch, b);
    vTaskDelay(1000*10); // Czekaaj 10s
    audiolchStopSound(); // 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 `audiolchGetSample`. 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 z 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, gdzie opóźnienia nie są akceptowalne, 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 dekodów nowych formatów plików wymaga minimalnej ingerencji w kod biblioteki. Został przygotowany pewien schemat funkcji które powinien implementować dekodery aby mógł zostać wykorzystany przez naszą bibliotekę.

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

2.4.5 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 2.4.6 na stronie 54). 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 tę warstwę sterownik musiał by przeszukiwać wszystkich słuchaczy (ang. listener), sprawdzać który z nich jest zainteresowany zdarzeniem które ma zostać wysłane i ew. je wysłać. 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 menadżera wejść o zaistniałym zdarzeniu, należy wywołać funkcję `inputEventNotify`, lub jeżeli powiadamy 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 2.27 na stronie 53).

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
 - a. `EVENT_TOUCH_DOWN` – wciśnięcie rysika
 - b. `EVENT_TOUCH_UP` – zabranie rysika z powierzchni panelu dotykowego
 - c. `EVENT_TOUCH_MOVE` – przesunięcie rysika
2. `EVENT_KEY` – zdarzenie od klawiatury
 - a. `EVENT_KEY_DOWN` – wciśnięcie przycisku
 - b. `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
 - a. `EVENT_CHAR_GET` – odebranie znaku od urządzenia
4. `EVENT_RCU` – zdarzenie z odbiornika podczerwieni
 - a. `EVENT_RCU_CODE_RECEIVED` – pomyślne odebranie kodu od pilota
5. `EVENT_DISK` – zdarzenie zamontowania lub odmontowania dysku
 - a. `EVENT_DISK_INSERTED` – zamontowanie dysku
 - b. `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 pól struktury. Wszystkie potrzebne do zgłoszenia zdarzenia informacje przekazywane są przez parametry funkcji zgłaszającej. Kod źródłowy 2.28 przedstawia przykładową funkcję skrótową dla zgłoszenia zdarzenia od panelu dotykowego.

Kod źródłowy 2.28. Funkcja zgłaszająca 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*”.

2.4.6 User

Podczas tworzenia nowej aplikacji konieczne jest stworzenie 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 2.29 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 API²¹ 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 wspomnianego wcześniej 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 58).

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, opisane zostaną 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

²¹ (ang. Application Programming Interface – API)

komunikatów zapewniać płynność animacji na wyświetlaczu, zastosowanie funkcji `msgListenerGet` byłoby niewskazane. Do tego celu przygotowana została funkcja `msgListenerPeek` która nie zatrzymuje wykonywania programu w przypadku braku komunikatów oczekujących na obsługę. Kod źródłowy 2.30 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 to `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. Konkretnie komunikaty oraz parametry jakie trzeba z nimi przekazywać zostaną dokładniej opisane podczas omawiania biblioteki interfejsu graficznego użytkownika (patrz rozdział 2.4.7 na stronie 58). 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 2.31 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. Takie właśnie było założenie, aby aplikacje na proste urządzenia wbudowane pisało się tak samo łatwo jak dla systemu operacyjnego Windows – udało się to osiągnąć.

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

2.4.7 Graficzny interfejs użytkownika (GUI)

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 kontrolki takie jak okno główne, tekst statyczny, przycisk oraz pasek postępu.

2.4.7.1 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`, dla której jako parametr podajemy wskaźnik na okno które ma zostać aktywowane.

Aby stworzyć okno, trzeba 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. Kod źródłowy 2.32 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 2.33 na stronie 59). Parametry funkcji przedstawia Tabela 2.3. Dla wszystkich innych kontroltek istnieje jedna, wspólna funkcja o zbliżonej budowie (patrz Kod źródłowy 2.34 na stronie 61).

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 próbuje 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 zaimplementowane, 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 kontroltek za pomocą 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 kontroltek 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.

2.4.7.2 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 kontroltek 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 wykonania tego polecenia będzie pojawienie się napisu w obszarze okna podanego jako rodzic (patrz Rysunek 2.5 na stronie 61).

Kod źródłowy 2.34. Przykład tworzenia tekstu statycznego

```
stTitle = guiCreateWindow
(
    "statictext", // Klasa kontrolki
    "Przykładowy 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
);
```



Tekst Statyczny

Rysunek 2.5. Kontrolka „tekst statyczny”

Źródło: Opracowanie własne

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

Poprzez opisywane wcześniej funkcje `msgSend` oraz `msgPost`, do każdego z okien możemy wysłać komunikaty, które powodują ich określone zachowanie. Niektóre z komunikatów są uniwersalne i dotyczą wszystkich kontrollek. Kontrolka tekstu statycznego nie obsługuje żadnych dodatkowych komunikatów, reaguje tylko na komunikaty uniwersalne (patrz Tabela 2.4 na stronie 62).

Większość komunikatów jest zgodna z API systemu operacyjnego Windows. Każdy z komunikatów przyjmuje dokładnie takie same parametry i powoduje bardzo zbliżone zachowanie kontrollek.

Tabela 2.4. Lista komunikatów uniwersalnych

Źródło: Opracowanie własne

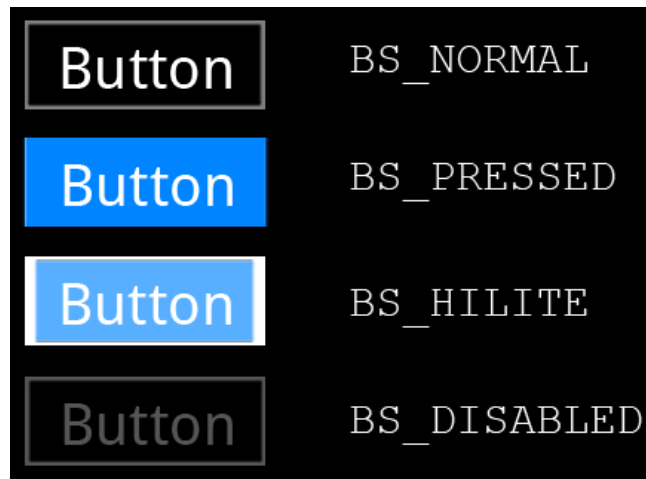
Komunikat	Parametr 1	Parametr 2
MSG_PAINT Przerysowanie kontrolki	Nie dotyczy	Nie dotyczy
MSG_ENABLE Włączenie lub wyłączenie kontrolki.	Wartość logiczna oznaczająca czy kontrolka ma zostać włączona (TRUE) czy wyłączona (FALSE).	Nie dotyczy
MSG_SETFONT Zmienia czcionkę którą wyświetlany jest tytuł kontrolki	Wskaźnik na czcionkę	Wartość logiczna określająca czy kontrolka ma zostać odrysowana (TRUE) bezpośrednio po zmianie
MSG_SETTEXT Zmiana tytułu kontrolki	Nie dotyczy	Wskaźnik na zmienną zawierającą tekst
MSG_GETTEXT Pobiera tytuł kontrolki.	Ilość znaków do skopiowania	Wskaźnik na bufor dla kopiowanych znaków
MSG_KEYDOWN Symulacja wciśnięcia klawisza	Kod wciśniętego klawisza	Nie dotyczy

2.4.7.3 Przycisk

Przycisk tworzymy w sposób identyczny jak tekst statyczny zmieniając tylko nazwę klasy na „button”. Po takiej operacji otrzymujemy w pełni funkcjonalny przycisk posiadający cztery stany (patrz Rysunek 2.6 na stronie 63):

1. BS_NORMAL – stan domyślny
2. BS_PRESSED – wciśnięty (po przytrzymaniu rysika)
3. BS_HILITE – podświetlony (po najechaniu kursorem)
4. BS_DISABLED – wyłączony

Przycisk obsługuje wszystkie uniwersalne komunikaty oraz komunikaty wysyłane przez system, powiadamiające kontrolkę o najechaniu na nią kursorem oraz o przytrzymaniu na niej rysika. Komunikaty te nie powinny być jednak wysyłane z poziomu aplikacji.



Rysunek 2.6. Kontrolka „przycisk”

Źródło: Opracowanie własne

W odpowiedzi na wciśnięcie, przycisk odsyła do okna „rodzica” komunikat MSG_COMMAND, który jako starsze szesnaście bitów pierwszego parametru przyjmuje kod wykonanej operacji (w tym przypadku BN_CLICKED) a jako młodsze 16 bitów identyfikator przycisku. Drugim parametrem jest wskaźnik na strukturę opisującą przycisk. Obsługę tego komunikatu w procedurze zdarzeniowej okna przedstawia Kod źródłowy 2.35.

Kod źródłowy 2.35. Obsługa wciśnięcia przycisku

```

UINT32 notify, id;
switch (pMsg)
{
    case MSG_COMMAND:
        notify = pParam1 >> 16;
        id = pParam1 & 0xFFFF;
        if (notify == BN_CLICKED)
        {
            switch(id)
            {
                case IDC_BTN_1: LOG("Przycisk 1"); break;
                case IDC_BTN_2: LOG("Przycisk 2"); break;
            }
        }
        break;
    //...
}

```

Przykład ten również pokazuje na jak wysokiej warstwie abstrakcji pracujemy. Stworzenie dowolnego urządzenia, bazującego na tak przygotowanym systemie, staje się znacznie mniej skomplikowane, aniżeli tworzenie całej logiki działania systemu od podstaw.

2.4.7.4 Pasek postępu

Ostatnią zaimplementowaną aktualnie kontrolką jest pasek postępu. Stworzenie odtwarzacza audio wymagało jego zastosowania, jako pasek postępu odtwarzania. Musiał on umożliwiać zmianę postępu odtwarzania poprzez przytrzymanie rysika na pasku oraz przeciągnięcie w jedną ze stron. Po zwolnieniu rysika postęp odtwarzania ustawiany jest na wskazane miejsce. W kodzie źródłowym funkcjonalność ta występuje pod sformułowaniem „*dragging*” i możemy ją włączyć, poprzez podanie odpowiedniego stylu, podczas jego tworzenia. Lista stylów obsługiwanych przez pasek postępu (style można łączyć poprzez sumę logiczną):

1. `PBS_ALLOW_DRAGGING` – pasek obsługujący przeciąganie.
2. `PBS_NOTIFY` – powiadamia okno rodzica o wewnętrznych zmianach stanów, np. rozpoczęcie przeciągania (`PBN_DRAGGINGSTART`).
3. `PBS_VERTICAL` – pasek pionowy.

Pasek postępu z systemu „Windows” nie posiada wbudowanej funkcji przeciągania – oczywiście można ją stworzyć samodzielnie z poziomu aplikacji.

Klasa okna paska postępu nosi nazwę „*progressbar*”. Po stworzeniu ze stylem `PBS_NOTIFY` pasek postępu wysyła komunikaty `MSG_COMMAND`, o takich samych parametrach jak przycisk, do okna rodzica. Możliwe kody operacji:

1. `PBN_REACHMAX` – pasek osiągnął pozycję maksymalną.
2. `PBN_REACHMIN` – pasek osiągnął pozycję minimalną.
3. `PBN_CLICKED` – użytkownik wcisnął rysik na pasku.
4. `PBN_CHANGED` – stan paska został zmieniony (tylko podczas przeciągania).
5. `PBN_DRAGGINGSTART` – rozpoczęto przeciąganie.
6. `PBN_DRAGGINGEND` – zakończono przeciąganie.

Powiadomienia obsługujemy w sposób analogiczny do obsługi wcisnięcia przycisku (patrz Kod źródłowy 2.35 na stronie 63).

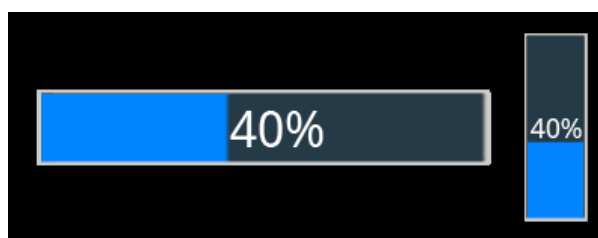
Pasek postępu obsługuje dodatkowe komunikaty, pozwalające na kontrolę jego pozycji oraz ustawienie zakresu wartości. Listę komunikatów przedstawia Tabela 2.5.

Tabela 2.5. Komunikaty obsługiwane przez pasek postępu

Źródło: Opracowanie własne

Komunikat	Parametr 1	Parametr 2
PBM_SETPOS Ustawienie pozycji	Pożądana pozycja bezwzględna. Jeżeli wartość poza zakresem, zostanie ograniczona do wartości maksymalnej lub minimalnej	Nie dotyczy
PBM_GETPOS Pobranie aktualnej pozycji	Wskaźnik na zmienną, w której zapisana zostanie pobrana pozycja	Nie dotyczy
PBM_DELTAPOS Względna zmiana pozycji	Względna zmiana pozycji. Podanie wartości ujemnej spowoduje zmniejszenie pozycji o podaną wartość.	Nie dotyczy
PBM_SETRANGE Ustawienie zakresów (pozycja minimalna oraz maksymalna)	Wartość minimalna	Wartość maksymalna

Rysunek 2.7 przedstawia kontrolkę paska postępu w wersji poziomej oraz pionowej.



Rysunek 2.7. Kontrolka „pasek postępu”

Źródło: Opracowanie własne

Najlepszym przykładem wykorzystania wszystkich zaimplementowanych kontrollek jest kod źródłowy aplikacji demonstracyjnej, znajdujący się na dołączonej płycie CD, pod ścieżką: „./source/app/windows/player.c”.

ROZDZIAŁ 3

Aplikacja demonstracyjna

Na potrzeby pracy wykonana została aplikacja demonstracyjna, którą jest odtwarzacz audio. Aplikacja ta demonstruje niektóre z możliwości zaprojektowanego systemu.

Niniejszy rozdział ma na celu przedstawienie założeń jakie miała ona spełniać oraz prezentację wyników pracy. Omówiony zostanie interfejs użytkownika oraz obsługa aplikacji przez pilot zdalnego sterownia.

3.1 Założenia

Gdy przyjrzymy się dostępnym w Internecie aplikacją odtwarzaczy audio dla płyt ewaluacyjnych zbudowanych na podstawie prostych mikrokontrolerów, szybko okazuje się, że żadna z nich nie jest w pełni funkcjonalnym odtwarzaczem. Często aplikacja ogranicza się do wybrania pliku który będzie odtwarzany albo nawet odtwarzany jest tylko pierwszy plik znaleziony na karcie SD. Głównym założeniem naszej aplikacji demonstracyjnej było dostarczenie w pełni funkcjonalnego odtwarzacza audio, z przyjaznym dla użytkownika interfejsem graficznym.

Oczywistym do spełnienia warunkiem jest możliwość decydowania o tym czy dźwięk ma być w danej chwili odtwarzany czy nie. Aby było to możliwe konieczne jest umieszczenie przycisku pauzy. Przycisk ten powinien zmieniać swoją funkcję, a co za tym idzie swój tytuł, na „play”. Do rozwiązania pozostaje kwestia domyślnego stanu po uruchomieniu urządzenia. Aby uniknąć możliwego zaskoczenia użytkownika, podjęta została decyzja, że domyślnie po włączeniu urządzenia, dźwięk nie będzie odtwarzany. Do czasu wciśnięcia przycisku „play”, na wyświetlaczu pokazany będzie komunikat zachęcający do jego wciśnięcia.

Kolejną funkcją którą musi posiadać każdy odtwarzacz audio jest możliwość zmiany aktualnie odtwarzanego utworu. Tutaj można zastosować dwie możliwości.

Jedną z nich jest zaprojektowanie okna zawierającego listę plików które znajdują się na dołączonej karcie SD, przez które będziemy mieli możliwość dodania ich do listy odtwarzania. Drugi sposób jest znacznie prostszy. Polega na udostępnieniu użytkownikowi przycisków służących do przełączania utworów na następny oraz poprzedni. Oczywiście pierwszy ze sposobów stwarza więcej możliwości doboru odtwarzanego materiału - został on zatem uznany jako lepszy. Niestety tego założenia nie udało się zrealizować z powodu ograniczeń czasowych. Jako namiastkę, udało się wykonać przycisk przełączający tryby odtwarzania. Dostępne są trzy tryby: odtwarzanie kolejnych utworów, odtwarzanie losowe oraz powtarzanie jednego utworu. Przycisk poprzedniego utworu, jeżeli postęp odtwarzania jest większy niż 10 sekund, powinien pełnić funkcję powrotu do początku utworu. Rozwiązanie to jest również dobrze znane z typowych odtwarzaczy.

Powinna istnieć możliwość przewijania utworów. Aby nie mnożyć ilości dostępnych przycisków zdecydowano się, na znane z odtwarzaczy komputerowych, przeciąganie paska postępu w dowolne miejsce. Powoduje to przejście do odtwarzania utworu od wybranego miejsca.

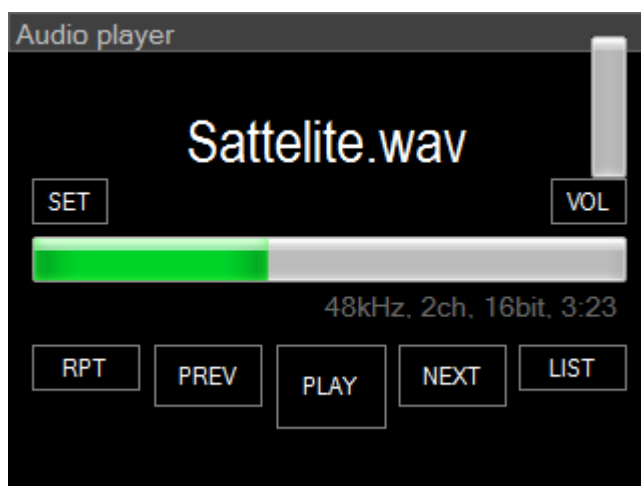
Każdy odtwarzacz muzyczny powinien mieć możliwość regulacji głośności. Może to być regulacja skokowa ograniczona do kilkunastu kroków, lub regulacja procentowa. Zastosowana zostanie dokładna regulacja procentowa. Ponownie pojawia się pytanie jaka wartość głośności powinna zostać ustalona po włączeniu urządzenia. Aby uniknąć możliwego uszkodzenia słuchu dla wartości wysokich, zastosowana została dokładnie połowa skali głośności. Ze względu na logarytmiczną skalę regulacji, zastosowaną w wykorzystywanym przetworniku DAC, jest to wartość bezpieczna.

Odtwarzacz muzyczny powinien prezentować nazwę odtwarzanego aktualnie utworu oraz w miarę możliwości, część informacji technicznych takich jak na przykład częstotliwość próbkowania. Jako że pliki WAV nie posiadają dodatkowych informacji o wykonawcy utworu oraz jego tytule – wyświetlana będzie nazwa odtwarzanego pliku.

W założeniach odtwarzacza znalazło się również odtwarzanie plików w formacie FLAC. Niestety z powodu braku czasu nie udało się zrealizować tego założenia. Nadal widnieje ono w planach rozwoju projektu.

3.2 Interfejs użytkownika

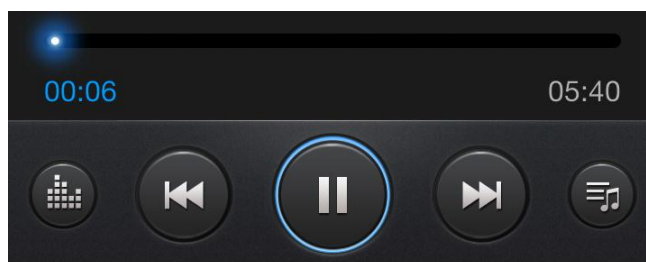
Projekt interfejsu użytkownika został wykonany w programie „Microsoft Visual C++ 2010 Express” (patrz Rysunek 3.1 na stronie 68). Narzędzie to pozwala zaplanować układ kontrolek na oknie, przeciągając je z paska narzędzi. Po odpowiednim ułożeniu elementów można było odczytać ich pozycje, po czym przenieść je do projektowanej aplikacji.



Rysunek 3.1. Projekt interfejsu użytkownika

Źródło: Opracowanie własne

Pionowy pasek postępu służący do regulacji głośności pojawia się po wciśnięciu przycisku „VOL” i znika zaraz po wybraniu, przez użytkownika, pożądanej wartości. Inspiracją do ułożenia przycisków, był układ z odtwarzacza audio systemu operacyjnego „Android” w wersji 4.1.2 (patrz Rysunek 3.2 na stronie 68).



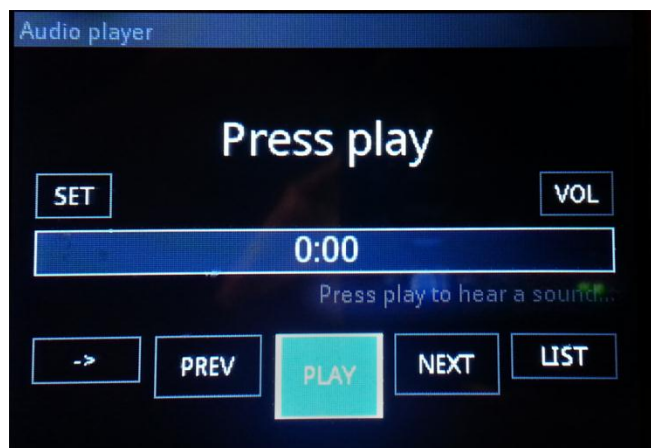
Rysunek 3.2. Odtwarzacz audio systemu „Android”

Źródło: Zrzut ekranu z urządzenia „Samsung Galaxy S III”

Przycisk zatytułowany „SET” został przygotowany na przyszłość. Po jego wciśnięciu użytkownik miał zobaczyć listę ustawień. Przycisk „LIST” w założeniach miał prowadzić do listy dostępnych utworów. Oba przyciski zostały wyłączone.

Rysunek 3.3 przedstawia stan urządzenia zaraz po włączeniu. Widoczny jest napis zachęający do wciśnięcia przycisku „PLAY”. Kolejne rysunki przedstawiają urządzenie podczas odtwarzania utworu muzycznego (patrz Rysunek 3.4 na stronie 69) oraz zdjęcie pracującego odtwarzacza (patrz Rysunek 3.5 na stronie 69), wraz z pilotem zdalnego sterowania.

Aplikacja demonstracyjna została przetestowana pod kątem stabilności. Testowane były różne kombinacje kolejności wciskania przycisków oraz jednoczesne wciskanie przycisków na pilocie i panelu dotykowym. Dodatkowo przeprowadzony został 24-godzinny test odtwarzania. Podczas testów aplikacja pracowała stabilnie, nie udało się jej zawiesić – testy wypadły pomyślnie.



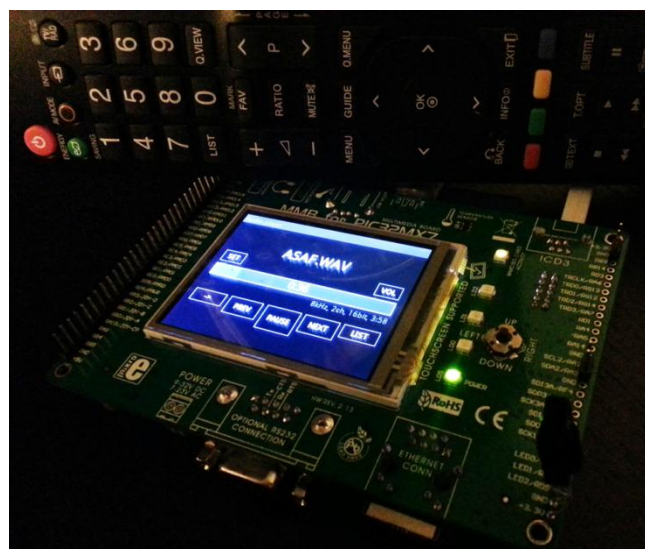
Rysunek 3.3. Stan aplikacji po włączeniu urządzenia

Źródło: Opracowanie własne



Rysunek 3.4. Odtwarzanie utworu

Źródło: Opracowanie własne



Rysunek 3.5. Pracujące urządzenie

Źródło: Opracowanie własne

Zakończenie

Celem pracy była nie tylko realizacja projektu odtwarzacza audio, ale również stworzenie uniwersalnego systemu operacyjnego, dedykowanego dla prostych urządzeń wbudowanych. Pomimo pewnych jego braków i niedoskonałości, cel ten udało się zrealizować. Napisany kod na pewno nie jest idealny i posiada wiele błędów. Pomimo rozwijania projektu od ponad roku nie udało się zaimplementować wielu funkcji. Po części wynika to z faktu, że tego typu oprogramowanie tworzone jest przez grupy architektów oprogramowania, programistów, elektroników oraz projektantów grafiki, a nie przez jedną osobę, pełniącą wszystkie te funkcje jednocześnie.

Projekt będzie nadal rozwijany. Rozszerzona zostanie funkcjonalność odtwarzania plików muzycznych o kolejne formaty. Pierwszym formatem, nad którym prowadzone będą prace, będzie format FLAC. W planach rozwoju znalazło się również dodanie możliwości odtwarzania kilku dźwięków jednocześnie. Razem z kolegą projektującym układ graficzny FPGA chcielibyśmy rozwinąć jego funkcjonalność. W chwili pisania tej pracy, rozkazy, które potrafi zinterpretować projektowany układ, nie są wystarczające aby wyświetlić wszystkie elementy graficznego interfejsu użytkownika. Rozbudowy wymaga również domyślna baza dostępnych kontrolek w GUI systemu.

Udało się osiągnąć zakładany poziom abstrakcji podczas pisania aplikacji użytkowych. Jednym z założeń było, aby aplikacje pisało się równie łatwo jak w systemie operacyjnym Windows.

Zarówno podczas projektowania modułów, jak i pisania kodu, autorowi pracy udało się zdobyć wiele wiedzy na temat wewnętrznej budowy systemów operacyjnych do której na pewno nigdy by nie dotarł, gdyby nie potrzeba jej wykorzystania. Głównym źródłem wiedzy były kody źródłowe innych systemów operacyjnych oraz bibliotek.

Kod źródłowy projektu rozpowszechniany jest na zasadach licencji GNU GPL v3 [15] i jest dostępny do pobrania pod adresem: „<http://code.google.com/p/intensesx/>”. Kiedy osiągnie on odpowiednie stadium rozwoju, informacja o dostępności takiego oprogramowania rozpowszechniona zostanie na forach internetowych grupujących pasjonatów w tej dziedzinie. Jeżeli znajdą się osoby, które uznają projekt za godny uwagi, będzie to dla autora pracy niesamowita satysfakcja.

DODATEK A

Zawartość dodatkowej płyty CD

Na dołączonej płycie CD znajduje się kompletny kod źródłowy systemu operacyjnego wraz z aplikacją testową. Aby skompilować oprogramowanie należy pobrać ze strony internetowej www.microchip.com/mplabx/ i zainstalować środowisko „*MPLAB X IDE*” (zalecana wersja 1.00), po czym otworzyć w nim projekt. W środowisku „*MPLAB X IDE*” podczas otwierania projektu wybieramy katalog „*./source*” z dołączonej płyty CD. W celu kompilacji z menu wybieramy „*Run -> Build Main Project*”.



PŁYTA CD

Bibliografia

- [1] Wikipedia, Otwarte oprogramowanie: http://pl.wikipedia.org/wiki/Open_source
- [2] Wikipedia, SoC: <http://pl.wikipedia.org/wiki/System-on-a-chip>
- [3] MikroElektronika: <http://www.mikroe.com/multimedia/pic32mx7>
- [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
- [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/~bouxhobza/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
- [15] GNU General Public License version 3, <http://www.gnu.org/licenses/gpl.html>, 2007

Spis tabel

Tabela 2.1. Struktura katalogów projektu	20
Tabela 2.2. Lista poleceń układu graficznego FPGA.....	42
Tabela 2.3. Lista parametrów funkcji <code>guiCreateMainWindow</code>	59
Tabela 2.4. Lista komunikatów uniwersalnych.....	62
Tabela 2.5. Komunikaty obsługiwane przez pasek postępu.....	65

Spis rysunków

Rysunek 1.1. Zdjęcie płyty "multimedia for PIC32MX7" z obu stron	12
Rysunek 1.2. Montaż odbiornika podczerwieni TSOP1138	13
Rysunek 1.3. Wyniki testów biblioteki FatFS	17
Rysunek 2.1. Wyjętek procesora.	22
Rysunek 2.2. Przepelnienie stosu.	23
Rysunek 2.3. 16-bitowy interfejs komunikacyjny i80.....	41
Rysunek 2.4. Zrzut ekranu konsoli, po wykonaniu polecenia „hellowworld”	49
Rysunek 2.5. Kontrolka „ <i>tekst statyczny</i> ”	61
Rysunek 2.6. Kontrolka „ <i>przycisk</i> ”	63
Rysunek 2.7. Kontrolka „ <i>pasek postępu</i> ”	65
Rysunek 3.1. Projekt interfejsu użytkownika	68
Rysunek 3.2. Odtwarzacz audio systemu „ <i>Android</i> ”	68
Rysunek 3.3. Stan aplikacji po włączeniu urządzenia	69
Rysunek 3.4. Odtwarzanie utworu	69
Rysunek 3.5. Pracujące urządzenie	69

Spis kodów źródłowych

Kod źródłowy 2.1. Wybór sterownika wyświetlacza w czasie kompilacji	24
Kod źródłowy 2.2. Konfiguracja sprzętowa sterownika wyświetlacza.....	24
Kod źródłowy 2.3. Prototyp funkcji <code>boardInit</code>	25
Kod źródłowy 2.4. Struktura bazowa dla sterowników	26
Kod źródłowy 2.5. Przykład wykorzystania funkcji <code>hldDeviceGetByType</code>	26
Kod źródłowy 2.6. Możliwe stany pracy urządzenia	27
Kod źródłowy 2.7. Wykorzystanie pola <code>priv</code> struktury <code>hldDevice</code>	28
Kod źródłowy 2.8. Warstwa abstrakcji dla sterowników portu szeregowego	29
Kod źródłowy 2.9. Warstwa abstrakcji dla sterowników urządzeń znakowych	29
Kod źródłowy 2.10. Warstwa abstrakcji dla sterowników przetworników ADC	31
Kod źródłowy 2.11. Funkcja obsługi przerwania przetwornika	32
Kod źródłowy 2.12. Funkcja odczytująca wartość podanego kanału przetwornika ..	32
Kod źródłowy 2.13. Warstwa abstrakcji dla sterownika odbiornika podczerwieni ...	33
Kod źródłowy 2.14. Parametry konfiguracyjne sterownika odbiornika IR	34
Kod źródłowy 2.15. Inicjalizacja urządzenia „Input Capture”	34
Kod źródłowy 2.16. Obsługa przerwania wykrycia zbocza.....	35
Kod źródłowy 2.17. Warstwa abstrakcji dla sterowników audio.....	36
Kod źródłowy 2.18. Warstwa abstrakcji dla sterowników dysków	38
Kod źródłowy 2.19. Warstwa abstrakcji dla sterowników układów graficznych	40
Kod źródłowy 2.20. Rysowanie piksela przez układ graficzny FPGA	43
Kod źródłowy 2.21. Warstwa abstrakcji sterownika panelu dotykowego	44
Kod źródłowy 2.22. Przykład użycia filtru cyfrowego	47
Kod źródłowy 2.23. Użycie modułu LOG	48
Kod źródłowy 2.24. Przykładowa funkcja konsolowa.....	49
Kod źródłowy 2.25. Przykład rejestracji funkcji w konsoli	49
Kod źródłowy 2.26. Przykład wykorzystania biblioteki audio	51
Kod źródłowy 2.27. Struktura opisująca zdarzenie.....	53
Kod źródłowy 2.28. Funkcja zgłaszająca zdarzenie od panelu dotykowego	54
Kod źródłowy 2.29. Przykładowa pętla komunikatów	55
Kod źródłowy 2.30. Pętla komunikatów dla gier.....	57
Kod źródłowy 2.31. Przykład użycia funkcji <code>msgSend</code> oraz <code>msgPost</code>	58
Kod źródłowy 2.32. Przykład definicji klasy okna	59
Kod źródłowy 2.33. Prototyp funkcji <code>guiCreateMainWindow</code>	59
Kod źródłowy 2.34. Przykład tworzenia tekstu statycznego.....	61
Kod źródłowy 2.35. Obsługa wcisnięcia przycisku	63