Akademia WSB

Wydział Zamiejscowy w Cieszynie
Kierunek studiów: Informatyka

PRACA DYPLOMOWA INŻYNIERSKA

Michał Czyż

Mikrokomputer rozszerzający możliwości liczników samochodowych typu „HIGH” stosowanych w samochodach marki BMW o oznaczeniach E34 oraz E32 konfigurowany za pomocą aplikacji mobilnej

Praca inżynierska
napisana pod kierunkiem
Dr hab. inż.
Bartłomieja Zielińskiego

Pracę przyjmuję, dnia……

…….
podpis promotora

CIESZYN 2023

Spis treści

Wstęp

Informatyzacja samochodów osobowych bardzo mocno posunęła się do przodu w ciągu ostatnich kilkunastu lat. Wraz z upowszechnieniem się technologii, konsumentom coraz bardziej zależy na wygodzie oraz dostępności rozrywki w trakcie korzystania z pojazdu. Pojazdy oferują teraz szybki dostęp do internetu, pozwalają na wysyłanie maili, odtwarzanie muzyki i filmów na żądanie, a także na nawigowanie oraz komunikację z innymi ludźmi.

Spowodowało to, że „Youngtimery", czyli samochody klasyczne pozostały w tyle i nie oferują nawet ułamka technologii oraz rozwiązań, które serwują nam samochody nowoczesne. Jednakże, pierwsze implementacje tych technologii wciąż są w tych samochodach i w poszczególnych przypadkach pozostawiają możliwość ich rozszerzenia o całkowicie nowe funkcje poprzez zewnętrzne rozwiązania. Pierwsze rozwiązania opierające się o komunikacje wielu mikrokomputerów zastosowane w tych samochodach nie posiadają żadnych mechanizmów szyfrowania lub zabezpieczeń, co bardzo ułatwia wszelkie modyfikacje oraz możliwości przystosowania do nowoczesnych rozwiązań. Rozwiązania te pozawalają zwiększyć funkcjonalność samochodu zachowując jego oryginalny wygląd, stylistykę okresową oraz wartość.

Termin "period correct" to pojęcie, które nabiera w ostatnim czasie szczególnego znaczenia w kontekście youngtimerów. Odnosi się on do zachowania klasycznego wyglądu samochodu oraz klimatu z danego okresu, przy jednoczesnym dążeniu do wprowadzenia funkcjonalności, która podniesie komfort użytkowania pojazdu. Dzięki temu, nasz samochód nie tylko będzie piękny, ale także będzie spełniał współczesne wymagania użytkowników.

Nie ma wątpliwości, że zachowanie oryginalnych cech klasycznych samochodów ma ogromną wartość. Nie tylko ze względów estetycznych, ale także ze względu na wartość kolekcjonerską, która może być znacznie wyższa dla samochodów z zachowaną oryginalną stylistyką. Jednocześnie, warto zdawać sobie sprawę z faktu, że dążenie do zachowania oryginalnego wyglądu samochodu nie musi oznaczać rezygnacji z dodatkowej funkcjonalności, która może znacznie zwiększyć komfort jazdy i użytkowania pojazdu.

Warto również pamiętać, że takie udoskonalanie samochodu ma wpływ na jego „długość życia” na drodze. Średni czas życia samochodu w Europie wschodniej wynosi około 28,4 roku według badań www.ncbi.nlm.nih.gov [1][1]. Poszczególne modyfikacje mogą pozytywnie wpłynąć na długość życia samochodu, przykładowo zamiast kupować nowy samochód który posiada grzane fotele oraz kierownice to wystarczy doposażyć w podobne elementy nasz, już sprawdzony samochód. Powoduje to brak potrzeby budowy nowego samochodu co finalnie przekłada się na mniejsze wytwarzanie gazów cieplarnianych. Takie przedsięwzięcia są w aktualnych czasach dosyć proste z powodu prostego dostępu do wideo poradników oraz oficjalnej dokumentacji samochodu.

Zważając na wszystkie powyższe pozytywne aspekty doposażania youngtimerów stwarzana jest możliwość budowy rozwiązania które rozszerza wspomniane już wcześniej wczesne wersje systemów multimedialnych o nowe funkcje zachowując styl danego auta. Samochody marki BMW o modelach E32 oraz E34 sumarycznie produkowane w latach 1986-1994 podsiadają odpowiedni wyświetlacz oraz manetki które można zaadoptować za pomocą specjalnie zaprojektowanych sterowników i interfejsów do współpracy z nowoczesnymi mikrokomputerami jak na przykład Arduino lub RaspberryPi. Praca inżynierka zakłada budowę platformy która pozwoli w uproszczony sposób pisać aplikacje wykorzystujące szesnastoznakowy wyświetlacz LCD instrumentów pomiarowych jako ekran aplikacji oraz używać manetek jako przycisków manipulujących aplikacjami. Całość będzie konfigurowana z poziomu strony internetowej gdzie będzie można zdefiniować ustawienia aktualnie wgranej aplikacji, mapowanie przycisków oraz kilka innych funkcji. Pierwsza testowa aplikacją na platformę jest prosta implementacja usługi Discord. Discord pozwala na głosowe połączenia z innymi użytkownikami oraz wiadomości tekstowe zarówno bezpośrednie jak i wysyłane na publicznych kanałach. Ta implementacja ma zaczytywać, kolejkować oraz następnie w sposób prosty i czytelny wyświetlać wiadomości które otrzymuje kierowca. Kierowca może zdefiniować które wiadomości chce widzieć oraz może je pominąć klikając odpowiedni przycisk. Budowa jaki zasada działania tytułowego rozwiązania została przedstawiona w dalszej części pracy.

[1] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7829067 data dostępu: 03.02.2023

Rozdział 1 Cel i zakres pracy

1.1 Cel pracy

Celem pracy jest zaprojektowanie oraz napisanie biblioteki pozwalającej na uproszczone pisanie oprogramowania współpracującego z ekranem LCD który znajduje się w wspomnianych w tytule zegarach samochodowych oraz prostą aplikacje celem zademonstrowania możliwości biblioteki którą można konfigurować za pomocą aplikacji mobilnej. Następnie należy zbudować mikrokomputer oparty na RaspberryPi na którym jest możliwość uruchomienia wcześniej napisanego oprogramowania. Komputer ma posiadać odpowiednie złącza umożliwiające bezinwazyjne podłączenie do zegarów samochodowych które zostaną wykorzystane przez oprogramowanie jako interfejs.

Motywacją do podjęcia tematu jest chęć rozbudowy samochodów klasycznych o nowe możliwości, na miarę XXI wieku oraz ułatwienie innym hobbistom i majsterkowiczom rozszerzenie możliwości swojego samochodu, jak i zdolności programistycznych podczas pisania aplikacji z wykorzystaniem tej biblioteki.

Końcowym efektem realizacji pracy jest mikrokomputer pozwalający na rozszerzenie możliwości zegarów typu „HIGH” w samochodach BMW E34 oraz E32 poprzez wyświetlanie dodatkowych informacji oraz manipulacje za pomocą manetek oraz przycisków na kierownicy aplikacją wgraną na wcześniej wspomniany komputer.

1.2 Zakres pracy

Zakres pracy obejmuje zaprojektowanie oraz zbudowanie aplikacji na mikrokomputer RaspberryPi, zaprojektowanie połączeń mikrokomputera z instalacją samochodową, zaprojektowanie obudowy i jej wydruk za pomocą drukarki 3D, montaż całości rozwiązania oraz testy z użyciem zegarów VDO 110.008.548/061.

1.3 Założenia

Projekt zakłada zaprojektowanie oraz napisanie oprogramowania sterującego mikrokomputerem Raspberry Pi, które za pomocą dodatkowego mikrokomputera Arduino będzie mogło używając UART, by wysyłać komendy wyświetlania tekstu na ekranie LCD zegarów samochodowych wspomnianych w tytule pracy. Dodatkowo zakłada się, że do Raspberry Pi zostanie podłączony fizyczny przycisk umożliwiający

sterowanie aplikacją, który w rzeczywistym zastosowaniu może zostać podmieniony na wcześniej wspomniany przycisk „BC” umieszczony na manetce w pojeździe. Kolejne założenia zakłada napisanie aplikacji webowej która pozwoli na konfiguracje aplikacji prze jej uruchomieniem oraz dostosowanie jej do potrzeb użytkownika. Samo urządzenie składa się z Raspberry Pi Zero 2 WH na którym zostanie uruchomia całość aplikacji, Arduino UNO które działa jako pośrednik komunikacji z ekranem LCD zegarów oraz odpowiednich przewodów.

Zbudowane urządzenie zostało odpowiednio podłączone oraz przetestowane w testowym środowisku składającym się z zegarów VDO 110.008.548/061 pochodzących z samochodu BMW E34, zasilacza 12V DIN30W12 POS POWER oraz płyt stykowych. Do testów użyto napisaną w ramach pracy inżynierskiej aplikację Discord która wraz z całym środowiskiem została uruchomiona na mikrokomputerze Raspberry Pi.

Jako główny język programowania projektu przyjęto PHP w wersji 7.4 na którym oparto większość kodu. Wspierany przez biblioteki „Symfony 5.4”, „Discord-php 7.3” oraz „Doctrine 2.4”. Wcześniej wspomniany język jest odpowiedzialny za pobieranie wybranych wiadomości z API serwisu Discord, ich zapisania do bazy, przefiltrowania oraz zakolejkowania do wyświetlenia na ekranie. Również zapewnia API dla aplikacji mobilnej za której pomocą możemy zapisać, modyfikować oraz sprawdzać konfigurację aplikacji. Sama aplikacja mobilna została napisana w HTML 5, CSS 3 z użyciem biblioteki Bootstrap oraz Javascript. Do napisania sterownika łączącego główną aplikację z Arduino napisano prosty skrypt w języku Python który odpowiada za komunikację z wcześniej wspomnianym mikrokomputerem za pomocą UART. Finalnie na Arduino oprogramowanie manipulujące wyświetlaczem LCD zegarów zostało napisane w języku C.

Mikrokomputery czyli Raspberry Pi oraz Arduino działają zasilane napięciem 3.3V oraz używają UART TTL. Tytułowe zegary są zasilane napięciem 12V oraz działają w oparciu o RS232, 12V. Konieczne jest użycie konwertera TTL do RS232 celem umożliwienia komunikacji mikrokomputerów z zegarami bez ryzyka ich uszkodzenia.

Urządzenie ma współpracować z wszystkimi zegarami typu „HIGH” znajdującymi się w samochodach BMW E34 oraz E32.

Ostatnim założeniem jest to, aby projekt miał konstrukcję modułową – możliwość pisania kolejnych aplikacji oraz rozszerzania możliwości projektu bez potrzeby jego całkowitej przebudowy. Umożliwiać to będzie odpowiednio zaprojektowana obudowa urządzenia oraz sam sposób projektu oraz budowy oprogramowania.

Rozdział 2 Oprogramowanie

2.1 Budowa

Główny komputer sterujący cały układem, Raspberry Pi działa w oparaciu o system Raspbian. Ma on zainstalowany dowolny serwer PHP oraz MySQL. Aplikację można podzielić na cztery moduły:

- Interfejs wejścia
- Rdzeń aplikacji
- Interfejs wyjścia
- Witryna ustawień

2.2 Interfejs wejścia

Interfejs wejścia jest modułem o który zbiera interakcję od użytkownika. Posiada on szereg klas które są sztywno zintegrowanie i nie zakłada się, że podczas pisania rdzenia aplikacji, o którym później, będzie on modyfikowany.

```
class Input
{
    private InputDriver $inputDriver;
    private InputMapper $inputMapper;

    public function __construct(
        InputDriver $inputDriver,
        MappingProvider $mappingProvider)
    {
        $this->inputDriver = $inputDriver;
        $this->inputMapper = InputMapper::createWithMapping(
                $mappingProvider->retrieveUserSetMapping()
        );
    }

    public function getInputIfAny(): ?InputCommand
    {
        $buttonPress = $this->inputDriver->readInput();
        if ($buttonPress === null) {
            return null;
        }

        return $this->inputMapper->mapButtonToCommand($buttonPress);
    }
}
```

Fot. 2.1. Ciało klasy Input, odpowiedzialnej za interfejs wejścia.

Źródło: src/ClusterManager/Input/Input.php

Klasa składa się z dwóch serwisów oraz jednej funkcji odpowiedzialnej za koordynowanie jej pracy z całą aplikacją. Co „tick” próbuje ona pobrać aktualnie wciśnięty przycisk przez użytkownika, lub jeśli nic takiego się nie dzieje to również zwraca odpowiednią informacją. Pracę tej klasy umożliwiają dwa serwisy.

InputMapper, czyli klasa odpowiedzialna za tłumaczenie poszczególnych akcji użytkownika na zdefiniowane przez niego samego komendy. Podczas uruchomienia pobiera ona konfigurację klawiszy które zdefiniował użytkownik na stronie internetowej aplikacji i na jej podstawie przekłada poszczególne sygnały na komendy zrozumiałe przez rdzeń aplikacji.

```
class InputMapper
{
    public const BC_STALK_BUTTON = 'BC_STALK_BUTTON';
    public const TELEPHONE_STALK_BUTTON = 'TELEPHONE_STALK_BUTTON';
    public const TELEPHONE_BUTTON = 'TELEPHONE_BUTTON';
    public const RT_BUTTON = 'RT_BUTTON';
    public const PLUS_BUTTON = 'PLUS_BUTTON';
    public const MINUS_BUTTON = 'MINUS_BUTTON';
    public const LEFT_BUTTON = 'LEFT_BUTTON';
    public const RIGHT_BUTTON = 'RIGHT_BUTTON';

    public const POSSIBLE_INPUTS = [
        self::BC_STALK_BUTTON,
        self::TELEPHONE_STALK_BUTTON,
        self::TELEPHONE_BUTTON,
        self::RT_BUTTON,
        self::PLUS_BUTTON,
        self::MINUS_BUTTON,
        self::LEFT_BUTTON,
        self::RIGHT_BUTTON
    ];

    private Mapping $currentMappingSet;
```

Fot. 2.2. Stałe oraz zmienne klasy InputMapper, odpowiedzialnej za mapowanie akcji.
Źródło: src/ClusterManager/Input/Mapper/InputMapper.php

Klasa posiada spis wszystkich możliwych manipulatorów które może użyć użytkownik do wprowadzenia zmian w aplikacji podczas jej działania. Wszystkie klasy w aplikacji, jeśli muszą zaczerpnąć definicję przycisków to odwołują się do tej klasy.

```
/**
 * @throws UnknownButtonException
 */
public function mapButtonToCommand(ButtonPress $button): InputCommand
{
    if (!isset($this->currentMappingSet)) {
        $this->currentMappingSet = Mapping::getDefaultMapping();
    }

    if ($this->isMapped($button)) {
        $command = $this->currentMappingSet-
>getButtonToCommandMap()[(string)$button];
        return InputCommand::createWithCommand($command);
    }

    throw UnknownButtonException::createWithUnknownButton($button);
}
```

Fot. 2.3. Funkcja mapButtonToCommand klasy InputMapper.

Źródło: src/ClusterManager/Input/InputMapper.php

Główna funkcja klasy czyli „mapButtonToCommand" polega na zwykłym sprawdzeniu czy w aktualnie załadowanym mapowaniu występuje akcja odpowiadająca wciśniętemu przez użytkownika przyciskowi. Jeśli nie występuje to zostaje rzucony wyjątek który jest obsłużony w dalszej części aplikacji.

Drugi serwis klasy Input to InputDriver. Input driver w porównaniu do input mapper to interfejs. Klasa Input zakłada różne instancje klasy Input Driver, o których w dalszej części będzie mowa jako o sterownikach. Sterownik pełni jedną funkcję, czyli zbiera z zdefiniowanego przez swoją budowę źródła informacje o stanie przycisków i ją zwraca. Ciało interfejsu wygląda następująco:

```
interface InputDriver
{
    public function readInput(): ?ButtonPress;
}
```

Fot. 2.4. Ciało interfejsu InputDriver. Dziedziczą po nim wszystkie sterowniki wejścia.

Źródło: src/ClusterManager/Input/Drivers/InputDriver.php

Możliwość prostej zamiany aktualnie pracującego sterownika jest możliwa poprzez kontener biblioteki Symfony oraz wzorzec projektowy wstrzykiwania zależności. Aktualnie aplikacja posiada pięć sterowników wejścia które można dowolnie

podmieniać co jest przydatne gdy chcemy coś przetestować lub przenieść aplikację na nowe środowisko. Aktualne sterowniki:

Wybór aktualnego sterownika następuje w pliku gdzie definiujemy zależności kontenera. Wygląda to następująco:

- NullInputDriver, sterownik który zawsze zwraca brak wciśniętego przycisku. Jest on przydatny gdy chcemy coś napisać i/lub przetestować co nie wymaga interakcji użytkownika oraz nie chcemy żadnej konfiguracji.
- FileBasedInputDriver, sterownik który zaczytuje z pliku informację na temat wciśniętych przycisków. Umożliwia on integracje z innymi skryptami co umożliwia oddelegowanie odpowiedzialności sterownika poza język PHP który ma swoje limitacje. Aktualnie jest on zintegrowany z skryptem napisanym w języku Python który zapewnia komunikację UART z Arduino które jest odpowiedzialne za wyświetlanie tekstu na zegarach.
- BMWStalksAndIBusSteeringWheelButtonsBasedInputDriver, natywny sterownik obsługujący całość kierownicy multifunkcyjnej oraz obu manetek, telefonu oraz komputera pokładowego. Wymaga implementacji, do której potrzebny jest moduł „i-bus", czyli interfejsu którego używa BMW jako medium połączeń pomiędzy urządzeniami na pokładzie samochodu.

```
#================== Drivers ==================#

E32CM\ClusterManager\Input\Drivers\InputDriver:
    '@E32CM\ClusterManager\Input\Drivers\FileBasedInputDriver'
```

Fot. 2.5. Część pliku services.yaml, sekcja Input.

Źródło: config/application/container/services.yaml

W interfejs InputDriver została wstrzyknięta klasa FileBasedInputDriver.

2.3 Rdzeń

Głównym rdzeniem aplikacji który reguluje oraz umożliwia komunikację wejścia, wyjścia oraz właściwej logiki załadowanej aplikacji jest klasa „Main”.

```
class Main
{
    private Input $input;
    private Output $output;
    private ClusterApplication $clusterApplication;
```

Fot. 2.6. Część pliku main.php.

Źródło: src/ClusterManager/Main/Main.php

Można zauważyć, że klasa składa się z trzech modułów:

- Input, czyli wejście aplikacji. Moduł ten został opisany w rozdziale 2.2.
- Output, czyli wyjścia aplikacji. Moduł ten został opisany w rozdziale 2.6.
- ClusterApplication, czyli właściwa logika załadowanej aplikacji. Moduł jest opisany w rozdziale 2.4 oraz 2.5.

Klasa ta posiada główną funkcję która zapętla aplikację. Funkcja „run” która nie przyjmuje argumentów jest wykonywana w nieskończonej pętli. To ona czyni aplikację responsywną oraz umożliwia przepływ informacji pomiędzy modułami w każdym kierunku. Posiada ona obsługę błędów aplikacji które są wypisywane na konsole deweloperską. Dodatkowo jest w niej użyta funkcja „usleep” która ustanawia częstotliwość aplikacji. Umożliwia to zmianę zachowania aplikacji co ma wpływ na responsywność oraz zużycie zasobów maszyny na które uruchomiona jest aplikacja.

```
public function run(): void
{
    while (true) {
        usleep(100000);
        try {
            $inputCommand = $this->input->getInputIfAny();
        } catch (UnknownButtonException $exception) {
            $this->debugOutput->writeln('Unknown button!');
            $inputCommand = null;
        }

        if ($inputCommand === null) {
            $outputInstruction = $this->clusterApplication->tick(
                $this->lastOutputStatus
            );
        } else {
            $outputInstruction = $this->clusterApplication->processCommand(
                $inputCommand
            );
        }

        if ($outputInstruction instanceof DisplayMessageCommand) {
            try {
                $this->output->displayMessage(
                    $outputInstruction->getMessage(),
                    $outputInstruction->getScrollSpeed(),
                    $outputInstruction->getDisplayMode()
                );
                $this->lastOutputStatus = $this->output->tick();
            } catch (\Exception $exception) {
                echo $exception->getMessage();
            }
            continue;
        }
```

Fot. 2.6. Dalsza część pliku main.php, funkcja run.

Źródło: src/ClusterManager/Main/Main.php

Na powyższej ilustracji można dostrzec przepływ poleceń pomiędzy „wejściem”, a aplikacją oraz stworzenie i przekazanie rozkazu z aplikacji do „wyjścia”. Można zauważyć, że aplikacja posiada dwie funkcje: „tick” oraz „processCommand”. Funkcja „tick” zakłada brak interakcji użytkownika z urządzeniem, więc aby mieć obraz stanu całości urządzenia w funkcji podawany jest parametr w postaci ostatniego statusu „wyjścia”. Pozwala to dostrzec aplikacji czy przypadkiem wyświetlanie tekstu się nie zakończyło lub czy nie wystąpił inny stan „wyjścia”. Funkcja „processCommand” bierze pod uwagę tylko i wyłącznie polecenie użytkownika. Dzieje się tak ponieważ interpretowanie poleceń użytkownika jest niezależne od stanu wyświetlacza, jak i całości aplikacji.

2.4 Aplikacja

Aplikacja jest tą częścią całości oprogramowania gdzie opisane jest zachowanie urządzenia z punktu widzenia użytkownika. To w niej zaimplementowano sposób reakcji na poszczególne działania użytkownika oraz innych czynników, np. zewnętrznych API.

Interfejs opisujący wspomnianą wcześniej aplikację to „ClusterApplication". Zwiera on trzy główne funkcje bez których działanie jest niemożliwe:

- Configure, umożliwia załadowanie konfiguracji do klasy aplikacji. Każda aplikacja może zawierać inne dane w konfiguracji, w zależności od potrzeb, lub może nie wymagać ich wcale. Umożliwia ona również podmianę konfiguracji na „ciepło" podczas działania aplikacji.

- ProcessCommand, czyli funkcja umożliwiająca interakcje użytkownika z urządzeniem. Jako parametr przyjmuje ona klasę „InputCommand" która ma w sobie informacje o poleceniu które nadał użytkownik. Zadaniem ciała tej funkcji jest odpowiednie zareagowanie na te polecenia i zwrócenie odpowiedniego rozkazu dla „wyjścia".

- Tick, to funkcja wywoływana jeśli użytkownik w danym cyklu nie przeprowadził żadnej interakcji z urządzeniem. Jako parametr otrzymuje ona ostatni stan klasy odpowiedzialnej za „wyjście" i na jej podstawie odpowiednio reaguje na aktualny stan aplikacji. To w jaki sposób się to dzieje zależy już od faktycznej implementacji.

```
interface ClusterApplication
{
    public function configure(AppConfiguration $configuration): void;

    public function processCommand(InputCommand $command):
ApplicationCommand;

    public function tick(string $lastOutputStatus): ApplicationCommand;
}
```

Fot. 2.7. Interfejs ClusterApplication.

Źródło: src/ClusterManager/Main/ClusterApplications/ClusterApplication.php

2.5 Discord - przykładowa implementacja aplikacji

Aplikacją która została napisana celem umożliwienia demonstracji całości platformy jest prosta implementacja usługi Discord. Jednym z głównych możliwości Discorda jest pisanie wiadomości na poszczególnych kanałach, prywatnych lub publicznych. Aplikacja zakłada możliwość pobrania wiadomości z podanych przez użytkownika kanałów komunikacji, które może wskazać w aplikacji webowej, ich zakolejkowanie oraz następnie wyświetlenie na ekranie LCD znajdującym się na zegarach samochodowych. Wyświetlanie następuje płynnie, znaki odpowiednio przesuwają się od prawej do lewej strony wyświetlacza. Gdy tekst zniknie z ekranu, zostaje z kolejki pobrana kolejna wiadomość. Użytkownik ma możliwość pomijania aktualnej wiadomości i żądanie wyświetlenia kolejnej poprzez wciśnięcie zdefiniowanego przez niego w aplikacji mobilnej przycisku, domyślnie manetki „BC".

```
class Discord implements ClusterApplication
{
    public const APP_NAME = 'DISCORD';

    public const USING_OUTPUT = 'USING_OUTPUT';
    public const WAITING = 'WAITING';
    public const READY = 'READY';
    public const CONFIGURE = 'CONFIGURE';

    private MessageConsumer $messageConsumer;

    private DiscordConfiguration $configuration;
```

Fot. 2.8. Implementacja interfejsu ClusterApplication w klasie Discord. Wycinek kodu prezentuje możliwe stany aplikacji, jej nazwę oraz usługi z których korzysta.
Źródło: src/ClusterManager/Main/ClusterApplications/ClusterApplication.php

Można zauważyć, że aplikacja potrafi przyjąć 4 stany:

- „USING_OUTPUT" – aplikacja przyjmuje taki stan gdy na ekranie jest wyświetlany tekst. Sygnalizuje to, że każda kolejna wiadomość zostanie zakolejkowana, a chęć jej natychmiastowego wyświetlania musi być oznaczona odpowiednią flagą

- WAITING – aplikacja wyświetliła wszystkie możliwe wiadomości z kolejki i oczekuje na kolejne. Stan utrzymuje się do momentu w którym nie zostanie dodana jakakolwiek wiadomość do kolejki.

- READY – domyślny stan przy inicjalizacji klasy. Natychmiast nadpisywany przez kolejne stany zgodnie z logiką biznesową aplikacji.

- CONFIGURE – stan krytyczny w który wchodzi aplikacja jeśli konfiguracja jest niekompletna, nie istnieje lub jej format jest nieprawidłowy. Stan ten jest natychmiastowo sygnalizowany na ekranie LCD zegarów. Nie jest możliwa zmiana tego stanu, wymagana jest rekonfiguracja aplikacji oraz jej ponowne uruchomienie.

Fot. 2.8. Stan CONFIGURE aplikacji jest sygnalizowany poprzez wyświetlenie wiadomości „NO CONFIG SET”. Urządzenie aby pracować musi zostać zrekonfigurowane oraz uruchomione ponownie.
Źródło: opracowanie własne

Klasa Discord aby móc pobierać wiadomości z kolejki używa serwisu MessageConsumer. Sam MessageConsumer jest tylko interfejsem, lecz jego faktyczną implementacją w tym momencie jest MessageConsumerBasedOnMysql. Klasa ta zgodnie z nazwą jest kolejką opartą o bazę danych MySQL. Jej jedyną rolą jest pobieranie pierwszej wiadomości z kolejki oraz jej usunięcie. Wiadomość następnie jest przetwarzana przez właściwą aplikację.

Główna logika aplikacji jest zaimplementowana w funkcji „tick”. Ona wymagając podania ostatniego stanu „wyjścia” odpowiednio reaguje na to co się dzieje i decyduje co aplikacja chce zrobić, co wyświetlić oraz co zlogować w systemie. Co każde wywołanie funkcji aplikacja sprawdza czy jej stan się zmienił. Jeśli taka sytuacja nastąpiła to w konsoli deweloperskiej przestawione jest aktualne zużycie pamięci oraz stan aplikacji.

```
public function tick(string $lastOutputStatus): ApplicationCommand
{
    if ($this() === self::CONFIGURE) {
        return DisplayMessageCommand::createWithMessageAndScrollSpeed(
            '  NO CONFIG SET ',
            Output::NO_SCROLLING,
            DisplayMessageCommand::FORCE_DISPLAY
        );
    }

    if ($this->lastKnownOutputState !== $lastOutputStatus) {
        $this->lastKnownOutputState = $lastOutputStatus;
        \E32CM\log('==================================================');
        \E32CM\log('Discord detected that output is '
            . $lastOutputStatus . '!
         (MEM: ' . memory_get_peak_usage() .        ')');
    }

    if ($lastOutputStatus === Output::READY) {
        try {
            return $this->displayNextDiscordMessage();
        } catch (NoMessagesException $exception) {
            $this->currentAppState = self::WAITING;
        }
    }

    if ($lastOutputStatus === Output::BUSY) {
        return IdleCommand::create();
    }

    return IdleCommand::create();
}
```

Fot. 2.9. Całość funkcji tick.

Źródło: src/ClusterManager/Main/ClusterApplications/Discord/Discord.php

Funkcja sprawdza w jakim stanie jest aplikacja. w zależności od tego jaki jest jej stan, to jej zachowanie jest zgoła inne. Tak jak wcześniej zostało wspomniane, stan „CONFIGURE” jest stanem krytycznym aplikacji i służy zablokowaniu aplikacji dodatkowo wyświetlając monit na ekranie. Następnie sprawdzane jest czy „wyjście” zmieniło swój stan. Zmiana stanu wyjścia oznacza, że rozpoczął on lub zakończył wyświetlanie wiadomości na ekranie. Stan „READY” czyli zakończenie daje zielone

światło na pobranie z bazy wiadomości, jej spreparowanie oraz wysłanie do wyświetlenia. Jeśli, żaden z scenariuszy się nie spełnia to aplikacja wysyła sygnał o swojej bezczynności.

Preparowanie wiadomości, czyli ich odpowiednie ubranie w metadane jest niezbędne przed wyświetleniem. Tylko w ten sposób możemy poinformować o tym od kogo jest dana wiadomość oraz na jakim kanale została wysłana. Klasa Discord również została oddelegowana do tego zadania, aby w odpowiedni sposób preparować wiadomości. Funkcja „formulateFinalMessageToDisplay" przyjmuje argument „message" który przechowuje w sobie informacje na temat tego kto wysłał daną wiadomość, na jakim kanale, kiedy, czy jest to wiadomość prywatna, czy ma załącznik oraz samą jej treść. Na to, jak finalnie wygląda wiadomość ma wpływ to czy jest to wiadomość prywatna, czy nastąpiła zmiana nadawcy względem poprzedniej wiadomości, oraz czy nastąpiła zmiana kanału rozmowy.

Tab. 2.1. Budowa przykładowej wiadomości „B” o treści „Ala ma kota” od użytkownika Kajtek na serwerze Ognisko i kanale Namioty wysłana po wiadomości „A”

Scenariusz	Treść wiadomości „B”
Wiadomość A oraz B zostały wysłane na różnych serwerach	Ognisko->Namioty->Kajtek: Ala ma kota
Wiadomość A oraz B zostały wysłane na tych samych serwerach, ale różnych kanałach	Namioty->Kajtek: Ala ma kota
Wiadomość A oraz B zostały wysłane na tych samych serwerach, tych samych kanałach, lecz przez innych użytkowników	Kajtek: Ala ma kota
Wiadomość A oraz B zostały wysłane na tych samych serwerach, tych samych kanałach i przez tych samych użytkowników	Ala ma kota
Wiadomość A była wiadomością prywatną, a wiadomość B jest wiadomością publiczną	Ognisko->Namioty->Kajtek: Ala ma kota
Wiadomość A była wiadomością prywatną, tak samo jak wiadomość B, lecz wiadomość B pochodzi od innego nadawcy	PM->Kajtek: Ala ma kota

Wiadomość A oraz B była wiadomością prywatną od tego samego nadawcy	Ala ma kota

Źródło: opracowanie własne.

Ostatnia ważną funkcją klasy Discord jest funkcja „processCommand”. Jako argument przyjmuje ona aktualnie wciśnięty przez użytkownika przycisk. To w tej funkcji zachodzi interpretacja i wykonanie odpowiedniej akcji na postawie zachowania użytkownika. Aplikacja w tym momencie przewiduje tylko jedną akcję, jest to akcja „OK”. Jeśli aplikacja wykryje taką operacje to zwracane jest polecenie pominięcia aktualnie wyświetlanej wiadomości. Daje to użytkownikowi możliwość pomijania dowolnie wybranych wiadomości.

```
public function processCommand(InputCommand $command): ApplicationCommand
{
    if ($this() === self::USING_OUTPUT
        && $command->getCommand() === InputCommand::OK)
    {
            return SkipMessageCommand::create();
    }

    return IdleCommand::create();
}
```

Fot. 2.10. Funkcja processCommand.

Źródło: src/ClusterManager/Main/ClusterApplications/Discord/Discord.php

2.6 Interfejs wyjścia

Ostatnim głównym komponentem aplikacji jest interfejs wyjścia. Za tę funkcjonalność jest odpowiedzialna klasa Output. Klasa ta w przeciwieństwie do interfejsu wejścia posiada stan i potrafi dwustronnie się komunikować z samą aplikacją. Wyjście potrafi przyjąć trzy stany:

- BUSY – stan który jest przyjmowany w momencie gdy zachodzi wyświetlanie czegoś na wyjściu. Wtedy jedyną możliwością zamiany aktualnie wyświetlanej informacji na inną jest dostarczenie rozkazu z flagą FORCE.

- READY – interfejs wyjścia jest gotowy do pracy i/lub zakończył wyświetlać tekst.
- WAITING – niewykorzystywany, zarezerwowany dla współpracy z dodatkowymi integracjami.

```
class Output
{
    public const BUSY = 'BUSY';
    public const WAITING = 'WAITING';
    public const READY = 'READY';

    public const LIST_OF_SCROLLING_SPEEDS = [
        self::NO_SCROLLING,
        self::VERY_SLOW_SCROLLING,
        self::SLOW_SCROLLING,
        self::NORMAL_SCROLLING,
        self::FAST_SCROLLING,
        self::VERY_FAST_SCROLLING,
        self::PRESENTATION_SCROLLING
    ];

    private OutputDriver $outputDriver;
```

Fot. 2.11. Wycinek klasy Output.

Źródło: src/ClusterManager/Output/Output.php

Można zauważyć, że oprócz stanów klasa posiada definicje prędkości przesuwania tekstu po ekranie oraz sterownik wyjścia. To z jaką prędkością jest wyświetlana dana wiadomość jest definiowane przez samą aplikację, co pośrednio może być skonfigurowane przez użytkownika w aplikacji mobilnej.

Sterownik wyjścia, czyli OutputDriver ma podobną zasadę działania co sterowniki użyte w interfejsie wejścia. Definiują one medium przez które i na które jest wysyłana informacja o wyświetlanym tekście. Wybór aktualnego sterownika następuje w pliku gdzie definiujemy zależności kontenera. Wygląda to następująco:

```
E32CM\ClusterManager\Output\Drivers\OutputDriver:
'@E32CM\ClusterManager\Output\Drivers\FileDriver'
```

Fot. 2.12. Część pliku services.yaml, sekcja Output.

Źródło: config/application/container/services.yaml

W tym momencie interfejs wyjścia posiada pięć sterowników które można dowolnie podłączać i podmieniać wzajemnie pomiędzy uruchomieniami aplikacji:

- SerialDriver – implementacja sterownika wyjścia z użyciem interfejsu UART/RS232 z wykorzystaniem klasy PhpSerial. Wymaga konfiguracji przed wykorzystaniem.
- NullDriver – implementacja która nie wykonuje żadnej akcji. Wykorzystywana w celach czysto testowych.
- SimpleConsoleDriver – implementacja wykorzystująca okno verbose PHP jako ekran. Pozwala na proste uruchomienie oraz debugowanie aplikacji bez konieczności połączenia całości urządzenia.
- SymfonyConsoleDriver – do zaimplementowania, pozwala na to samo co SimpleConsoleDriver lecz z użyciem frameworka Symfony, komponentu konsolowego.
- FileDriver – implementacja wykorzystująca plik tekstowy jako bufor zapisu wyjścia ekranowego. Wraz z użyciem skryptu pomocniczego który komunikuje się z docelowym urządzeniem pozwala na wyświetlenie tekstu użytkownikowi.

Klasa Output spełnia swoją role za pomocą trzech funkcji. Podobnie jak Input posiada dwie główne funkcje czyli „tick”, funkcję uruchamianą w wypadku braku interakcji z użytkownikiem i funkcję „displayMessage” która przyjmuje informacje

z zewnątrz jako parametr. Wyjątkiem tutaj jest funkcja „skipCurrentMessage” która po prostu pomija aktualnie wyświetlaną wiadomośc.

Funkcja „tick” to główna funkcja wykorzystowana przez interfejs wyjścia. Jej odpowiedzialnością jest pobieranie wiadomości z kolejki, ich preparowanie i wyświetlanie, odpowiednie przesuwanie po ekranie. Również ta funkcja odmierza czas pomiędzy poszczególnymi przesunięciami tekstu na ekranie licząc czas od ostatniej zmiany tekstu na ekranie oraz w odpowiednich przypadkach zmienia globalny stan wyjścia i pośrednio informuje o nim aplikację która następnie na to reaguje w odpowiedni sposób.

```
public function tick(): string
{
    $this->doTickTimeCheck = true;
    try {
        $this->getMessageForThisTick();
    } catch (NoMessageToDisplayException $exception) {
        return $this->currentAppState;
    }

    if ($this->doTickTimeCheck) {
        $waitTime = constant(
                OutputTimeMapping::class . "::"
                 . $this->currentScrollingSpeed);
        if ((microtime(true) - $this->lastScrollTickTime) < $waitTime) {
            usleep((int)($waitTime * 1000000));
            return $this->currentAppState;
        }
    }

    /** Has finished displaying? */
    if ($this->currentScrollingPosition === mb_strlen(
                $this->currentMessage)
        ) {
        $this->changeState(self::READY);
        $this->clearScreen();
        return $this->currentAppState;
    }

    /** Do something */
    $this->changeState(self::BUSY);
    $finalMessage = mb_substr(
        $this->currentMessage,
        $this->currentScrollingPosition,
        16,
        'UTF-8'
    );
    $this->outputDriver->displayMessage($finalMessage);
    $this->currentScrollingPosition += 1;
    $this->updateLastScrollTickTime();
    return $this->currentAppState;
}
```

Fot. 2.13. Funkcja tick, klasy Output.

Źródło: src/ClusterManager/Output/Output.php

Na podstawie powyższej ilustracji można bardzo prosto opisać sposób działania tej funkcji. w pierwszej kolejności sprawdza ona jaki tekst jest przeznaczony do wyświetlenia o ile jest. w dalszej kolejności sprawdza się, czy upłynęło dość czasu by usunąć wiadomość lub ją przesunąć po ekranie zgodnie z definicją szybkości przesuwania zawartą w samej wiadomości. Jeśli czas upłynął wyświetlania, to wiadomość jest odpowiednio preparowana oraz jeśli to wymagane przesunięta

o odpowiednią ilość liter. Następnie jest ona wysyłana do sterownika wyjścia. Finalnie jeśli jest to wymagane, jest aktualizowany stan wyjścia.

Funkcja kolejkująca wiadomości, czyli displayMessage przyjmuje trzy parametry:

- Message – treść wiadomości odpowiednio wstępnie przygotowana przez aplikację.
- ScrollMode – wartość opisująca sposób oraz prędkość przesuwania się tekstu po ekranie. Lista trybów jest zdefiniowana w tej samej klasie.
- DisplayMode – flaga pozwalająca na wymuszenie wyświetlenia właśnie dodawanej wiadomości z pominięciem kolejki oraz właśnie wyświetlanej wiadomości.

```
public function displayMessage(string $message, string $scrollMode, string
$displayMode): void
{
    if (!$this->isValidScrollMode($scrollMode)) {
        throw
InvalidScrollModeException::createWithInvalidScrollMode($scrollMode);
    }

    if ($scrollMode === self::NO_SCROLLING) {
        $this->outputDriver->displayMessage($message);
        sleep(10);
        return;
    }

    if ($displayMode === DisplayMessageCommand::FORCE_DISPLAY) {
        $this->displayQueue = [];
        $this->currentMessage = $message;
        $this->currentScrollingSpeed = $scrollMode;
        $this->currentScrollingPosition = 0;
        $this->changeState(self::BUSY);
        return;
    }

    $this->displayQueue[] = ['message' => "   " . $message, 'scrollMode' =>
$scrollMode];
}
```

Fot. 2.14. Funkcja displayMessage, klasy Output.

Źródło: src/ClusterManager/Output/Output.php

Zasada działania funkcji polega na sprawdzeniu czy podana wartość prędkością przesuwania tekstu jest prawidłowa. Jeśli wiadomość spełnia ten warunek to następnie sprawdzana jest flaga wymuszania nadpisania wiadomości. Jeśli jest ona włączona to interfejs wyjścia czyści kolejkę wiadomości oraz dodaję tę wiadomość na jej początek. Jeśli flaga jest nieaktywna to wiadomość jest dodawana na koniec kolejki wyświetlania, skąd w końcu zostanie pobrana gdy przyjdzie jej kolej.

2.7 Dodatkowe skrypty

Aplikacja może być wspomagana w swojej pracy poprzez dodatkowe skrypty działające w tle. Atutem takiego rozwiązania jest możliwość wykorzystania innych języków programowania które lepiej współpracują z interfejsami służącymi do komunikacji z urządzeniami zewnętrznymi jak na przykład COM, I2C czy 1Wire.

Sterownik wyjścia oparty na pliku, aby móc w pełni funkcjonować potrzebuje dodatkowego skryptu w języku Python który odpowiada za wysyłanie treści pliku przez port COM. Kod pobiera co określony interwał czasu treść pliku zapisaną przez sterownik wyjścia oraz przesyła ją przez wspomniany kanał komunikacji.

```
import serial
import time
import sys

ser = serial.Serial(port='COM9',baudrate=9600,parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)

if ser.isOpen():
    print(ser.name + ' is open...')

value = "                "

while True:  # value > -1 and value < 10:
    print(value.ljust(16, ' ').encode('utf-8'))
    ser.write(value.ljust(16, ' ').encode('utf-8'))
    time.sleep(1)
    f = open("VirtualLCD.txt", "r")
    value = (f.readline())
    f.close()
```

Fot. 2.15. Skrypt FileToSerialAdapter.py odpowiedzialny za komunikacje COM.

Źródło: FileToSerialAdapter.py

Bliźniaczym skryptem jest skrypt GPIOToFileAdapter.py. Jest to skrypt również napisany w języku Python który, zakładając odpowiednie podłączenie przewodów umożliwia sprawdzenie czy przycisk manetki BC został wciśnięty. Następuje to poprzez sprawdzenie stanu GPIO numer 18 i jeśli jego stan jest wysoki to odpowiednia fraza jest zapisywania pliku bufora używanym przez sterownik wejścia.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)

GPIO.setup(18, GPIO.IN)

while True:
    if GPIO.input(18) == GPIO.HIGH:
        with open("VirtualIbus.txt", "w") as file:
            file.write("BC_STALK_BUTTON")
    time.sleep(0.1)
```

Fot. 2.16. Skrypt GPIOToFileAdapter.py odpowiedzialny za sprawdzanie stanu wciśnięcia manetki BC.

Źródło: GPIOToFileAdapter.py

Aplikacja Discord potrzebuje do poprawnego działania uruchomienia w tle skryptu „discordListener” który jest oparty o konsole skryptową Symfony. Uruchamiany jest on komendą „php bin/console app:discordListener”. Po jego uruchomieniu, skrypt loguje się za pomocą tokena który został wpisany przez użytkownika w aplikacji mobilnej do platformy Discord, a następnie nasłuchuje on wiadomości i zapisuje do kolejki. Całość jest nadzorowana oraz oparta na bibliotece DiscordPHP[5].

```
$this->discord->on('ready', function (Discord $discord) {
    $discord->on(Event::MESSAGE_CREATE,
        function (DiscordMessage $message, Discord $discord) {
        /** no guild = private message */
        $isPrivateMessage = ($message->guild === null);

        if (!$isPrivateMessage) {

            if (!in_array(
                $message->channel->id,
                $this->allowedChannels))
            {
                return;
            }
        }

        /** Add strategy for image/files handling
         * to inform user that SOMETHING was sent?
         */
        $this->messageQueue->addToQueue(
            new Message(
                null,
                $message->content,
                $message->guild->name ?? null,
                $message->author->displayname,
                $message->channel->name ?? null,
                $isPrivateMessage,
                count($message->attachments) !== 0,
                null
            )
        );
        echo($message->author->displayname. ': '
            . $message->content . PHP_EOL);
    });
});
```

Fot. 2.17. Wycinek funkcji run, klasy Main.

Źródło: src/ClusterManager/Main/ClusterApplications/Discord/DiscordListener/Main.php

Na powyższym wycinku kodu można zauważyć, że skrypt uruchamia eventListener na zdarzeniu „Event::MESSAGE_CREATE”. Co wystąpienie takiego zdarzenia zwracany jest obiekt wiadomości który przechowuje wszystkie wymagane przez nas dane. Następnie te dane są przepakowywanie i wysyłane do kolejki za pomocą funkcji „addToQueue” w serwisie messageQueue który jest dowolną implementacją kolejki wiadomości.

Ostatnim oraz w tym wypadku najważniejszym skrypem, tym razem wyjątkowo uruchomionym nie na RaspberryPi lecz na Arduino jest kod odpowiedzialny ze

formułowanie oraz przesyłanie tesktu bezpośrednio na ekran LCD zegarów samochodowych.

```
#include <SoftwareSerialParity.h>
SoftwareSerialParity ClusterSerial(2, 3); // RX, TX
#define MAX_INPUT_LENGTH 16
char Message[17] = "1234567890ABCDEF";
void setup() {
  Serial.begin(9600);
  ClusterSerial.begin(9600, EVEN);
}

void loop() {
  char input[MAX_INPUT_LENGTH + 1];
  int index = 0;
  if (Serial.available()) {
    while (index < MAX_INPUT_LENGTH) {
      Serial.write(String(index).c_str());
        digitalWrite(LED_BUILTIN, HIGH);
      if (Serial.available()) {
        char c = Serial.read();  // odczytuje pojedynczy znak
        input[index] = c;  // zapisuje znak do bufora
        index++;  // zwiększa indeks
      }
        digitalWrite(LED_BUILTIN, LOW);
    }
    strcpy(Message, input);
  }
  delay(95);
  clusterdisplay(Message);
}

void clusterdisplay(char* input) {
  int x = 0;
  int sum = 1;
  int len = 16;
  for (int i = strlen(input); i < 16; i++) {
    input[i]=0x20;
  }
  int length = strlen(input);
  if (length > 0) {
      input[17 - 1] = '\0';
  }
  Serial.println(input);
  ClusterSerial.write(input);
  while (len > 0){  //calculates checksum
    sum += input[x];
    len -=1;
    x += 1;
    }
  ClusterSerial.write(sum);  //sends checksum
}
```

Fot. 2.18. Kod odpowiedzialny za komunikacje z wyświetlaczem LCD zegarów

Źródło: opracowanie własne

Arduino pobiera z RaspberryPi poprzez UART aktualnie zadany do wyświetlenia ciąg znaków oraz następnie wysyła go po odpowiedniej obróbce do wyświetlacza LCD. Wyświetlacz oczekuje dokładnie 16 znaków w formacie ASCI oraz następnie sumy kontrolnej[2]. Suma kontrolna to suma wartości ASCI znaków danego ciągu dla którego obliczamy sumę. Suma musi zostać wysłana natychmiast po wiadomości która ma zostać wyświetlona. w przeciwnym wypadku całość zostanie zignorowana przez wyświetlacz.

Rozdział 3 Hardware

3.1 Raspberry Pi – połączenia elektryczne

Raspberry Pi, żeby móc zostać uruchomione wraz z startem samochodu wymaga podania 5V na pin drugi oraz czwarty, jak i również masy na pin numer sześć. Do komunikacji z Arduino wymagane jest podłączenie pinu ósmego, UART TX oraz dziesiątego UART RX z odpowiednio pinem zerowym oraz pierwszym na Arduno. Aby udostępnić użytkownikowi podstawową możliwość interakcji z urządzeniem należy również poprzez manetkę BC podpiąć pin numer jeden, czyli 3.3V do pinu osiemnastego. Finalnie połączenie elektryczne Raspberry Pi wygląda następująco:

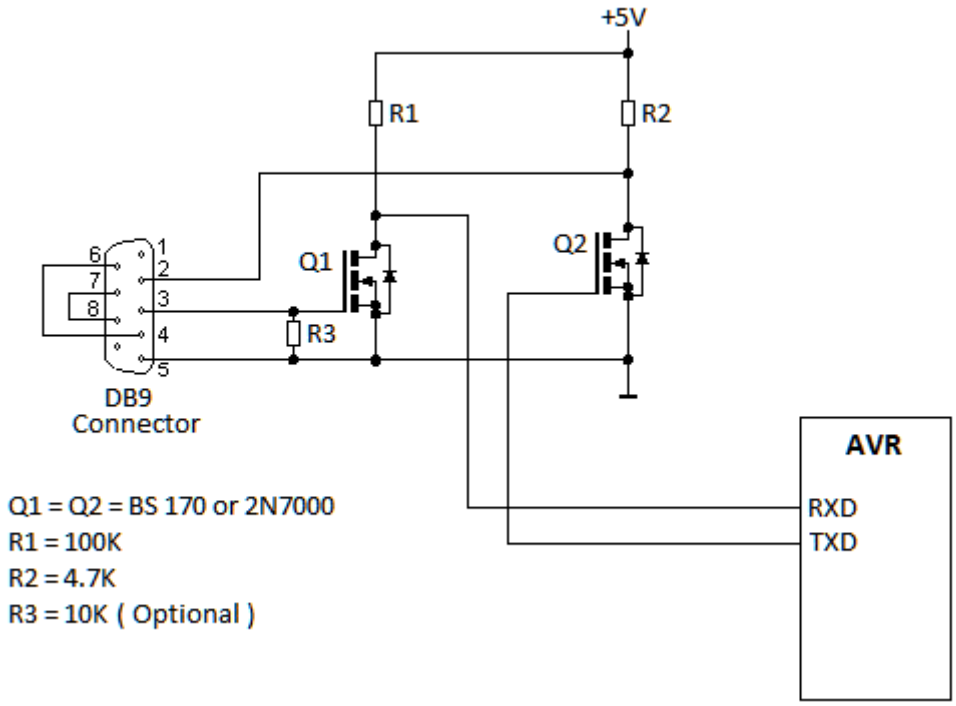
Tab. 3.1. Połączenia elektryczne Raspberry Pi

Pin	Nazwa	Połączono z	Uwagi
2	5V	Zasilanie 5V zewnętrzne	Podawane podczas pracy silnika oraz stanu Ignition
4	5V	Zasilanie 5V zewnętrzne	Podawane podczas pracy silnika oraz stanu Ignition
6	Masa	Masa samochodu	
8	UART TX	Pin 0 Arduino	Wymagany do komunikacji z LCD
10	UART RX	Pin 1 Arduino	
1	3.3V	Pin 18 Raspberry Pi	Połączony poprzez przycisk manetki BC

3.2 Arduino oraz zegary – połączenia elektryczne

Ardunio jest używane jako medium komunikacji z zegarami samochodu. Ta rola powoduje konieczność odpowiedniego połączenia z zegarami jak i Raspberry Pi celem przesyłania informacji wysyłanych przez aplikacje uruchomioną na wspomnianym mikrokomputerze. Aby zostać uruchomionym Arduino musi zostać mieć podłączony pin

VIN do magistrali 12V samochodu oraz GND pod masę. Wykorzystane w tym przypadku Arduino UNO cierpi na problem zbyt małej ilości złącz UART. Mianowicie posiada ono tylko jeden taki interfejs. Jednakże jest możliwe dodatnie dodatkowego, programowego interfejsu UART co zostało wykorzystane w tym projekcie, czego implementację można zaobserwować na fot 2.18. Nowo stworzony UART na pinie numer 3 podłączamy do pinu numer 23 złącz X16 zegarów z BMW poprzez konwerter RS-232 / TTL. Przykładowy użyty konwerter wygląda następująco:



Fot. 3.1. Schemat konwertera napięć RS232 / TTL.

Źródło: http://www.peterfleury.epizy.com/avr-uart.html

Natywne porty UART czyli pin 0 oraz 1 odpowiednio łączymy z pinami 8 oraz 10 na mikrokomputerze RaspberryPi.

Tab. 3.2. Połączenia elektryczne Arduino

Pin	Nazwa	Połączono z	Uwagi
VIN	12V	Zasilanie 12V zewnętrzne	Podawane podczas pracy silnika oraz stanu Ignition
GND	Masa	Masa samochodu	
0	UART RX	Pin 8 Raspberry Pi	Pobieranie treści wiadomości do wyświetlenia
1	UART TX	Pin 10 Raspberry Pi	
3	Software UART RX	Magistrala DAC / Pin 23 złącza X16 zegarów samochodowych	Komunikacja z LCD

Rozdział 4 Aplikacja mobilna

4.1 Pierwsze uruchomienie

Główną metodą zmiany ustawień urządzenia jest aplikacja internetowa oparta o Javascript, CSS oraz PHP. Jest ona dostępna na adresie mikrokomputera na którym jest uruchomiona aplikacja ClusterManager.

Przy pierwszej wizycie, aplikacja przywita nas wstępnym, uproszczonym panelem konfiguracyjnym.

Fot. 4.1. Strona „Wizarda" która przeprowadza wstępna konfigurację

Źródło: opracowanie własne

Panel jest przystosowany pod widok z telefonu, lecz obsługuje również inne media wyświetlania. Po wybraniu interesującego nas zestawi ustawień, lub zdefiniowaniu własnego możemy korzystać z aplikacji.

4.2 Ustawienia aplikacji

Po wyborze ustawień klawiszy, od użytkownika wymagane jest również w wypadku aplikacji Discord podanie klucza logowania na konto użytkownika. Tylko w ten sposób aplikacja może się zalogować w jego imieniu.

Fot. 4.2. Wycinek podstrony „Settings”

Źródło: opracowanie własne

Sumarycznie podstrona „settings” pozwala na ustawienie:

- Mapowania przycisków, czyli zdefiniowanie który przycisk odpowiada za którą akcję w aplikacji
- Tokena Discord, czyli jak wspomniano wyżej klucza dostępu do konta celem umożliwienia pobierania wiadomości
- Białej listy kanałów, czyli listy kanałów z których użytkownik chcę mieć wyświetlanie wiadomości. Identyfikatory kanałów wpisujemy po przecinku
- Szybkość przesuwania tekstu, czyli prędkość z jaką aplikacja będzie przesuwała litery po ekranie celem zmieszczenia wiadomości dłuższych niż 16 liter
- Wybór samej aplikacji

Rozdział 5 Uruchamianie oraz testowanie

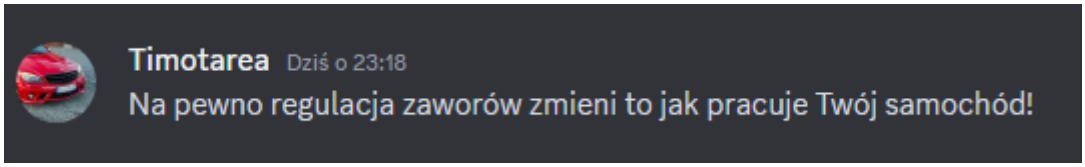
5.1 Środowisko testowe

Środowisko zostało podpięte zgodnie z wytycznymi przytoczonymi w rozdziałach 3.1 oraz 3.2. Użyto mikrokomputerów Raspberry Pi oraz Arduino UNO, zgodnie z tym co przytoczono w wspomnianych rozdziałach oraz zegarów VDO 110.008.548/061.

Po podłączeniu, skonfigurowaniu systemu operacyjnego na Raspberry Pi oraz sprawdzeniu połączeń multimetrem, skonfigurowano ustawienia w aplikacji mobilnej. Wybrano układ przycisków, uzupełniono pole z tokenem logowania, dodano odpowiednie identyfikatory kanałów do białej listy oraz wybrano prędkość „przesuwania" tekstu na ekranie LCD.

5.2 Testowanie

Rozwiązanie zostało przetestowanie poprzez wysłanie przykładowej wiadomości do teoretycznego użytkownika który posiada zegary jako medium odbioru wiadomości Discord. Wiadomość testowa ma treść „Na pewno regulacja zaworów zmieni to jak pracuje Twój samochód!" oraz została wysłana jako wiadomość prywatna do innego użytkownika. Natychmiast po wysłaniu, wiadomość powinna wyświetlić się na wyświetlaczu LCD zegarów samochodowych, a następnie zostać odpowiednio przesuwana, tak by wyświetlić jej całość na szesnastoznakowym ekranie.



Fot. 5.1. Wiadomość wysłana przez użytkownika Timotarea w aplikacji Discord
Źródło: opracowanie własne

Wiadomość natychmiastowo pojawiła się na ekranie LCD zegarów oraz następnie została przewinięta w równych interwałach czasu w lewo tak aby umożliwić jej odczytanie przez użytkownika mimo bycia dłuższą niż 16 znaków na które pozwala wyświetlacz.

Fot. 5.2. Przewinięta wiadomość do momentu „no regulacja zaw”

Źródło: opracowanie własne

Każda kolejna wiadomość została również wyświetlona. Jedyną różnicą było kolejkowanie, które następowało, gdy wiadomość była wysyłana częściej niż poprzednia zdążyła się całkowicie wyświetlić. w takiej sytuacji mikrokomputer sekwencyjnie wyświetlał je po kolei na ekranie zegarów samochodowych.

Zakończenie

Celem pracy było zaprojektowanie aplikacji oraz urządzenia pozwalającego na komunikację z wyświetlaczem LCD zegarów samochodowych oraz uruchomienie na niej dowolnej aplikacji dedykowanej specjalnie pod nie.

Zakres pracy obejmował projektowanie oraz implementację oprogramowania w języku PHP umożliwiającego dwustronną komunikację z dowolnymi urządzeniami wyjścia oraz wejścia, zdefiniowanymi poprzez sterowniki wybrane przez użytkownika. Obejmował również napisanie przykładowej aplikacji współpracującej z tym oprogramowaniem, wybranie medium transmisji z ekranem LCD zegarów samochodowych, zaprojektowania, zbudowania oraz poprawnego ich połączenia. Jako główny mikrokomputer nadzorujący całość użyto komputera Raspberry Pi, a jako wsparcie medium połączenia z zegarami wybrano Arduino UNO. Na samym końcu zaprojektowano oraz zbudowano stronę internetową z nastawieniem na urządzenia mobilne która umożliwia zmianę ustawień aplikacji.

Cele przedstawione w pracy zrealizowano zgodnie z przyjętymi założeniami, które zostały zweryfikowane podczas prototypowania, a końcowym efektem pracy jest urządzenie które umożliwia rozbudowę samochodów BMW E34 oraz E32 z zegarami typu HIGH o nowe rozwiązania oraz udogodnienia. Idea oraz schematy urządzenia zostaną udostępnione do wykorzystania przez pasjonatów wcześniej wymienionych samochodów celem własnoręcznej budowy oraz możliwości ulepszenia rozwiązania.

Na podstawie opracowanego rozwiązania warto stwierdzić, że modularny sposób budowy aplikacji bardzo mocno zwiększa możliwości oraz łatwość rozbudowy rozwiązania. Tworzenie własnych sterowników które umożliwiają nie tylko podłączanie dowolnych mediów, jak i również emulowanie poszczególnych skrajnych przypadków drastycznie skraca czas wymagany na debuggowanie rozwinięć aplikacji jak i trud jej testowania.

Odpowiedni wybór medium transmisji z szyną DAC zegarów samochodowych był niezwykle ważny. Wszelkie próby połączenia bezpośrednio z Raspberry Pi oraz próba obsługi połączenia za pomocą języka PHP z biblioteką PHPSerial zakończyły się niepowodzeniem. Zegary wymagają bardzo dużej precyzji przesyłanych danych, co w połączeniu w językiem PHP nie współgra za dobrze. Dlatego oddelegowano tę

odpowiedzialność dla mikrokomputera Arduino. Kod napisany w języku C spełnił swoje zadanie. Kolejnym bardzo ważnym elementem jest dobór odpowiedniego konwertera RS232/UART. Zwykła konwersja 3.3V do 5V jest niewystarczająca, gdyż zegary szynę DAC obsługują napięciem 12V.

Całość można uznać za wersję wczesną, dosyć eksperymentalną. Wynika to z faktu otwartości rozwiązania oraz chęci konsultacji całości z programistyczną społecznością retro-samochodów. Bycie zamkniętym na wszelkie idee może przyćmić przydatność społeczną projektu, a to właśnie ta cecha przyświeca całości od początku. Jednakże można wspomnieć, że główne założenia pracy zostały spełnione.

Przewidywanym dalszym kierunkiem rozwoju pracy na pewno jest rozbudowa panelu aplikacji mobilnej o statusy urządzenia oraz możliwości ustawiania właściwości sieciowych. Dodatkowa implementacja sterowników, z myślą o czym właśnie została napisana aplikacja również jest możliwa. Głównie pożądanym sterownikiem wejścia jest sterownik I-BUS oraz oczywiście kolejne implementacje rozwiązań pod aplikację inną niż napisany w ciągu tej pracy Discord.

Literatura

[1] Strona internetowa ncbi.nlm.nih.gov - Lifespans of passenger cars in Europe: empirical modelling of fleet turnover dynamics: https://docs.arduino.cc/tutorials/nano-every/run-4-uart – dostęp 03.02.2023

[2] Strona internetowa i-code.net – Injecting UART Messages into the BMW 750iL Instrument Cluster LCD: https://i-code.net/injecting-custom-uart-messages-into-the-bmw-750il-instrument-cluster-lcd/ – dostęp 04.02.2023

[3] Strona internetowa docs.arduino.cc – Communicating with Four Boards Through UART with Nano Every: https://docs.arduino.cc/tutorials/nano-every/run-4-uart – dostęp 10.02.2023

[4] Strona internetowa peterfleury.epizy.com - Serial Port Interface for an AVR: http://www.peterfleury.epizy.com/avr-uart.html - dostęp 10.02.2023

[5] Strona internetowa github.com - DiscordPHP : https://github.com/discord-php/DiscordPHP - dostęp 11.02.2023