

Raspberry Pi BASIC

Aleksandra Kosińska

Cezary Stajszczyk

Praca inżynierska

Promotor: dr Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

31 sierpnia 2023

Streszczenie

Celem niniejszej pracy jest implementacja systemu z interpreterem języka programowania BASIC działającego na mikrokomputerach z serii Raspberry Pi w wersji 3. i 4. Zaproponowany dialekt bazuje na tym wykorzystywanym w latach '80 w popularnych komputerach Commodore 64. Program nie tylko umożliwia pracę w starym stylu na współczesnym sprzęcie, ale także prezentuje możliwości jakie dają te popularne płytki i pokazuje jak wykorzystać je programując bare-metal, co stanowi większe wyzwanie niż w przypadku gdy jest ona kontrolowana przez dedykowany system operacyjny Raspbian. Wykorzystując wbudowane złącza projekt wspiera obsługę karty SD oraz wyjście wideo HDMI, a także poprzez wyprowadzone piny GPIO umożliwia podłączenie zewnętrznej klawiatury ze złączem mini-DIN6 lub DIN5.

The aim of this thesis is to implement a system with a BASIC programming language interpreter that runs on Raspberry Pi series microcomputers versions 3. and 4. The proposed dialect is based on the one used in the '80s in the popular Commodore 64 computers. The program not only allows retro-style operation on modern hardware, but also demonstrates the capabilities of these popular boards. In addition, it presents how to take advantage of the Raspberry Pi's features by programming bare-metal, which is rather more demanding than when managed by a dedicated Raspbian operating system. Using the built-in connectors, the project provides support for SD card and HDMI video output, along with the ability to connect an external keyboard with a mini-DIN6 or DIN5 port through the derived GPIO pins.

Spis treści

Wstęp	7
1. Podstawy projektu	9
1.1. Budowa projektu	9
1.1.1. Cross-compiler	10
1.1.2. Struktura katalogów	10
1.1.3. Proces budowy	10
1.1.4. Armstub	10
1.2. Przygotowanie karty pamięci	11
1.2.1. Niezbędne pliki	11
1.3. Bootowanie	12
1.3.1. Funkcja pliku config.txt	12
1.3.2. Proces bootowania RaspberryPi	12
1.3.3. Sygnalizacja statusu bootowania za pomocą LED	12
1.4. Debugowanie	13
1.4.1. Mechanizmy obsługi przerwań i wyjątków	13
1.4.2. Deasemblacja	13
2. BASIC	15
2.1. Sesja	15
2.1.1. Struktury Modułu Session	16
2.1.2. Operacje na strukturze sessionS	17
2.2. Parser	17

2.3.	Interpreter	17
2.3.1.	Tryb bezpośredni	18
2.4.	Ewaluator	18
2.4.1.	Algorytm ewaluacji wyrażeń	18
2.4.2.	Ewaluacja poszczególnych tokenów	19
2.4.3.	Istotność kolejności ewaluowania wyrażeń	19
2.4.4.	Sprawdzenie poprawności wyrażenia	19
3.	Pamięć	21
3.1.	Zarządzanie pamięcią	21
3.1.1.	Sztywna i dynamiczna alokacja pamięci	22
3.1.2.	Narzędzia ułatwiające pracę z pamięcią	22
3.2.	MMU	23
3.3.	DMA	23
3.3.1.	Proces włączenia DMA w Raspberry PI	23
3.3.2.	Proces wysyłania danych przez DMA	24
4.	Komunikacja	25
4.1.	GPIO	25
4.2.	UART	26
4.3.	Klawiatura	26
4.4.	HDMI	28
4.4.1.	Inicjalizacja i konfiguracja HDMI	28
4.4.2.	Aktualizacja bufora i wydajność	29
4.5.	Karta SD	29
4.5.1.	Komunikacja z kartą SD	30
4.5.2.	Przechowywanie danych	31
4.5.3.	Obsługa plików	32
	Podsumowanie	33
	Bibliografia	35

Wstęp

Dynamiczne tempo rozwoju w dziedzinie informatyki prowadzi do powstawania nowoczesnych technologii oraz narzędzi programistycznych. Niemniej jednak, fundamentalne zrozumienie działania systemów komputerowych na jak najniższej warstwie abstrakcji zachowuje swoją edukacyjną i badawczą wartość. Celem niniejszej pracy inżynierskiej jest zaprojektowanie oraz implementacja systemu operacyjnego działającego w środowisku bare-metal na platformie RaspberryPi, umożliwiającego tworzenie i uruchamianie programów napisanych w języku BASIC. Zainteresowanie programowaniem niskopoziomym, systemami wbudowanymi oraz retroinformatyką wpłynęło na wybór tematu pracy. Przyjęty temat pracy umożliwia pogłębienie wiedzy w zakresie zarządzania sprzętem w środowisku pozbawionym klasycznego systemu operacyjnego oraz analizę wyzwań związanych z implementacją interpretera języka programowania na takim systemie.

Rozdział 1.

Podstawy projektu

Raspberry Pi to nazwa serii komputerów jednopłytkowych stworzonych przez brytyjską organizację charytatywną, której celem jest edukowanie ludzi w zakresie informatyki i ułatwianie dostępu do edukacji informatycznej. Oryginalnie Raspberry Pi zostało wyprodukowane z myślą o języku Python, jednak w prezentowanej pracy obrano zupełnie inne podejście do użytkowania tego urządzenia. Skupiono się na programowaniu niskopoziomowym - tworząc kod w języku C i Asembler który działa na sprzęcie bezpośrednio, bez żadnej podstawowej abstrakcji, takiej jak system operacyjny.

1.1. Budowa projektu

Programowanie systemów wbudowanych, stanowi wyjątkowe wyzwanie inżynierijne. Charakter tego typu projektów wymaga nie tylko zaawansowanej wiedzy technicznej, ale również wykorzystania odpowiednich narzędzi, które uwzględniają specyfikę docelowej platformy sprzętowej. Przy niewłaściwym podejściu istnieje ryzyko, że RaspberryPi nie uruchomi się w sposób zamierzony. W niniejszym rozdziale przedstawiona zostanie struktura oraz proces konstrukcji projektu systemu niskopoziomowego przeznaczonego dla platformy RaspberryPi.



Rysunek 1.1: Płytki Raspberry Pi 3B

1.1.1. Cross-compiler

W ramach projektowania systemów niskopoziomowych dla specyficznych platform, takich jak RaspberryPi, kluczowym narzędziem jest cross-compiler (kompilator skrośny). Ze względu na to, że architektura procesorów RaspberryPi różni się od konwencjonalnych komputerów, kompilacja bezpośrednio na nich byłaby nieefektywna. Cross-compiler, jak nazwa wskazuje, pozwala kompilować kod na jednej platformie tak, aby był on przeznaczony do wykonania na innej. W niniejszym projekcie, jako kompilator skrośny został wykorzystany `aarch64-linux-gnu`, który jest dedykowany dla architektury ARM.

1.1.2. Struktura katalogów

Projekt jest zorganizowany w przejrzystą strukturę katalogów. Katalog `src` zawiera pliki źródłowe z rozszerzeniem `.c` i `.S`, definiujące logiczną strukturę projektu. Natomiast, katalog `include` gromadzi pliki nagłówkowe `.h`, które umożliwiają deklaracje funkcji, struktur danych oraz innych kluczowych elementów używanych w kodzie źródłowym.

1.1.3. Proces budowy

Ważnym elementem każdego projektu programistycznego jest automatyzacja procesu budowy. W tym celu zostało wykorzystane narzędzie `Makefile`. Za jego pośrednictwem pliki źródłowe są kompilowane w kolejności, następnie łączone, tworząc finalny plik wykonywalny w formacie `.elf`. Zastosowanie narzędzia `objcopy` umożliwia transformację tego pliku do końcowego obrazu systemu `kernel8.img`. Linkowanie to proces łączenia plików obiektowych oraz bibliotek w jednolitą całość, składającą się na plik wykonywalny. W przedstawionym projekcie jest to kluczowy krok, który pozwala na właściwe zorganizowanie segmentów pamięci, zapewnienie poprawnych odwołań do funkcji oraz właściwą inicjalizację systemu.

1.1.4. Armstub

W `Makefile`, oprócz zdefiniowanych procedur kompilacji, zawarte są również instrukcje dotyczące tworzenia `armstub`'a. W omawianym projekcie odgrywa on niezwykle istotną rolę. Jest to składnik, który pozwala na zapewnienie kompatybilności między różnymi wersjami RaspberryPi. Dzięki użyciu własnego `armstub`'a, możliwe jest ujednolicenie procesu bootowania niezależnie od specyfikacji sprzętowej poszczególnych wersji platformy.

1.2. Przygotowanie karty pamięci

Aby Raspberry Pi mogło uruchomić napisany program, należy najpierw odpowiednio przygotować kartę pamięci. Pierwszy wpis w MBR musi wskazywać na partycję typu FAT16 lub FAT32. Jej zalecany przez producenta rozmiar to 256MB. Do samego uruchomienia komputera ta jedna partycja jest wystarczająca, ale na potrzeby danego projektu rekomendowane jest utworzenie drugiej partycji do przechowywania danych. Proces ten zostanie szczegółowo opisany w jednym z kolejnych rozdziałów. W internecie można znaleźć listy kompatybilnych kart, jednak producent nie wydał żadnych oficjalnych rekomendacji co do wspieranych typów. W praktyce jednak zdecydowana większość kart nadaje się do pracy z Raspberry Pi.

1.2.1. Niezbędne pliki

Gdy karta SD jest już odpowiednio sformatowana, należy na partycji odpowiedzialnej za bootowanie umieścić odpowiednie pliki:

- **config.txt** - plik z początkową konfiguracją płytki czytany przez bootloader. W komputerach domowych taka początkowa konfiguracja jest zazwyczaj przechowywana w pamięci BIOSu. Raspberry Pi jednak go nie posiada, dlatego konfiguracją przechowywana jest w zwykłym pliku tekstowym;
- **bootcode.bin** - zawiera kod bootloadera. Wykonuje on podstawową konfigurację, a następnie ładuje plik **start.elf**. Jest wymagany tylko na RPi3, a w nowszych wersjach płytki został zastąpiony przez kod zapisany w pamięci EEPROM bezpośrednio na płycie. Można go pozyskać z instalacji zwykłego Raspbiana dostępnego na stronie producenta;
- **start.elf** - kod z firmwarem, odpowiedzialny za uruchomienie GPU oraz przeczytanie i wdrożenie konfiguracji z pliku **config.txt**. Również dostępny z Raspbianem;
- **kernel8.img** - najważniejszy plik zawierający cały kod projektu. Ósemka na końcu nazwy oznacza, że jest to obraz 64-bitowy. Zostanie on domyślnie załadowany pod adres 0x80000 (512KB);
- **armstub.bin** - ten plik w ogólności nie jest niezbędny do uruchomienia mikrokomputera, ale jest szczególnie istotny z perspektywy tego projektu, ponieważ w założeniu projekt ma działać na Raspberry Pi zarówno w wersji 3. jak i 4. Jest to dodatkowa część bootloadera ładowana pod adres 0x0, która wykonywana jest przed kodem z **kernel8.img** i ma na celu ujednolicenie stanu maszyny przed przejściem do kodu głównego, ponieważ domyślnie różne wersje płytek bootują się z nieco innymi konfiguracjami.

1.3. Bootowanie

Bootowanie systemów wbudowanych, takich jak RaspberryPi, to kluczowy proces inicjalizacji, który determinuje, czy system zostanie poprawnie załadowany i uruchomiony. Każdy element tego procesu odgrywa istotną rolę w zapewnieniu właściwej funkcji urządzenia.

1.3.1. Funkcja pliku `config.txt`

Pierwszym istotnym elementem procesu bootowania jest plik `config.txt`. W tym pliku zawarte są kluczowe informacje konfiguracyjne dla bootloadera, które otrzymuje podczas startu systemu. Umożliwia on określenie parametrów takich jak np. częstotliwość taktowania zegara.

1.3.2. Proces bootowania RaspberryPi

Proces bootowania RaspberryPi rozpoczyna się od wczytania z karty SD pliku `bootcode.bin`, który jest odpowiedzialny za inicjalizację podstawowych komponentów układu. Następnie, wczytywany jest plik systemowy `start.elf`, który pełni rolę bootloadera. Gdy proces bootowania osiągnie etap ładowania `kernel8.img`, rozpoczyna się wykonywanie kodu zawartego na początku tego pliku. W przypadku przedstawionego projektu, jest to kod źródłowy pochodzący z `boot.S`. Ten fragment kodu Asemblera odpowiada za inicjalizację kluczowych elementów systemu, takich jak ustawienie odpowiedniego poziomu wyjątku, aktywacja jednostki Floating-Point Unit oraz dezaktywacja nieużywanych rdzeni procesora. Jego rozmieszczenie na początku obrazu jest gwarantowane przez instrukcje zawarte w pliku `linker.ld`. Po pomyślnym przejściu przez te wstępne operacje, kontrola jest przekazywana do funkcji `main()`.

1.3.3. Sygnalizacja statusu bootowania za pomocą LED

Wbudowany mechanizm diagnostyczny RaspberryPi pozwala na szybką identyfikację problemów występujących w trakcie bootowania. Sygnalizacja odbywa się za pomocą zintegrowanej zielonej diody LED na płycie RaspberryPi. Znaczenie poszczególnych sekwencji mrugnięć jest następujące:

- ciągle światło - wskazuje, że karta SD została prawidłowo podłączona;
- szybkie mruganie - oznacza aktywny odczyt lub zapis na kartę SD;
- trzy mrugnięcia - sygnalizują niepowodzenie procesu bootowania;
- cztery mrugnięcia - informują o braku pliku `start.elf` na karcie SD;

- siedem mrugnięć - wskazują na brak pliku `kernel.img`;
- osiem lub dziewięć mrugnięć - sygnalizują błąd pamięci SDRAM;
- dziesięć mrugnięć - stan "halt", wskazujący na krytyczny błąd systemu.

1.4. Debugowanie

W środowisku embedded, diagnozowanie oraz wykrywanie błędów jest procesem niezwykle skomplikowanym ze względu na bezpośrednią interakcję napisanego systemu z warstwą sprzętową. Kolejnym utrudnieniem jest brak wsparcia rozbudowanych mechanizmów diagnostycznych, które dostępne są w pełnych systemach operacyjnych.

1.4.1. Mechanizmy obsługi przerwań i wyjątków

W omawianym projekcie, podczas inicjalizacji systemu ustawiany jest poziom uprzywilejowania dostępu na Exception Level 1. Mechanizmy przerwań stanowią kluczowy element w systemach bare-metal, umożliwiając skuteczną komunikację i reakcję na specyficzne zdarzenia generowane przez komponenty sprzętowe. Odpowiednio skonfigurowane procedury pozwalają na obsługę przerwań, które są oczekiwane. Sytuacje nieprzewidziane, które również mogą wystąpić, prowadzą do generowania przerwań niespodziewanych. W takim wypadku, procedura diagnostyczna systemu reaguje poprzez wyświetlenie identyfikatora danego przerwania oraz wartości zawartych w rejestrach `elr_el1` oraz `esr_el1`. Rejestr `elr_el1`, będący skrótem od Exception Link Register, przechowuje adres powrotu po zakończeniu obsługi wyjątku. Z kolei `esr_el1` (Exception Syndrome Register) zawiera informacje na temat przyczyn wystąpienia konkretnego zdarzenia. Dzięki interpretacji danych z tych rejestrów, możliwe jest precyzyjne zidentyfikowanie i rozwiązanie źródła problemu.

1.4.2. Deasemblacja

Plik Makefile zawiera polecenie, które umożliwia generowanie zdezasemblerowanego obrazu pliku `.elf`. Wykorzystując komendę `aarch64-linux-gnu-objdump -D ./build/kernel8.elf > code.txt` uzyskujemy przetworzony kod do formy tekstowej, reprezentującej sekwencje instrukcji assemblerowych.

Rozdział 2.

BASIC

BASIC to język, a w zasadzie cała rodzina języków programowania, której historia rozpoczyna się w 1963 roku w Dartmouth College. Twórcami oryginalnej wersji byli John G. Kemeny oraz Thomas E. Kurtz. Na przestrzeni 60 lat pojawiło się wiele dialektów tego języka, jednak ich ogólne założenia, składnia i zbiór instrukcji były do siebie bardzo zbliżone. Dialekt zaproponowany w tej pracy bazuje na tym wykorzystywanym w popularnych komputerach Commodore 64, jednak nie jest jego dokładnym odwzorowaniem.

```
10 PRINT "FIBONACCI"  
20 INPUT "N?" N  
30 DIM TAB[N]  
40 LET TAB[0] = 1 : TAB[1] = 1  
50 FOR I=2 TO N-1  
60 LET TAB[I] = TAB[I-1] + TAB[I-2]  
70 NEXT  
80 PRINT "FIB(";N;")=";TAB[N-1]  
90 END
```

Struktura programu wygląda inaczej niż obecnie pisane w zwykłym edytorze tekstu. Każda instrukcja poprzedzana jest odpowiadającym jej numerem linii. Zwyczajowo instrukcje numerowane są kolejnymi wielokrotnościami liczby 10, aby pozostawić miejsce na dodawanie nowych instrukcji pomiędzy już istniejące. Dodanie instrukcji o już istniejącym numerze nadpisuje ją. Moduły odpowiedzialne za obsługę BASIC'a zostały bardziej szczegółowo opisane w następnych podrozdziałach.

2.1. Sesja

W wielu językach programowania, organizacja i zarządzanie danymi są kluczem do skutecznej i efektywnej egzekucji programu. W przedstawionym projekcie wpro-

wadzony został moduł o nazwie “Session”, który pełni znaczącą rolę w tym zakresie. Jest on odpowiedzialny nie tylko za przechowywanie wszystkich danych związanych z programem ale również za trzymanie informacji o aktualnym stanie wykonywania programu i ewentualnych kodach błędu. Ponadto jego rolą jest prawidłowe przekazywanie zapisanych instrukcji do interpretera zważając na ich kolejność oraz potencjalne skoki. Funkcjonowanie modułu ma bezpośredni wpływ na zdolność użytkownika do pisania, uruchamiania, czy modyfikowania programów w języku BASIC. W związku z tym, moduł “Session” jest jednym z najważniejszych elementów w części systemu, która obsługuje interpretację języka programowania BASIC.

2.1.1. Struktury Modułu Session

Głównym elementem modułu “Session” jest struktura `sessionS`. Została ona zdefiniowana następująco:

```
typedef struct sessionS {
    metadataS metadata;
    forS for_stack[8];
    u64 return_address_stack[32];
    dataQueueS data_queue[128];
    variableS variables[64];
    functionS functions[64];
} sessionS;
```

Język BASIC udostępnia instrukcje, które mogą skakać do wyznaczonych linii programu, dlatego potrzebny jest stos adresów powrotu, który wykorzystany został do zapamiętywania numerów linii w celu prawidłowej kolejności wykonania programu. Kolejnym z komponentów jest kolejka danych, z której użytkownik może ściągać i odkładać zmienne za pomocą dedykowanych instrukcji. W środowisku sesji znajdują się również zapisane funkcje oraz zmienne, które w swoich strukturach oprócz wartości przechowują zdefiniowane w programie nazwy, aby można się było do nich odwołać. Kluczowym elementem są metadane, które zawierają wskaźniki umożliwiające edytowanie oraz poruszanie się po liście instrukcji. Struktura `metadataS` zawiera również informacje o bieżącym stanie sesji, występującym kodzie błędu (znacząco ułatwia obsługę i diagnostykę problemów) oraz szczegółowe informacje podstruktur `sessionS` (np. liczba zmiennych). Ważnym elementem jest flaga skoku, której celem jest poinformowanie modułu o ewentualnej zmianie kolejności wykonywanych instrukcji. Metadane te są niezbędne dla wielu funkcji zawartych w tym module, umożliwiając im prawidłowe działanie. Instrukcje programu są przechowywane w formie listy dwukierunkowej. Każdy węzeł tej listy, oprócz wskaźników na następny i poprzedni węzeł, przechowuje również numer linii oraz treść instrukcji jako ciąg znaków.

2.1.2. Operacje na strukturze sessionS

Zarządzanie sesją w opisywanym module nie ogranicza się wyłącznie do przechowywania danych. Zawiera on również różnego rodzaju funkcje, które umożliwiają efektywne operowanie na tej strukturze. Funkcje w *Session* dają możliwość dodawania, modyfikowania i usuwania elementów takich jak: zmienne, funkcje oraz instrukcje. Umożliwiają także przeprowadzanie operacji na kolejce danych zgodnie z wymaganiami języka BASIC. Co więcej, moduł pozwala na uruchomienie całego programu, co jest jego najważniejszą czynnością. Ostatecznie wszystkie operacje dostępne w module tworzą pełny zestaw fundamentów, niezbędny do przetwarzania i wykonywania kodu w języku BASIC. Dzięki temu użytkownik może w pełni korzystać z możliwości interpretera, mając pewność, że każda operacja jest przeprowadzana w sposób prawidłowy.

2.2. Parser

Moduł parsera jest zdecydowanie mniejszy i prostszy od pozostałych. Jego główna funkcja `get_next_token()` ma za zadanie dla podanego ciągu znaków zwrócić ID tokenu występującego na początku i przesunąć wskaźnik za przeczytany token. W przypadku niektórych tokenów (na przykład liczby lub nazwy zmiennych) oprócz ID tokenu niezbędna jest również znajomość jego dokładnej wartości, dlatego przeczytany ciąg znaków jest kopiowany do bufora podanego jako argument funkcji. Kolejność sprawdzania tokenów jest następująca: znak końca linii, liczby (również te w formacie szesnastkowym), słowa kluczowe i operatory, a na końcu nazwy zmiennych i funkcji. Kolejność ta powoduje, że stworzenie zmiennej o nazwie będącej słowem kluczowym jest niemożliwe - parser nigdy nie rozpozna tego jako nazwę zmiennej. Parser ma również wbudowaną funkcjonalność zgłaszania błędów - gdy użytkownik wie, jakiego konkretnego tokenu się spodziewa, może podać jego ID jako argument funkcji. Jeżeli odnaleziony token nie będzie pasować, parser sam zgłosi błąd.

2.3. Interpreter

Zadaniem interpretera jest wykonanie instrukcji na podstawie ciągu tokenów zwracanych mu przez parser. Skutkiem wykonania instrukcji mogą być dwie rzeczy: zmiana struktury sesji (zdecydowana większość przypadków) lub wydrukowanie czegoś na ekranie (na przykład z pomocą instrukcji `PRINT`, ale mogą to też być informacje o błędach). Do wykonania każdej z dostępnych instrukcji napisana została osobna funkcja. Konstrukcja BASIC'a jest na tyle prosta, że już po pierwszym przeczytaniu tokena wiadomo jakiej instrukcji się spodziewać. Główna funkcja czyta

pierwszy token, a następnie na jego podstawie wybiera odpowiednią funkcję i przekazuje jej ciąg znaków będący pozostałą częścią instrukcji.

2.3.1. Tryb bezpośredni

Oprócz możliwości pisania całych programów możliwe jest również wykonywanie instrukcji w tak zwanym trybie bezpośrednim. Instrukcja wpisana w trybie bezpośrednim poznawana jest po tym, że nie jest poprzedzona numerem linii. Taka instrukcja nie jest nigdzie zapisywana, a jest wykonywana od razu. Daje możliwość na przykład opowiedniego przygotowania środowiska przed uruchomieniem programu i jest niezwykle przydatne podczas debugowania. W dowolnym momencie programu można się zatrzymać instrukcją `STOP`, a następnie przeprowadzić inspekcję w trybie bezpośrednim, skorygować ewentualne błędy i wznowić działanie programu wpisując `CONT`. Niektóre instrukcje (takie jak wspomniane `CONT` lub `RUN`) można wpisywać tylko w trybie bezpośrednim.

2.4. Ewaluator

Język BASIC dopuszcza używanie wyrażeń arytmetyczno-logicznych jako argument słów kluczowych. Oznacza to, że tak jak w większości innych języków programowania, możemy np. zdefiniować zmienną za pomocą działania arytmetycznego (`LET x = 1+1`). Z tego powodu w projekcie potrzebny jest moduł odpowiedzialny za obliczanie wartości wyrażeń.

2.4.1. Algorytm ewaluacji wyrażeń

Ewaluator jako jeden z argumentów - w postaci ciągu znaków - otrzymuje wyrażenie do obliczenia. Korzystając z funkcji Parsera `get_next_token` pobiera kolejno tokeny z danego działania. Następnie na podstawie zwróconego ciągu znaków oraz odpowiedniego mu tokena określa ich typ oraz je ewaluuje. Wówczas sprawdzany jest wyznaczony typ. Jeżeli jest operatorem to przechodzi przez ciąg instrukcji sprawdzający, czy należy go przerzucić na stos wynikowy. Gdy warunek jest spełniony to ze stosu ściągana jest odpowiednio jedna lub dwie wartości w zależności czy jest to operator jedno lub dwuargumentowy. Następnie wyliczany jest wynik wyrażenia z jego udziałem. Obliczona wartość działania trafia na szczyt stosu. W przeciwnym przypadku, gdy typ jest odpowiednio: liczbą całkowitą, liczbą zmiennoprzecinkową, stringiem lub wartością boolowską to na stos trafia wyewaluowana wartość tokenu. Algorytm wykonuje się dopóki otrzymywane tokeny są dopuszczalne w wyrażeniach lub gdy zostanie wykryty błąd. Po zakończeniu ciągu instrukcji obliczona wartość wyrażenia jako jedyna znajduje się na stosie.

2.4.2. Ewaluacja poszczególnych tokenów

W zależności od otrzymanego tokenu należy użyć innych metod do ewaluowania wartości pozyskanego od Parsera ciągu znaków. Przykładowo, gdy token wskazuje na numer, używana jest własna funkcja zmieniająca string na liczbę. Niezwykle istotne było w tym momencie wykrycie, czy jest to liczba zmiennoprzecinkowa, czy też całkowita, aby zapisać ewaluowane wartości do odpowiedniego typu danych. Należało zwrócić również uwagę na to, czy liczby są ujemne - łatwo to osiągnąć sprawdzając czy pierwszy znak bufora jest równy '-'. Token będący nazwą zmiennej powodował, że konieczne było zajrzenie do środowiska i pobranie wartości z listy zapisanych zmiennych. Gdy token okazał się funkcją lub tablicą to należało rekurencyjnie wywołać się odpowiednio na argumentach lub indeksach tych struktur, które mogły być również wyrażeniami. Ważne jest aby poprawnie obsłużyć każdy przypadek tokenu, ponieważ wszystkie mają unikalne sposoby ich ewaluowania.

2.4.3. Istotność kolejności ewaluowania wyrażeń

We wspomnianym wyżej algorytmie opisana jest sytuacja obliczania działań jedno lub dwuargumentowych z udziałem operatora. Warto wspomnieć, że operatory docelowo są odkładane na swój osobny stos. A czynność ewaluacji odbywa się wtedy i tylko wtedy, gdy operator, na którym wykonujemy instrukcje ma priorytet nie większy niż priorytet operatora zapisanego na stosie operatorów. Taki zabieg pozwala na poprawne obliczanie wyrażeń i zachowanie kolejności działań. Ważnymi operatorami są nawiasy, które również wpływają na hierarchię wykonywanych kalkulacji. Osiągnięcie bezbłędnej ewaluacji nie jest trudne, ponieważ gdy otrzymany operator okaże się prawym nawiasem, to wykonujemy ewaluację działań kolejno ściągając operatory aż nie napotkamy lewego nawiasu.

2.4.4. Sprawdzenie poprawności wyrażenia

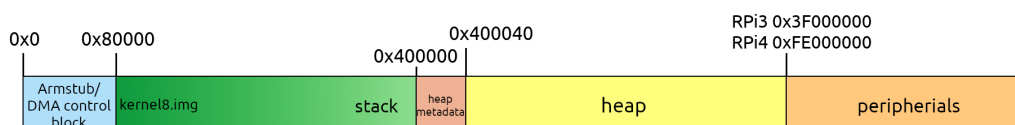
Ważne jest wykrycie, czy operator obsługuje dane typy oraz czy te dwa typy mogą znajdować się w jednym wyrażeniu. Kolejną istotną rzeczą jest sprawdzenie domknięcia nawiasów oraz brak powtarzających się operatorów. Sprawdzanie czy podane wyrażenie jest poprawnie wykonywane odbywa się podczas działania algorytmu. Taki zabieg pozwala na przejście ciągu znaków tylko raz.

Rozdział 3.

Pamięć

Zarządzanie pamięcią w systemach embedded znacznie różni się od podejścia stosowanego w maszynach, z których korzystamy na co dzień. Jedyne co oferuje RaspberryPi zaraz po uruchomieniu to obszar fizycznej pamięci, którą zarządzać trzeba samodzielnie. Nie ma gotowego podziału na stos i stertę, nie ma dynamicznego alokatora pamięci i nie ma żadnych mechanizmów ochrony pamięci. Konieczność tworzenia wszystkiego od podstaw wymaga lepszego zrozumienia działania komputerów, niż w przypadku programowania na zwykły komputer osobisty z systemem operacyjnym.

3.1. Zarządzanie pamięcią



Rysunek 3.1: Mapa pamięci

Powyższa ilustracja pokazuje jak wygląda mapa pamięci stworzonego systemu. Może zwrócić uwagę, że pewne elementy zapisywane są zaczynając od adresu 0x0, który tradycyjnie kojarzony jest z nullpointer'em, na którym wykonanie jakiejkolwiek operacji zapisu lub odczytu kończy się błędem. W przypadku programowania bare-metal tak się jednak nie dzieje. Ponieważ nie ma algorytmów ochrony pamięci, adres 0 traktowany jest tak, jak każdy inny. Jest to w pewien sposób problematyczne, ponieważ niektóre błędy, które normalnie byłyby raportowane automatycznie, tutaj są trudniejsze do znalezienia.

3.1.1. Sterta i dynamiczna alokacja pamięci

Alokator pamięci w tym projekcie został zaprojektowany z naciskiem na prostotę i efektywność. Jego konstrukcja opiera się na liście, składającej się z trzech segmentów: header'a, payload'u oraz footer'a. Header i footer zajmują po 4 bajty i przechowują informacje o rozmiarze danego bloku pamięci oraz jego statusie (zaalokowany czy wolny). Natomiast payload to miejsce, w którym przechowywane są faktyczne dane. Dzięki takiej konstrukcji, alokator jest w stanie skutecznie i efektywnie zarządzać przestrzenią pamięci. Co ważne, cała lista znajduje się bezpośrednio w przestrzeni RAM przeznaczonej na stertę, co pozwala na minimalizację zajmowania miejsca na stosie. Proces alokacji pamięci w opisywanym systemie przebiega następująco: alokator przeszukuje listę w poszukiwaniu pierwszego bloku o rozmiarze nie mniejszym niż żądany. Gdy odpowiedni blok zostaje zidentyfikowany, dzielony jest on na dwa oddzielne bloki. Pierwszy ma rozmiar zgodny z wymaganiami, natomiast drugi reprezentuje resztę niezaalokowanej pamięci. Po dokonaniu podziału, blok spełniający kryteria jest oznaczany jako zaalokowany, a system zwraca do niego wskaźnik. Należy podkreślić, że zwracane wskaźniki są wyrównane do 4 bajtów. Jest to procedura niezbędna ze względu na specyfikę architektury procesora RaspberryPi. Wyrównanie dostępu do pamięci do granic 4 bajtów ma krytyczne znaczenie, ponieważ zapewnia optymalne wykorzystanie zasobów procesora. W projektach typu bare-metal niewyrównane dostępy mogą prowadzić do zbędnych cykli procesora, co z kolei wpłynęłoby negatywnie na wydajność całego systemu. W kontekście zwalniania pamięci, kiedy dany wskaźnik jest uwalniany, odpowiadający mu blok jest oznaczany jako wolny. Alokator następnie sprawdza, czy sąsiednie bloki są również wolne. W przypadku pozytywnej weryfikacji, bloki te są łączone w jeden, co przyczynia się do bardziej efektywnego zarządzania przestrzenią oraz redukcji fragmentacji pamięci.

3.1.2. Narzędzia ułatwiające pracę z pamięcią

Zarządzanie pamięcią w systemach embedded wymaga nie tylko skutecznego alokatora, ale również narzędzi monitorujących i diagnostycznych. Oszczędność miejsca jest kluczowym czynnikiem w środowisku embedded, dlatego każdy niewłaściwie obsługany fragment pamięci należy wykryć. Aby unikać wycieków pamięci, które mogą poważnie zakłócić działanie systemu i prowadzić do nieprzewidywalnych błędów, wprowadzone zostały różne narzędzia diagnostyczne. W środowisku programowania embedded, gdzie dostęp do standardowych narzędzi, takich jak Valgrind jest ograniczony, konieczne stało się stworzenie dedykowanych mechanizmów do debugowania. Jednym z takich narzędzi w projekcie jest moduł o nazwie Lickitung. Jego główne zadanie polega na monitorowaniu i weryfikacji procesu zwalniania pamięci. Po zakończeniu sesji, identyfikowanej przez polecenie `SESSEND`, Lickitung analizuje metadane sterty w poszukiwaniu niezwołnionych bloków. Wykrycie takich bloków skutkuje generowaniem komunikatu o błędzie, informującym o potencjalnym wy-

cieku pamięci, co jest niezwykle pomocne w procesie debugowania. Dodatkowo, dla celów diagnostycznych, wprowadzona została możliwość wizualizacji struktury pamięci za pomocą instrukcji MEM. Ta funkcjonalność pozwala na wydrukowanie elementów sterty, co znacząco ułatwiało proces identyfikacji potencjalnych problemów z zarządzaniem pamięcią.

3.2. MMU

Domyślnie procesory o architekturze AArch64 operują na pamięci w restrykcyjnym trybie **nGnRnE**. Wymaga on między innymi, aby wszystkie dostępy do pamięci były wyrównane do ich rozmiaru. Próba wykonania operacji na niewyrównanym adresie kończyła się wyjątkiem procesora. Okazało się to problematyczne, ponieważ w niektórych przypadkach była potrzeba użycia upakowanych struktur (`__attribute__((packed))`). Aby zmienić tryb i rozwiązać ten problem należy aktywować jednostkę zarządzającą pamięcią (*Memory Management Unit, MMU*). Jednak aktywacja MMU niesie za sobą konieczność utworzenia mapy pamięci. Ponieważ projekt ten działa tylko w trybie uprzywilejowanym i nie zachodzi potrzeba tworzenia wielu przestrzeni wirtualnych istnieje tylko jedno mapowanie - jest to proste mapowanie 1:1.

3.3. DMA

Direct Memory Access, znane jako DMA, odgrywa kluczową rolę w zaawansowanych systemach komputerowych, oferując skuteczne mechanizmy przesyłu danych. W opisywanym projekcie, jednym z głównych celów jest zapewnienie użytkownikowi doskonałej jakości interakcji wizualnej. Odbywa się to dzięki modułowi HDMI, którego wydajność jest ściśle związana z mechanizmami przesyłu danych. Aby zoptymalizować ten proces i zapewnić płynność działania, technologia DMA staje się niezbędna. Umożliwia ona bezpośredni transfer ustalonej ilości bajtów między pamięcią a innymi podzespołami sprzętowymi, eliminując potrzebę pośrednictwa procesora.

3.3.1. Proces włączenia DMA w Raspberry PI

Inicjalizacja DMA na platformie Raspberry Pi rozpoczyna się od wyboru odpowiedniego kanału. W ramach realizowanego projektu zdecydowano się na wykorzystanie kanału o nazwie `DMA_CHANNEL_NORMAL`. Po tej operacji następuje konfiguracja transferu, która obejmuje szereg parametrów. Parametry, takie jak: źródło i cel transferu, rozmiar danych, rodzaj operacji (odczyt czy zapis), wraz z innymi specyfikacjami (ilość danych przeznaczonych do transmisji, prędkość transferu), są ustalane

w celu zapewnienia optymalnej funkcjonalności mechanizmu DMA. Dokładna konfiguracja tych parametrów jest niezbędna, aby zagwarantować skuteczność działania mechanizmu DMA.

3.3.2. Proces wysyłania danych przez DMA

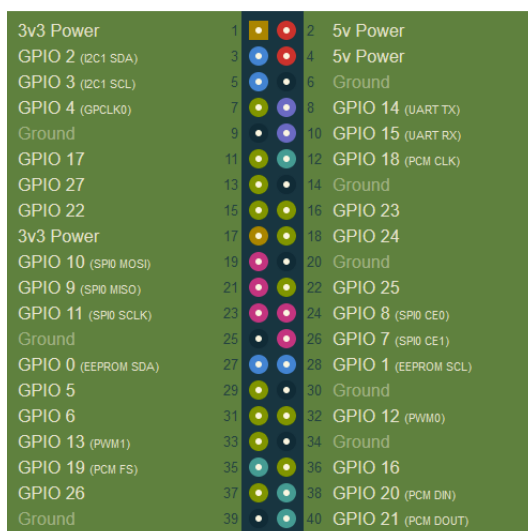
Przesyłanie danych za pomocą DMA odbywa się poprzez skonfigurowane kanały, które operują na zdefiniowanych blokach pamięci. W przypadku modułu HDMI, dane graficzne są przesyłane jako sekwencje pikseli. Wprowadzenie DMA do tego procesu umożliwia buforowanie tych danych, a następnie ich sekwencyjne przesyłanie bez ciągłej interakcji z procesorem. Dzięki DMA, przesył danych odbywa się płynnie, z minimalnymi opóźnieniami, co jest szczególnie ważne przy transmisji w czasie rzeczywistym. Wykorzystanie DMA minimalizuje również ryzyko wystąpienia błędów transmisji, ponieważ dane są przesyłane w kontrolowanych blokach. Ponadto, znacznie zmniejsza to obciążenie procesora, pozwalając mu na jednoczesne wykonywanie innych, równie istotnych zadań. W efekcie, użytkownik otrzymuje obraz o wysokiej jakości i płynności, co poprawia wrażenia z interakcji z systemem.

Rozdział 4.

Komunikacja

Płytki Raspberry Pi, oprócz okazałego zestawu pinów GPIO wyprowadzonych na wlotowane w płytkę złącza goldpin, posiadają również gotowe porty, takie jak HDMI, USB czy slot na kartę micro SD, które można wykorzystać do komunikacji z mikrokomputerem. Niektóre z dostępnych pinów można skonfigurować tak, by obsługiwały pewne popularne protokoły komunikacyjne, takie jak na przykład I2C lub UART. Te rozwiązania znacznie ułatwiają stworzenie interfejsu do komunikacji człowiek - komputer.

4.1. GPIO



3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

Rysunek 4.1: Piny GPIO na płytce RPi3/RPi4

W projekcie zaimplementowany został moduł umożliwiający wygodne korzystanie z pinów GPIO. Pozwala on na przełączanie pinów w wybrane tryby, a także odczytywanie i ustawianie stanów na wybranych pinach. Oprócz standardowych trybów INPUT oraz OUTPUT niektóre piny o specjalnym przeznaczeniu mają również tak

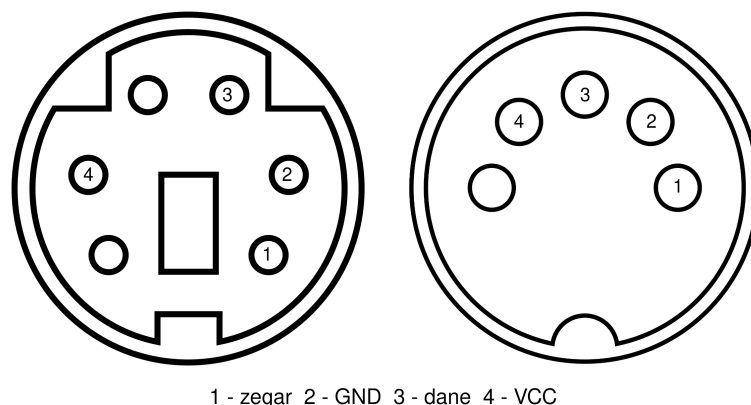
zwane „funkcje alternatywne”. Odpowiadają one za połączenie pinów z wewnętrznymi układami, na przykład odpowiedzialnymi za komunikację przez specjalne protokoły.

4.2. UART

Najprostszym sposobem na komunikację z RaspberryPi jest UART. Znany i sprawdzony protokół pozwalający na dwustonną komunikację, w procesie tworzenia projektu sprawdzał się znakomicie jako podstawowe urządzenie wejścia-wyjścia. W celu połączenia z komputerem niezbędny jest konwerter UART-USB. Raspberry Pi posiada tak naprawdę dwa interfejsy: Mini UART oraz PL011 UART. W tym przypadku konieczne jest skorzystanie z wersji mini, ponieważ to ona ma piny TX oraz RX wyprowadzone na zewnętrzne złącze GPIO, do którego można się podłączyć przewodami. Układ PL011 korzysta z wewnętrznych pinów i jest połączony z wewnętrznymi peryferiami, które nie są wykorzystywane w tym projekcie. Mini UART posiada własne przerwania, a to właśnie na przerwaniach oparty jest system wejścia w tym systemie. Po wejściu do procedury obsługi przerwania z odpowiedniego rejestru odczytywana jest wartość, która została odebrana, a następnie wysyłana do odpowiedniej funkcji modułu IO, gdzie jest odpowiednio obsługiwana. Jeżeli wartość odpowiada zwykłemu znakowi drukowalnemu, jest wysyłana z powrotem przez UART, aby użytkownik widział co wpisuje. Specjalnej obsługi natomiast wymagają backspace oraz enter. W przypadku tego pierwszego, z bufora danych wejściowych usuwany jest ostatni znak, następnie przez UART wysyłany jest ciąg spacji poprzedzony znakiem `\r` celem wyczyszczenia całej linii i ponownie drukowana jest zawartość bufora. Natomiast obsługa enter’a sprowadza się do przestawienia flagi `eol`, która mówi funkcji `readline()`, że może już zwrócić odczytany bufor. Korzystanie z UART’a, choć w teorii wystarczające, bo umożliwia komunikację w obydwie strony, jest o tyle uciążliwe, że wymaga drugiego komputera.

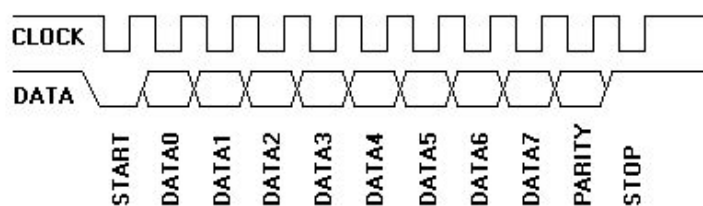
4.3. Klawiatura

Jako samodzielne urządzenie do wprowadzania danych wybór padł na stare klawiatury ze złączem mini-DIN6 lub DIN5. Korzystają one z prostego protokołu wykorzystującego dwa przewody - zegar i dane. Protokół ten jest półduplexowy, jednak na potrzeby tego projektu dwustronna komunikacja nie jest potrzebna, przyjmowanie danych od klawiatury do Raspberry jest w zupełności wystarczające.



Rysunek 4.2: Piny męskich złączy mini-DIN6 oraz DIN5

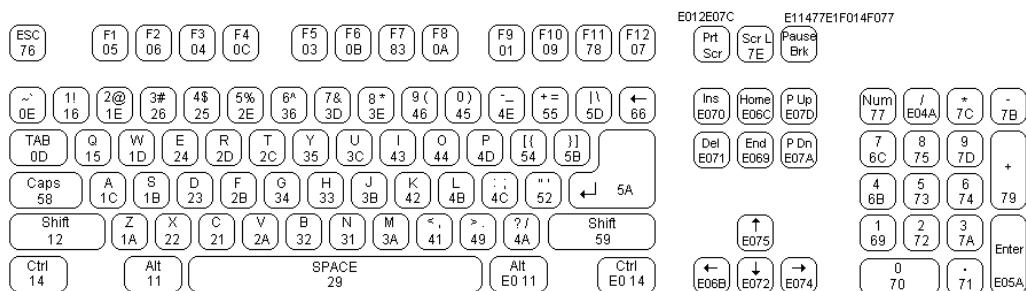
Na jeden kod wysłany przez klawiaturę składa się jedenaście bitów: osiem bitów danych, jeden bit parzystości, oraz po jednym bicie oznaczającym początek oraz koniec transmisji. W celu podłączenie klawiatury do mikrokomputera należy podłączyć przewód danych do pinu nr 5, a przewód zegarowy do pinu nr 6. Na pinie zegarowym włączone jest przerwanie aktywowane zboczem opadającym. Następnie w procedurze obsługi przerwania w pętli następuje odczytanie wszystkich 11 bitów, przekonwertowanie na kod ASCII i wysłanie do odpowiedniej funkcji odpowiadającej za obsługę wejścia, tej samej co w przypadku UART'a.



Rysunek 4.3: Sygnały CLOCK oraz DATA

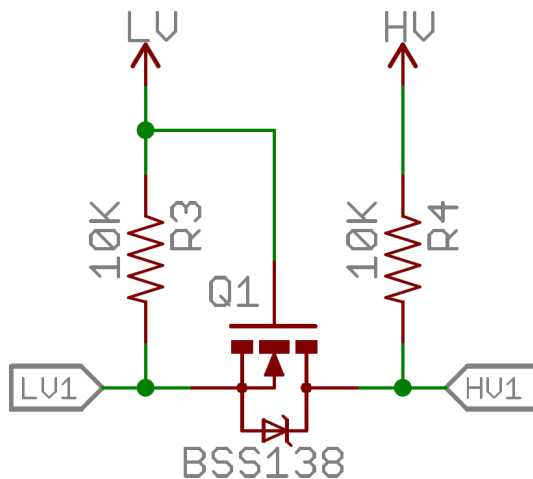
Wysyłane kody nie są jednak kodami ASCII zrozumiałymi dla języka C, należy je więc najpierw na takie przekonwertować. Oprócz kodów przypisanych do poszczególnych przycisków jest jeszcze specjalny kod o numerze 240, oznaczający zwolnienie przycisku o kodzie wysłany jako następny. Tak więc na pojedyncze wciśnięcie klawisza zazwyczaj składają się trzy kody: kod wciśniętego przycisku, 240 oraz ponownie kod wciśniętego przycisku. Dłuższe wciśnięcie jest obsługiwane przez klawiaturę poprzez wielokrotne wysyłanie kodu przycisku przed kodem 240.

Niektóre przyciski, takie jak prawy alt lub strzałki mają kody złożone z dwóch bajtów, z których pierwszy ma wartość 224. Te przyciski jednak nie mają żadnego zastosowania w projekcie, dlatego kod 224 jest pomijany. Z tego powodu wciśnięcie na przykład strzałki w prawo zostanie przez program zrozumiane tak samo, jak wciśnięcie 6 na klawiaturze numerycznej.



Rysunek 4.4: Kody przycisków na klawiaturze

Klawiatury standardowo działają z napięciem zasilania +5V i takie też napięcie wysyłają na piny danych i zegara. Piny GPIO używają napięcia +3.3V. Niektóre klawiatury działają z niższym napięciem, jednak by mieć pewność, że każda klawiatura będzie działać prawidłowo, w projekcie wykorzystany został prosty konwerter poziomów logicznych bazujący na tranzystorze BSS138.



Rysunek 4.5: Konwerter poziomów logicznych

4.4. HDMI

W ramach rozwoju przedstawionego systemu, kluczowym wyzwaniem stało się umożliwienie użytkownikowi wizualnej interakcji z interfejsem. W tym celu niezbędne okazało się zaprojektowanie i wdrożenie dedykowanego modułu HDMI, który odpowiedzialny jest za przesyłanie obrazu do urządzenia wyświetlającego.

4.4.1. Inicjalizacja i konfiguracja HDMI

Inicjalizacja HDMI w Raspberry Pi rozpoczyna się od aktywacji VideoCore za pośrednictwem mechanizmu Mailbox. Mailbox to specyficzny mechanizm komunikacyjny służący do wymiany wiadomości między głównym procesorem a procesorem

VideoCore. Dzięki temu rozwiązaniu możliwa jest dwukierunkowa komunikacja, co umożliwia głównemu procesorowi wysyłanie żądań konfiguracyjnych i otrzymywanie odpowiedzi od VideoCore, dedykowanej jednostki graficznej odpowiadającej za wiele operacji sprzętowych, w tym zarządzaniem sygnałami wideo. Następnie przeprowadzana jest konfiguracja modułu HDMI. Dla potrzeb projektu wybrano rozdzielczość 640x400 pikseli, nawiązując w ten sposób do proporcji charakterystycznych dla komputerów typu Commodore 64. Parametr "bits per pixel", ustalony na wartość 32, oznacza liczbę bitów używanych do reprezentowania koloru pojedynczego piksela (format RGBA). Ustawienie wartości w ten sposób umożliwia precyzyjne odwzorowanie szerokiej gamy kolorów, co przekłada się na wysoką jakość wyświetlanego obrazu.

4.4.2. Aktualizacja bufora i wydajność

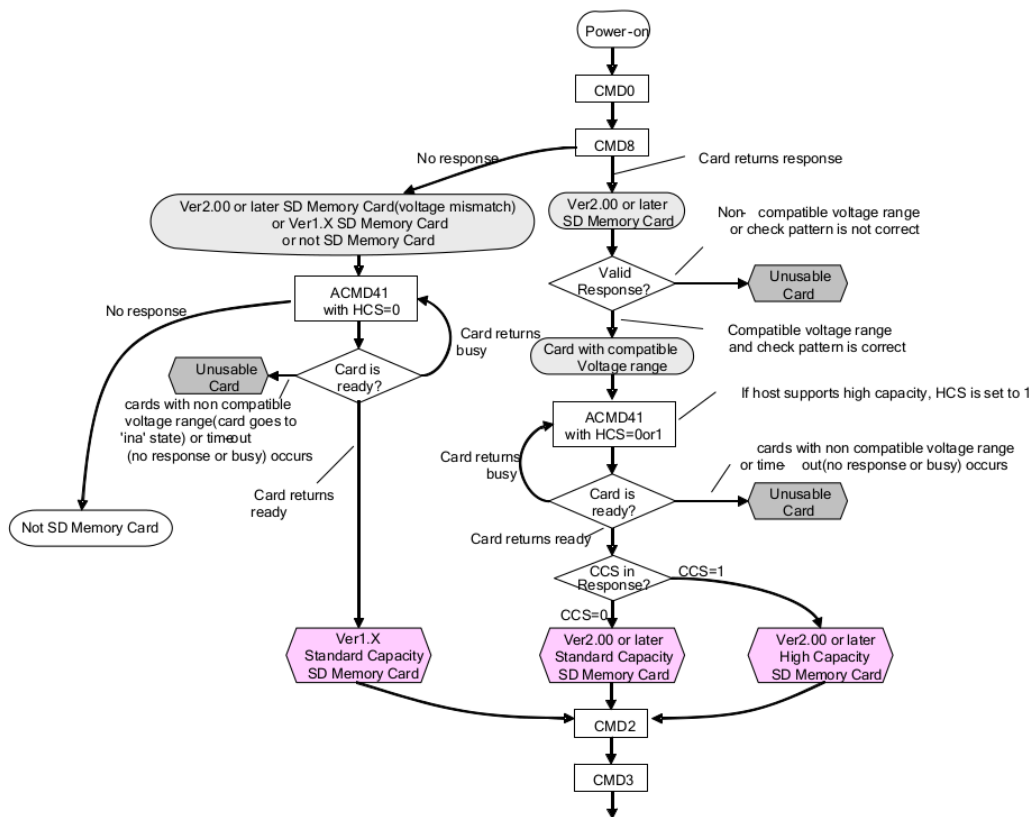
Aby uzyskać odpowiednią jakość wyświetlanego obrazu, kluczowe jest efektywne aktualizowanie ekranu. Proces ten polega na sekwencyjnym przypisywaniu wartości kolorów w formie 32-bitowych liczb reprezentujących poszczególne piksele. Chociaż taka metoda wydaje się prosta, napotyka pewne wyzwania techniczne. Sekwencyjne pisanie każdego piksela bezpośrednio pod wskaźnik bufora ramki może prowadzić do znaczących opóźnień, zwłaszcza gdy konieczne jest odświeżenie całego ekranu. Wynika to z konieczności przetwarzania każdej operacji zapisu oraz komunikacji z pamięcią RAM przez procesor, co wprowadza dodatkowe opóźnienia. Optymalne rozwiązanie tego problemu polega na zastosowaniu opisanej wcześniej technologii DMA. Dodatkowo, dla zwiększenia funkcjonalności, zaimplementowano możliwość scrollowania ekranu w dół. Aby osiągnąć ten efekt, zastosowano bufor cykliczny. Dodatkowo zapamiętywany jest indeks piksela, który służy jako punkt odniesienia do określenia, które dane mają być wyświetlane na początku ekranu podczas przewijania. Bufor cykliczny jest następnie przetwarzany w odpowiedniej kolejności dzięki technologii DMA, tworząc dla użytkownika iluzję ciągłego, nieskończonego przewijania ekranu.

4.5. Karta SD

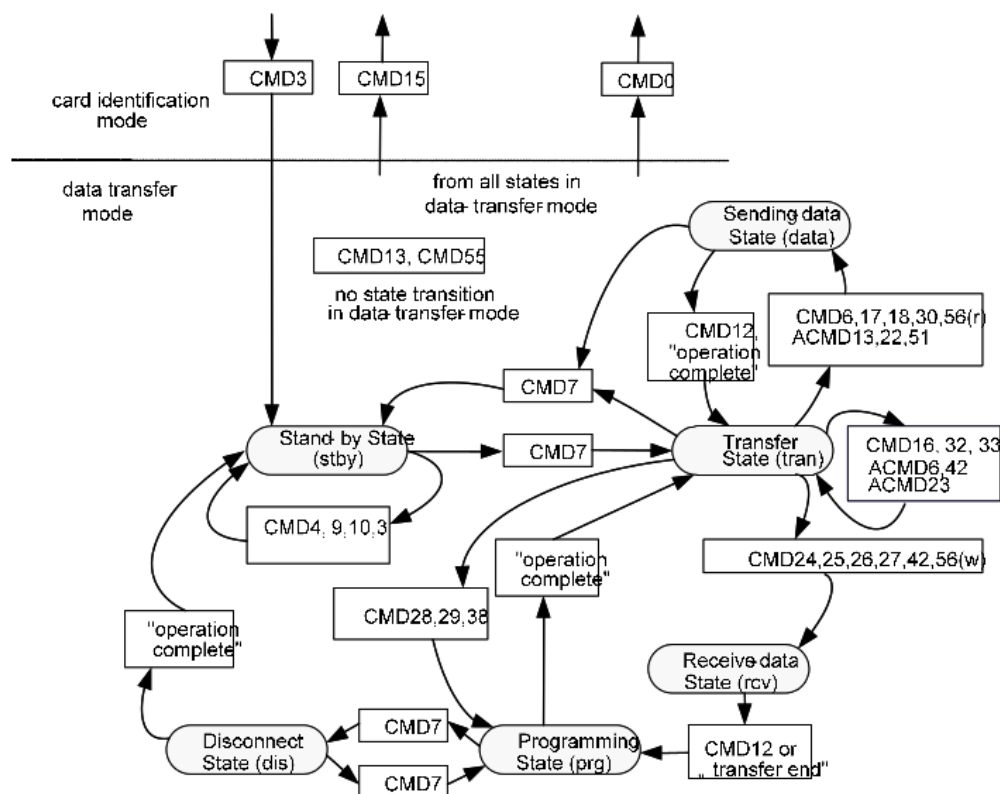
Konieczność pisania nowych programów z każdym uruchomieniem komputera byłaby bardzo uciążliwa, dlatego system wspiera zapisywanie stworzonych programów w zewnętrznej pamięci nieulotnej. Do tego celu wykorzystana została ta sama karta SD, z której ładowany jest program. Aby rozwiązanie zadziałało na karcie prócz partycji niezbędnej do bootowania płytki, musi zostać utworzona druga partycja o dowolnym typie, ponieważ dane przechowywane będą we własnościowym formacie.

4.5.1. Komunikacja z kartą SD

Do komunikacji procesora z kartą posługuje wbudowany w płytkę kontroler zewnętrznej pamięci masowej (.: *External Mass Media Controller, EMMC*). Aby z niego skorzystać, należy najpierw odpowiednio skonfigurować wykorzystywane piny GPIO: piny 34-39 jako INPUT oraz piny 48-52 jako funkcję ALT3. Komunikacja odbywa się za pomocą zestawu rejestrów oraz specjalnych komend wysyłanych przez te rejestry. Podstawowymi komendami są komendy nr 17 oraz 24, odpowiedzialne za odpowiednio odczyt oraz zapis bloku (512MB), a także komenda 0, która resetuje kartę i od której rozpoczyna się cała komunikacja. Poniższy diagram przedstawia proces łączenia się z kartą SD i transfery danych:



Rysunek 4.6: Inicjalizacja i połączenie z kartą SD



Rysunek 4.7: Zapis i odczyt

4.5.2. Przechowywanie danych

Program nie wspiera żadnego ze standardowych systemów plików, a dane przechowuje we własnościowym formacie. Po włączeniu mikrokomputera nastąpi próba inicjalizacji systemu plików. Za partycję danych posłuży pierwsza (po partycji z systemem) partycja znaleziona w głównym rekordzie rozruchowym (textit.: Master Boot Record, MBR), której rozmiar będzie wynosił przynajmniej 1 gigabajt. Pierwszy blok partycji danych przechowywał będzie tablicę plików, której pojedynczy element stanowi następująca struktura:

```
#define FILENAME_LEN 10

typedef struct {
    u32 size_in_bytes;
    u8 id;
    u8 type;
    char name[FILENAME_LEN];
} fileS;
```

Isnienie pliku w danym miejscu w tablicy wskazuje pole **type** - jeżeli jest różne

od zera, plik istnieje. Póki co, wspierane jest jedynie zapisywanie programów w BASICU, więc 8 bitów jest trochę na wyrost, ale jest to furtka na przyszłość do przechowywania w pamięci nieulotnej na przykład całych sesji lub obrazków. Tablica przechowuje dane o 32 plikach. Cała partycja danych dzielona jest na równe kawałki, które kolejno przyporządkowane są wpisom w tablicy plików. Pierwszy wpis jest specjalny i sygnalizuje systemowi istnienie całego systemu plików: jeżeli po wczytaniu pierwszego bloku partycji danych odnaleziony zostanie tam wpis o id równym 0, `type` równym 105 oraz nazwie `SPECIAL`", program potraktuje pierwszy blok jako gotową tablicę plików. W przeciwnym wypadku taka tablica zostanie utworzona.

4.5.3. Obsługa plików

Do korzystania z systemu plików udostępnione zostały następujące komendy:

- `LS` - wypisuje wszystkie pliki zapisane na karcie SD;
- `SAVE name` - zapisuje program z aktualnej sesji do pliku z podaną nazwą;
- `LOAD name` - ładuje program o podanej nazwie z karty SD do aktualnej sesji;
- `DELETE name` - usuwa podany program z karty SD.

W każdej z podanych komend `name` musi być różne od `SPECIAL`" i może mieć co najwyżej 10 znaków

Podsumowanie

Realizacja projektu polegającego na stworzeniu systemu bare-metal dla platformy Raspberry Pi z interpreterem języka BASIC była nie tylko ambitnym przedsięwzięciem, ale również źródłem wielu wyzwań oraz okazją do zdobycia cennych doświadczeń. Podczas pracy nad projektem niejednokrotnie napotymano na różnego rodzaju trudności i przeszkody, które wymagały nie tylko kreatywnego myślenia, ale także głębszego zrozumienia złożonych koncepcji informatycznych. Prace nad takim projektem dają możliwość pogłębienia wiedzy w wielu obszarach informatyki. W tym przypadku, zdobyto doświadczenie z zakresu programowania embedded, zrozumiano głębsze aspekty działania systemów oraz mechanizmów stojących za interpreterami. Znaczący rozmiar projektu pozwolił również na naukę efektywnego zarządzania dużą ilością kodu oraz organizacji pracy w dwuosobowym zespole. Rezultat prezentowanej pracy nie tylko demonstruje efektywne wdrożenie założonych celów, ale również otwiera pole do przyszłej dalszej rozbudowy zaprojektowanego systemu.

Bibliografia

- [1] Raspberry Pi documentation
(<https://www.raspberrypi.com/documentation/>)
- [2] BCM 2837 datasheet (RPi3)
(<https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf>)
- [3] BCM 2711 datasheet (RPi4)
(<https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf>)
- [4] 101 BASIC computer games
(https://annarchive.com/files/Basic_Computer_Games_Microcomputer_Edition.pdf)
- [5] Low Level Devel (<https://github.com/rockytriton/LLD/tree/main>)
- [6] uBASIC: a really simple BASIC interpreter
(<https://github.com/adamdunkels/ubasic>)
- [7] SD and SDIO (<http://yannik520.github.io/sdio.html>)
- [8] Bare Metal RPi4OS (<https://www.rpi4os.com/>)
- [9] Czcionka (<https://github.com/dhepper/font8x8>)
- [10] Mikrokomputery: Commodore 64 Bohdan Frelek