Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Raspberry Pi platform without an operating system

Petr Vanc

Supervisor: prof. Ing. Zahradník Pavel, CSc.
Subfield: Cybernetics and Robotics
May 2019

Acknowledgements

I thank my supervisor prof. Ing. Pavel Zahradník, CSc. for willingness, helpfulness and assistance with work. I thank department of Cybernetics and Robotics for making this work possible.

I also thank Zoltan Baldaszti, Leon de Boer a David Welch for very beneficial posts on Raspberry Pi forum.

Declaration

I declare that I have written this work independently and I have listed all the literature I used.

In Prague, 24. May 2019

Abstract

Purpose of this work is to make overview of working with bare metal applications on platform Raspberry Pi 3 due to insufficient documentation of product and its system on chip Broadcom BCM2837.

This thesis contains descriptions on work with elements of peripherals and features that this device have. But mainly program codes, that can be used to quick start of work on Raspberry Pi 3.

It includes two experiments on getting maximum speed of GPIO output and maximum speed of DA converter.

Keywords: Raspberry Pi, bare metal

Supervisor: prof. Ing. Zahradník Pavel, CSc.
Katedra telekomunikační techniky,
Technická 1902/2,
Praha 6

Abstrakt

Záměr této práce je vytvořit souhrn práce s platformou Raspberry Pi 3 bez operačního systému kvůli chybějící oficiální dokumentaci k chipu Broadcom BCM2837.

Tato práce obsahuje popis jednotlivých elementů periferií a dalších vlastností této desky. Hlavně však vzorové programové úseky pro rychlý start práce na Raspberry Pi 3.

Navíc obsahuje dva experimenty na získání maximální rychlosti GPIO výstupu a maximální rychlosti DA konverteru.

Klíčová slova: Raspberry Pi, bare metal

Překlad názvu: Platforma Raspberry Pi bez použití operačního systému

Contents

Figures

Tables

Part I

Theoretical part

Chapter 1

Theoretical introduction

1.1 What is Raspberry Pi

Raspberry Pi is single board computer in one chip developed in United Kingdom. Its purpose was and is to promote and teach computer science and revive microcomputer revolution of the 1980s. It was originally made for programming in Python language, but in this thesis, I will concentrate on low level programming using language C and Assembler in result of not using operating system.

Specifications between model B:

Version	RPi 1 B+	RPi 2 B v1.2	RPi 3 B+
Instruction set	ARMv6, 32-bit	ARMv8-A, 64-bit	ARMv8-A, 64-bit
SoC (Broadcom)	BCM2835	BCM2837	BCM2837B0
FPU	VFPv2	VFPv4+NEON	VFPv4+NEON
GPU	Broadcom VideoCore IV, 250 MHz		
CPU	ARM1176JZF	4x Cortex-A53	4x Cortex-A53
Frequency	700 MHz	900 MHz	1.4GHz
Memory (SDRAM)	512 MB	1 GB	1 GB

Table 1.1: Specifications of Raspberry Pi models.

All models have are powered by source with voltage 5V. Power is reaching value of 6 Watt with latest model 3B+ under stress. When testing older models I get values topping 3 Watts. Using source with current 1A is sufficient for most cases.

1.2 Raspberry Pi 3

Third version of Raspberry Pi is currently the latest version. This is the version I will be using for experiments and testing.

Figure 1.1: Raspberry Pi 3 model A, B+, [Upt18].

Official specifications:
CPU: ARM Cortex A-53, 1.2GHz (model B+ has 1.4GHz),
Caches: 32kB Level 1 and 512kB Level 2 cache memory
System of chips: Broadcom BCM2837
RAM: 1GB LPDDR2, 900MHz
Peripherals: GPIO 40 pins, HDMI, 3.5mm audio jack, 4X USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI) [Mag]

1.3 Architecture ARM

Architecture ARM is wide spread in the world. For example in most of smartphones, remote devices and now it is getting its way into laptops.

ARM Holdings company is providing Raspberry Pi a central processing unit. It is designing the ARM range of RISC processor cores, computing with reduces instruction set. The ARM company does not fabricate silicon itself. Other companies implements this design in their own architectures. In

Raspberry Pi case it is company Broadcom providing whole package under name system on chip called Broadcom BCM2837 in case of Raspberry Pi version 3B+. [Hol]

ARM architecture is divided at present days between group Cortex-A. These are intended for application use. They have often operating system and they are designed for third party applications. Second group is Cortex-R made for real-time signal processing. Especially where is significant demand for safety. Third and last group is Cortex-M. It is microcontroller-oriented processors for system on chip applications. They are optimized for small size and use in the lowest price chips.

Main processor in Raspberry Pi is made from group Cortex-A.

Besides groups, ARM architecture is also numbered on versions. Between versions ARMv3 and ARMv7 architecture was 32-bit. With ARMv8 comes the possibility to use 64-bit version. As I'm using Raspberry Pi 3B+, there is architecture version ARMv8-A on processor ARM Cortex A53. It uses instruction set AARCH64.

1.3.1 Architecture ARMv8-A

Newest version of ARM architecture covers the Applications profile only. Addition of a 64-bit operating capability is alongside 32-bit execution. Instruction set name is AARCH32 for 32-bit version and AARCH64 for 64-bit version. It is compatible with previous version of architecture.

Length of instruction set is fixed. Instructions are 32-bits in size. There are 31 general purpose registers, that are always accessible and are 64-bits wide. Last general purpose register is dedicated zero register. Program counter and stack pointer are not included in general purpose registers. [Gri]

Besides its older version, architecture ARMv8-A includes SIMD (single instruction, multiple data). The ability of performing the same operation on multiple data points simultaneously. It also has capability of processing numbers with decimal point effectively. There is also new exception model when processing interrupts.

Virtual addresses are stored in 64-bit registers. [Sho15]

Chapter 2

AARCH64 assembly

This chapter will be focused on changes between 32 bit assembly and 64 bit version AARCH64.

2.0.1 Registers

Instead of classic *r* register names, there are *x0-x28* 64-bit registers and their 32-bit version *w0-w28* registers. So there are 29 registers accessible in both 32-bit or 64-bit way. First 8 (*x0-x7*) are used to pass return values. Next ten registers (*x8-x18*) are temporary registers for every function. We cannot say anything about their value when returning from function. Next nine registers (*x19-x28*) are used by a function. Values must be saved when returning from function. [mod18]

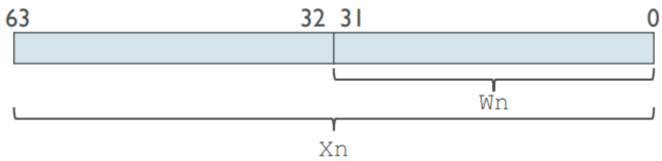

Figure 2.1: Registers in AARCH64 [Sho15].

Name	Size	Description
Wn	32-bits	General purpose registers 0-28
Xn	64-bits	General purpose registers 0-28
WZR	32-bits	Zero register
XZR	64-bits	Zero register
SP	64-bits	Stack pointer

Table 2.1: AACH64 Register table. [mod18]

2.0.2 Branching

Compare two values using instruction *cmp* and then use branching function *b.<cond>*, where <cond> is condition from flag register assigned in comparing instruction. They can be chosen from condition table:

Mnemonic	Description	Condition flags
EQ	Equal	Z set
NE	Not Equal	Z clear
CS or HS	Carry Set	C set
CC or LO	Carry Clear	C clear
MI	Minus	N set
PL	Plus, positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned Higher than or equal	C set and Z clear
LS	Unsigned Less than or equal	C clear or Z set
GE	Signed Greater than or Equal	N and V the same
LT	Signed Less than	N and V differ
GT	Signed Greater than	Z clear, N and V the same
LE	Signed Less than or Equal	Z set, N and V differ

Table 2.2: AARCH64 Branching. [Iba16]

Chapter 3

Cross compilers

3.1 Introduction

To cross-compile is to build on one platform a binary that will run on another platform.

When working with bare metal Raspberry Pi, it is needed to use Cross compiler, because there is no system on Raspberry Pi, that could compile a program as in case of using operating system. [Sys]

All cross compilers are running without any IDE. Advantage is that you don't need any licence and all compilers are free to use.

Wanted result is file in format image with name and extension kernel8.img. This file can be copied into Raspberry Pi 3B+ via micro SD card.

Boot sequence

When Raspberry Pi is turned on, it searches in micro SD card for *bootcode.bin* file. This SD card must be bootable. This can be done with Rufus program on Windows or *UNetbootin* program on Linux. Then Raspberry Pi searches for few files:

bootcode.bin
start.elf
fixup.dat [Foua]

These files can be found for example in *Raspbian* operating system. And they are only responsible for proper boot of image file.

kernel8.img

This image file is then transfered to RAM and bare metal program is ready to be executed. Raspberry Pi does not have BIOS. Instead it has ability to add configuartion file called:

config.txt

In this file there can be specified basic system configurations with very simple file format *property=value*. Value can be integer or string. Comments are made with character #.

Table of useful configuration choices:

Name	Property	Values
GPU memory	gpu_mem	from 0 to 944 (MB)
Disable l2 cache	disable_l2cache	0, 1 (default)
HDMI Safe setup	hdmi_safe	0, 1
GPIO Setup	gpio	0-54=ip,op,a0-a5
ARM Frequence	arm_freq	700-1400 (MHz)

Table 3.1: Configuration file.

Description

GPU memory: As the memory for CPU and GPU is same, this sets memory for GPU, the remaining is for CPU. With 1GB RAM with model 3B+, maximum possible RAM memory assigned to GPU is 944MB.
HDMI Safe setup: Setup HDMI with maximum compatibility. [Foub]
GPIO Setup: Set given GPIOs to input (ip), output (op), alternative function (a0-a5). It can set multiple GPIOs at once. Example: *gpio=0-5=op* sets first five GPIOs to output.
ARM Frequency: Sets Frequency on every processors in MHz.

3.2 GNU compilers

In this section will be introduced types of cross-compilers capable of running programs on Raspberry Pi 3.

3.2.1 Fasmarm compiler

For use of assembly code only, there is easy way to compile via flat assembler service called *Fasm* with ARM specified version *Fasmarm*.

Instalation

First step is to download executable binary version from official page of *Fasmarm*. Second step is to assign global variable in installation folder. This can be done with command:

Listing 3.1: Export path

```
export PATH="$PATH:/<path to installation folder>"
```

<path to installation folder> Alter to your installation folder

Compilation is very easy:

Listing 3.2: Compilation

```
fasmarm <your asm code> <result image file>
```

<your asm code> Represents Assembly code with extension *.asm or *.S. <result image file> Represents image file, that is ready to be load into Raspberry Pi flash memory

3.2.2 AARCH64 elf compiler

Default cross-compiler for most applications is AARCH64 compiler for making executable and link-able files.

Installation

Step 1: Download *binutils*, gcc compiler and other depedencies:

Listing 3.3: Download depedencies

```
wget https://ftpmirror.gnu.org/binutils/binutils-2.30.tar.gz
wget https://ftpmirror.gnu.org/gcc/gcc-8.1.0/gcc-8.1.0.tar.gz
wget https://ftpmirror.gnu.org/mpfr/mpfr-4.0.1.tar.gz
wget https://ftpmirror.gnu.org/gmp/gmp-6.1.2.tar.bz2
wget https://ftpmirror.gnu.org/mpc/mpc-1.1.0.tar.gz
wget https://gcc.gnu.org/pub/gcc/infrastructure/isl-0.18.tar.bz2
wget https://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz
```

Create symbolic values in *binutils* folder:

Listing 3.4: Create symbolic values

```
ln -s ../isl-* isl
```

In *gcc* folder:

Listing 3.5: Create symbolic values

```
ln -s ../isl-* isl
ln -s ../mpfr-* mpfr
ln -s ../gmp-* gmp
ln -s ../mpc-* mpc
ln -s ../cloog-* cloog
```

Listing 3.6: Build binutils

```
mkdir aarch64-binutils
cd aarch64-binutils
../binutils-*/configure --prefix=/usr/local/cross-compiler \
--target=aarch64-elf --enable-shared --enable-threads=posix \
--enable-libmpx --with-system-zlib --with-isl \
--enable-__cxa_atexit --disable-libunwind-exceptions \
--enable-clocale=gnu --disable-libstdcxx-pch \
```

```
--disable-libssp --enable-plugin --disable-linker-build-id \
--enable-lto --enable-install-libiberty \
--with-linker-hash-style=gnu --with-gnu-ld \
--enable-gnu-indirect-function --disable-multilib
--disable-werror \
--enable-checking=release --enable-default-pie \
--enable-default-ssp --enable-gnu-unique-object
make -j4
make install
cd ..
```

Build *gcc* compiler:

Listing 3.7: Build binutils

```
mkdir aarch64-gcc
cd aarch64-gcc
../gcc-*/configure --prefix=/usr/local/cross-compiler \
--target=aarch64-elf --enable-languages=c \
--enable-shared --enable-threads=posix --enable-libmpx \
--with-system-zlib --with-isl --enable-__cxa_atexit \
--disable-libunwind-exceptions --enable-clocale=gnu \
--disable-libstdcxx-pch --disable-libssp --enable-plugin \
--disable-linker-build-id --enable-lto --enable-install-libiberty \
--with-linker-hash-style=gnu --with-gnu-ld \
--enable-gnu-indirect-function --disable-multilib \
--disable-werror --enable-checking=release --enable-default-pie \
--enable-default-ssp --enable-gnu-unique-object
make -j4 all-gcc
make install-gcc
cd ..
```

Create global links:

Listing 3.8: Export path

```
export PATH="$PATH:/usr/local/cross-compiler/bin"
```

3.2.3 ARM Eclipse build plug-in

Second option is ARM Eclipse build plug-in with specified name *ARM-none-eabi*. Main features is high configuration ability and is recommended for bare metal applications. [Ecl19]

Difference between first option *AARCH64-elf* and *ARM-none-eabi* is their application binary interface. It describes how compiler should generate the assembler. Especially how functions should be called, arguments passed, etc. [Joh11]

Installation

Listing 3.9: Installation [Sol16]

```
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
sudo apt-get update
sudo apt-get install gcc-arm-embedded
```

3.3 Working with makefiles

Makefiles are a simple way to organize code compilation. It is needed when using extension files or compiling complex code structures. [Max]

Makefile when cross compiling for ARM device is almost the same as compiling inner computer application. To describe process of making final image file, I will use *AARCH64 elf GNU compiler* as covered before. First step is generating object files from assembler and C files using command *aarch64-elf-gcc*. Second step is to create executable and linkable file with extension *.elf* from created object files and custom made linker file with extension *.ld*. This is done with command *aarch64-elf-ld*. Third and last step is from this *.elf* file create final binary image file using command *aarch64-elf-objcopy*.

This is example of Makefile using starting assembler file *start.S* that will call *main.c*. It uses linker *link.ld*. You can add libraries and extensions contaning c file and header file into folder include. Objects will be saved into obj directory.

Listing 3.10: Makefile

```
STORAGE = <name of SD card>
USER = <user name>
CFLAGS = -Wall -O2 -ffreestanding -nostdinc -nostdlib

all: kernel8.img
```

```
start.o: start.S
        aarch64-elf-gcc $(CFLAGS) -c start.S -o obj/start.o

%.o: include/%.c
        aarch64-elf-gcc $(CFLAGS) -c $< -o obj/$@

main.o: main.c
        aarch64-elf-gcc $(CFLAGS) -c main.c -o obj/main.o

kernel8.img: start.o main.o start.o uart.o
        aarch64-elf-ld -nostdlib -nostartfiles obj/main.o \
        obj/start.o obj/uart.o -T link.ld -o kernel8.elf
        aarch64-elf-objcopy -O binary kernel8.elf kernel8.img

.PHONY: clean

clean:
        rm kernel8.elf
        rm kernel8.img
        rm obj/*.o

run:
        qemu-system-aarch64 -M raspi3 -kernel kernel8.img \
        -serial null -serial stdio

load:
        cp kernel8.img /media/$(USER)/$(STORAGE)
        umount /media/$(USER)/$(STORAGE)
```

Command *make* will clean everything and make final image file.
Command *make run* will run image file in *QEMU* simulator, which is described in capitol 3.1.
Command *make load* will load image on SD card by <name of SD card> and <user name>.

3.4 Linker

Job of linker is to take multiple object files and make from them one executable and link-able file. In linker file with extension **.ld* is described how the sections in the input files should be mapped into the output file. [Obe]

Origin address for 64-bit starts at *0x80000*. *0x8000* for older 32-bit models.

Sample linker file could look like:

Listing 3.11: Sample linker file

```
SECTIONS
{
    . = 0x80000;
    .text :
    {
        . = ALIGN(4);
        ___text_start___ = .;
        _start = .;
        KEEP(*(.text.startup))
        *(.text .text.* .gnu.linkonce.t.*)
        *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
        *(.init .init.*)
        . = ALIGN(4);
        ___text_end___ = .;
    }
    .bss :
    {
        . = ALIGN(4);
        ___bss_start___ = .;
        *(.bss .bss.* .gnu.linkonce.b.*)
        *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
        *(COMMON)
        . = ALIGN(4);
        ___bss_end___ = .;
    }
    .rodata :
    {
        . = ALIGN(4);
        ___rodata_start___ = .;
        *(.rodata .rodata.*)
        *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
        . = ALIGN(4);
        ___rodata_end___ = .;
    }
    .data :
    {
        . = ALIGN(4);       /* Normal data memory is align 8 */
        ___data_start___ = .;
        *(.data .data.* .gnu.linkonce.d.*)
        *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
        . = ALIGN(4);
```

```
        ___data_end___ = .;
    }
    _end = .;

  /DISCARD/ : { *(.comment) *(.gnu*) *(.note*) *(.eh_frame*) }
}
```

Defined sections are *.text* where are executable instructions stored. Next section is *.bss* where are undeclared variables and *.data* section with data itself. Saved marks at every section where section starts and ends are good for memory management used later.

Chapter 4

Troubleshooting and loading program

Easiest way to test program is to emulate Raspberry Pi program.

4.1 Simulating Raspberry Pi using QEMU

QEMU which stands for Quick emulator is open-source service capable of emulating ARM processors. It can emulate many interfaces as well. For example USB, UART, hard disk or display.

Installation

When using linux, installation by using *apt−get install qemu* will install older version of QEMU which does not support Raspberry Pi 3. I prefer installing it from source code.

Listing 4.1: QEMU Installation

```
wget https://download.qemu.org/qemu-4.0.0.tar.xz
tar xvJf qemu-4.0.0.tar.xz
cd qemu-4.0.0
./configure
make
```

Once again export path for running *QEMU* from anywhere.

Listing 4.2: Export path

```
export PATH="$PATH:<path to qemu installation>"
```

From now running Raspberry Pi 3 emulator is done with command:

Listing 4.3: Run QEMU

```
qemu-system-aarch64 -M raspi3 -kernel kernel8.img
```

where *-M* specifies machine. It works when emulating other versions of Raspberry Pi (raspi2, raspi) *-kernel* loads the name of image kernel. For Raspberry Pi version 1 it is kernel.img, For version 2 it is kernel7.img and for version 3 it is kernel8.img.

Other arguments can be:

-serial stdio enables UART communication *-drive file=$(<disk image file>,if=sd,format=raw* adds disk drive

If the program gives right result with emulator, it is time to load program to real device. Loading bare metal program to Raspberry Pi can be done by multiple ways.

4.2 Using SD card

More laborious way to load program is moving micro SD card from PC to Raspberry Pi. This cycle contains:

1. Compile program using command *make*
2. Load program on SD card and un-mount SD card using command *make load* (using Makefile described before)
3. Move SD card from PC slot to Raspberry Pi
4. Turn on Raspberry Pi using power switch

To quit program, it is possible to just power off the Raspberry Pi. Micro SD card reader is not the only source of uploading the program. Another source can be flash drive attached to any USB.

4.3 Bootloaders

Bootloader is service that sends the program into device via some periphery and execute itself. It is composed of two parts. First part is in PC and has to send the image file into device. Second part is in device itself and its work is to receive the program and execute it.

The best way is to represent first part by program *CuteCom*. It can be any other program, that can send file via serial port. Terminal protocol can be *XMODEM* or plain text. Send files can be **.bin* or **.hex*. In linker file is best to divide sections of bootloader and the loaded program.

Working variant of bootloader that I tested can be provided from David Welch. [Wel]

Part II

Practical part

Chapter 5

Basic interfaces

Definition of main base addresses

Definition of main peripherals base addresses.

Address	Name
0x3F000000	periph_base
0x200000	General Purpose IO controller
0x215000	mini UART
0x3000	System Timer
0xB000	Interrupt controller
0xB880	VideoCore mailbox

Table 5.1: Main base addresses table. [Inc12a]

5.1 General purpose input output (GPIO)

There are 54 general-purpose I/O (GPIO) lines split into two banks. All GPIO pins have at least two alternative functions within BCM. The alternate functions are usually peripheral IO and a single peripheral may appear in each bank to allow flexibility on the choice of IO voltage. [Inc12a]

The GPIO peripheral has three dedicated interrupt lines. These lines are triggered by the setting of bits in the event detect status register. Each bank has its own interrupt line with the third line shared between all bits. [Inc12b]

System of chips Broadcom BCM2837 does not have one specific register to change GPIOs. Instead of it, it has two registers. One sets the pins that has true value and ignores false (zero) values. Second resets the pins with also true value and ignores false (zero) values. This means, that setting all output registers to desired value with ones and zeros requires two instructions with write to register memory.

The periphery detects write to this registers with event status register. If is on SET register written the same value, the periphery will detect this writing and can set output GPIO value. Output values of GPIO pins can be only digital. Have only true/false values with voltage 3.3V/0V. **Physical base address:** periph_base + 0x200000

Address	Field Name	Description	Read/Write
0x00	GPFSEL0	GPIO Function Select 0	R/W
0x04	GPFSEL1	GPIO Function Select 1	R/W
0x08	GPFSEL2	GPIO Function Select 2	R/W
0x0C	GPFSEL3	GPIO Function Select 3	R/W
0x10	GPFSEL4	GPIO Function Select 4	R/W
0x14	GPFSEL5	GPIO Function Select 5	R/W
0x1C	GPSET0	GPIO Pin Output Set 0	W
0x20	GPSET1	GPIO Pin Output Set 1	W
0x28	GPCLR0	GPIO Pin Output Clear 0	W
0x2C	GPCLR1	GPIO Pin Output Clear 1	W
0x34	GPLEV0	GPIO Pin Level 0	R
0x38	GPLEV1	GPIO Pin Level 1	R
0x40	GPEDS0	GPIO Pin Event Detect Status 0	R/W
0x44	GPEDS1	GPIO Pin Event Detect Status 1	R/W
0x4C	GPREN0	GPIO Pin Rising Edge Detect 0	R/W
0x50	GPREN1	GPIO Pin Rising Edge Detect 1	R/W
0x58	GPFEN0	GPIO Pin Falling Edge Detect 0	R/W
0x5C	GPFEN1	GPIO Pin Falling Edge Detect 1	R/W
0x7C	GPAREN0	GPIO Pin Async. Rising Edge 0	R/W
0x80	GPAREN1	GPIO Pin Async. Rising Edge 1	R/W
0x88	GPAFEN0	GPIO Pin Async. Falling Edge 0	R/W
0x8C	GPAFEN1	GPIO Pin Async. Falling Edge 1	R/W

Table 5.2: GPIO control registers table [Inc12b].

All registers are 32-bit.

Number	Pull	ALT0	ALT1	ALT2	ALT3
GPIO0	High	SDA0	SA5	<reserved>	
GPIO1	High	SCL0	SA4	<reserved>	
GPIO2	High	SDA1	SA3	<reserved>	
GPIO3	High	SCL1	SA2	<reserved>	
GPIO4	High	GPCLK0	SA1	<reserved>	
GPIO5	High	GPCLK1	SA0	<reserved>	
GPIO6	High	GPCLK2	SOE_N/SE	<reserved>	
GPIO7	High	**SPI0_CE1_N**	SWE_N/SRW_N	<reserved>	
GPIO8	High	**SPI0_CE0_N**	SD0	<reserved>	
GPIO9	Low	**SPI0_MISO**	SD1	<reserved>	
GPIO10	Low	**SPI0_MOSI**	SD2	<reserved>	
GPIO11	Low	**SPI0_SCLK**	SD3	<reserved>	
GPIO12	Low	**PWM0**	SD4	<reserved>	
GPIO13	Low	**PWM1**	SD5	<reserved>	
GPIO14	Low	**TXD0**	SD6	<reserved>	
GPIO15	Low	**RXD0**	SD7	**ALT4 below**	
GPIO16	Low	<reserved>	SD8	**SPI1_CE2_N**	CTS0
GPIO17	Low	<reserved>	SD9	**SPI1_CE1_N**	RTS0
GPIO18	Low	PCM_CLK	SD10	**SPI1_CE0_N**	BSCSL SDA/MOSI
GPIO19	Low	PCM_FS	SD11	**SPI1_MISO**	BSCSL SCL/SCLK
GPIO20	Low	PCM_DIN	SD12	**SPI1_MOSI**	BSCSL/MISO
GPIO21	Low	PCM_DOUT	SD13	**SPI1_SCLK**	BSCSL/CE_N
GPIO22	Low	<reserved>	SD14	**ALT2 below**	
GPIO23	Low	<reserved>	SD15	<reserved>	
GPIO24	Low	<reserved>	SD16	<reserved>	
GPIO25	Low	<reserved>	SD17	<reserved>	
GPIO26	Low	<reserved>	<reserved>	<reserved>	
GPIO27	Low	<reserved>	<reserved>	<reserved>	
GPIO28	-	SDA0	SA5	PCM_CLK	<reserved>
GPIO29	-	SCL0	SA4	PCM_FS	<reserved>
GPIO30	Low	<reserved>	SA3	PCM_DIN	**CTS0**
GPIO31	Low	<reserved>	SA2	PCM_DOUT	**RTS0**
GPIO32	Low	GPCLK0	SA1	<reserved>	**TXD0**
GPIO33	Low	<reserved>	SA0	<reserved>	**RXD0**
GPIO34	High	GPCLK0	SOE_N/SE	<reserved>	<reserved>
GPIO35	High	**SPI0_CE1_N**	SWE_N/SRW_N		<reserved>
GPIO36	High	**SPI0_CE0_N**	SD0	TXD0	<reserved>
GPIO37	Low	**SPI0_MISO**	SD1	RXD0	<reserved>
GPIO38	Low	**SPI0_MOSI**	SD2	RTS0	<reserved>
GPIO39	Low	**SPI0_SCLK**	SD3	CTS0	<reserved>
GPIO40	Low	**PWM0**	SD4		<reserved>
GPIO45	-	**PWM1**	SCL0	SCL1	<reserved>

Table 5.3: GPIO table [Inc12c].

Numbers GPIO0-GPIO39 are contained on board with easy access. I added GPIO40 and GPIO45, which are pins connected to 3.5mm audio jack. *GPCLK0* have function general purpose clock. PWM function is Pulse-width modulation. *TXD* and *RXD* are for UART Transmit Data and Receive Data.

Sample code GPIO Setup

Listing 5.1: GPIO Setup

```
bool gpio_init(uint_fast8_t gpio, GPIO_MODE mode)
{
        if (mode < 0 || mode > GPIO_ALTFUNC3 || gpio > 54)
            return false;
        uint32_t bit = ((gpio % 10) * 3);
        uint32_t reg = GPIO->GPFSEL[gpio / 10];
        reg &= ~(7 << bit);
        reg |= (mode << bit);
        GPIO->GPFSEL[gpio / 10] = reg;
        return true;
}
```

Checks valid GPIO or mode. Creates bit mask, read the register and clears the mode bits and rewrite them to new mode bits. Then write value to register. *GPIO_MODE* is GPIO mode where input is 0, output is 1 and other are alternative functions as can be seen in GPIO table. Register *GPFSEL* can be found in table 5.2.

Sample code GPIO Input

Listing 5.2: GPIO Input

```
bool gpio_inp(uint8_t gpio)
{
        if (gpio > 54)
            return false;
        uint32_t bit = 1 << (gpio % 32);
        uint32_t reg = GPIO->GPLEV[gpio / 32];
        if (reg & bit)
            return true;
        return false;
}
```

Checks valid GPIO. 54 GPIOs are divided into two 32-bit registers. Function will choose register and return true or false by value of pin. Register *GPLEV* can be found in table 5.2.

Sample code GPIO Output

Listing 5.3: GPIO OUTPUT

```
bool gpio_out(uint8_t gpio, bool toggle)
{
        if (gpio > 54)
            return false;
        uint32_t bit = 1 << (gpio % 32);
        if (toggle) {
                GPIO->GPSET[gpio / 32] = bit;
        } else {
                GPIO->GPCLR[gpio / 32] = bit;
        }
        return true;
}
```

Checks valid GPIO. Create bit mask and sets gpio to given value. Register *GPLEV* can be found in table 5.2.

Sample code GPIO Edge detect

Listing 5.4: GPIO Edge detect

```
bool gpio_edge(uint8_t gpio, bool rising)
{
        if (gpio > 54)
            return false;
        uint32_t bit = 1 << (gpio % 32);
        if (rising) {
                else GPIO->GPREN[gpio / 32] = bit;
        } else {
                else GPIO->GPFEN[gpio / 32] = bit;
        }
        return true;
}
```

Checks valid GPIO. Create bit mask. If edge detect is wanted on rising edge, *GPREN* register is used. If it is wanted on falling edge, *GPFEN* register is used. For asynchronous edge detect are used *GPAREN* and *GPAFEN* registers.

5.2 Public timer

System timer peripheral provides four 32-bit timers and one 64-bit timer. Each have output register and compare register with specified value. When the two registers are the same, it triggers given action. [Inc12d] ARM system timers are with base frequency 1MHz. When using 64 bit timer, it is split to between two 32 bit registers.

Physical base address: periph_base + 0x3000

Addr. Offset	Reg. Name	Description
0x0	CC	System Timer Control/Status
0x4	CLO	System Timer Counter Lower 32 bits
0x8	CHI	System Timer Counter Higher 32 bits
0xC	C0	System Timer Compare 0
0x10	C1	System Timer Compare 1
0x14	C2	System Timer Compare 2
0x18	C3	System Timer Compare 3

Table 5.4: System Timer Registers table. [Inc12d]

Sample code System Timer

Listing 5.5: System Timer

```
uint64_t timer_getClockCount()
{
        uint64_t highReg;
        uint32_t lowReg;
        do {
                highReg = SYSTEMTIMER->CHI;
                lowReg = SYSTEMTIMER->CLO;
        } while (highReg != (uint64_t)SYSTEMTIMER->CHI);
        uint64_t finalReg =
            (uint64_t)highReg << 32 | lowReg;
        return finalReg;
}
```

Because timer is 64-bit, it is read from high 32-bit register *CHI* and low register *CLO*. Function checks if high register isn't rolling. It returns one composite value. Delay function can be written from this example.

5.3 Universal asynchronous receiver transmitter (UART)

Mini UART is composed only with two pins. First is receiver *RXD* and *TXD* as shown in table 5.3. It can be used with valid clock rate via mailboxes.

Addr. Offset	Reg. Name	Description
0x0	DR	Data Register
0x18	FR	Flag register
0x24	IBRD	Integer Baud rate divisor
0x28	FBRD	Fractional Baud rate divisor
0x2C	LCRH	Line Control register
0x30	CR	Control register
0x34	IFLS	Interupt FIFO Level Select Register
0x38	IMSC	Interupt Mask Set Clear Register
0x3C	RIS	Raw Interupt Status Register
0x40	MIS	Masked Interupt Status Register
0x44	ICR	Interupt Clear Register
0x48	DMACR	DMA Control Register
0x80	ITCR	Test Control register
0x84	ITIP	Integration test input reg
0x88	ITOP	Integration test output reg
0x8C	TDR	Test Data reg

Table 5.5: UART Registers Table. [Inc12e]

All registers are 32-bit.

Sample code UART Initialization

Listing 5.6: UART Initialization

```
void uart_init()
{
    mbox_prop_msg(void, 36, MBOX_REQUEST,
    MBOX_TAG_SETCLKRATE, 12, 8, 2, 4000000, 0);
```

```
    register unsigned int r=*GPFSEL1;
    r&=~((7<<12)|(7<<15));
    r|=(4<<12)|(4<<15);
    *GPFSEL1 = r;
    *GPPUD = 0;
    r=200;
    while(r--) { asm volatile("nop"); }
    *GPPUDCLK0 = (1<<14)|(1<<15);
    r=200;
    while(r--) { asm volatile("nop"); }
    *GPPUDCLK0 = 0;

    *UART0_ICR = 0x7FF;
    *UART0_IBRD = 2;
    *UART0_FBRD = 0xB;
    *UART0_LCRH = 0b11<<5;
    *UART0_CR = 0x301;
}
```

It sends mailbox request for setting clock rate. It maps UART to GPIO pins on GPIO14 and GPIO15 which is ALT0 function. Setting registers from table 5.5. Clear interrupts and setting baud rate to 115200Hz.

Sample code UART Send

Listing 5.7: UART Send

```
void uart_send(unsigned int c) {
    do{ asm volatile("nop"); }
        while(!(*AUX_MU_LSR & 0x20));
    *AUX_MU_IO=c;
}
```

It waits for ready to send data and then write the character to buffer.

Sample code UART Receive

Listing 5.8: UART Receive

```
char uart_getc() {
    do{asm volatile("nop");}
        while(!(*AUX_MU_LSR & 0x01));
    return (char)(*AUX_MU_IO);
}
```

It waits for buffer to have a data and return them.

5.4 Working with memory

Following global variables initialization methods besides classic ones can be useful for some applications.

Global variable

In assembly language. It aligns specific length of variable in program data. If it is marked as global *myVar* variable will be accessible.

Listing 5.9: Assignment of global variable

```
.globl myVar;
myVar : .4byte 0;
```

Sample code Memory Allocate function

Since bare metal applications haven't got classic *malloc* function. It can be implemented marking area of memory in linker file. For this example is defined *_end* mark as *MEMORY_START* after *bss* section in linker file and ends with *periph_base* as MEMORY_END.

Listing 5.10: Memory Allocate function

```
unsigned long memory_pointer = MEM_START;
void *malloc(unsigned int size)
{
  if (size < 1 || memory_pointer + size > MEM_END)
    return (void*)0;
  memory_pointer += size;
  return (void*)(memory_pointer - size);
}
```

5.5 Interrupts

There are three types of interrupts. The GPU peripheral interrupts, CPU ARM control peripheral interrupts and special events interrupts. For each interrupt there is interrupt enable bit (Read/write) and interrupt pending bit (Read only). Interrupts generated by ARM control block are level sensitive, which means that they are remain enabled until enable bit is cleared or they are disabled. [Inc12l]

I will use ARM timer register structure, that will call interrupt. To get GPU clock I'm using mailboxes. **Physical base address:** periph_base + 0xB000

Address offset	Description
ARM Interrupt register part	
0x200	IRQ basic pending
0x204	IRQ pending 1
0x208	IRQ pending 2
0x20C	FIQ control
0x210	Enable IRQs 1
0x214	Enable IRQs 2
0x218	Enable Basic IRQs
0x21C	Disable IRQs 1
0x220	Disable IRQs 2
0x224	Disable Basic IRQs
ARM Timer register part	
0x400	Load
0x404	Value (Read Only)
0x408	Control
0x40C	IRQ Clear/Ack (Write only)
0x410	RAW IRQ (Read Only)
0x414	Masked IRQ (Read Only)
0x418	Reload
0x41C	Pre-divider
0x420	Free running counter

Table 5.6: ARM Interrupt and ARM Timer register table. [Inc12f] [Inc12g]

Sample code Timer Interrupt setup

Listing 5.11: Timer Interrupt setup

```
void irq_init (uint32_t us,
                                TimerIrqHandler function)
{
        uint32_t divisor;
        uint32_t Buffer[5] = { 0 };
        ARMTimer->Control.TimerEnable = false;

        mbox_prop_msg(&Buffer[0], 5,
                MAILBOX_TAG_GET_CLOCK_RATE,
                8, 8, 4, Buffer[4]);
        Buffer[4] /= 250;
        divisor = ((uint64_t)us*Buffer[4])/1000000;
        setTimerIrqAddress(function);
        Irq->EnableBasicIRQs.Enable_Timer_IRQ = true;
        ARMTimer->Load = divisor;
        ARMTimer->Control.Counter32Bit = true;
        ARMTimer->Control.Prescale = Clkdiv1;
        ARMTimer->Control.TimerIrqEnable = true;
        ARMTimer->Control.TimerEnable = true;
        return;
}
```

Function has two arguments. First means waiting period in microseconds and second is address of function that will be launched. Function first stops timer for safety reasons. It calls mailbox to get GPU clock. Calculate divisor for setting period. Then fill ARMTimer with values of divisor, 32-bit mode turned on, pre-scale divider set to 1 and enabling the timer with interrupt. For setting called interrupt address is used special function.

Listing 5.12: Set timer interrupt address function

```
setTimerIrqAddress:
        msr daifset,#2
        ldr x1, =TimerIrqAddr
        str x0, [x1]
        ret
```

When this function is launched address of function is saved in register *x0* as default. First the function disable all interrupts and save address on place where interrupt handler can call it. Interrupt handler is created with Vector Table saved as macro in assembly file. It can look like this:

Listing 5.13: Vector Table

```
.balign 0x800
.globl VectorTable
VectorTable:
```

```
vector  _start
vector  hang
vector  hang
vector  hang

vector  hang          // synchronous
vector  irq_handler     // irq
vector  hang          // fast interrupt
vector  hang            // SErrorStub

vector  hang
vector  hang
vector  hang
vector  hang

vector  hang
vector  hang
vector  hang
vector  hang
```

In this table each of four units corresponding to differnt types of interrupt. For this application it is second unit second entry. It will call *irq_handler* function which can launch wanted custom function.

5.6 Direct memory access (DMA)

DMA controller is directly connected to peripherals. DMA controller must be setup to use physical addresses of the peripherals. BCM2837 provides 16 independent DMA channels. [Inc12m]

DMA is using control blocks (cb) data structure. In this control block is defined the DMA transfer. It contains source adrress, destination address, length of transfer, stride, address of next control block of trasfer and information about transfer. There is possible to specify for example wait between transfers. Stride is used when sending more blocks and it defines spaces between them.

32-bit Word Offset	Description	Associated Read-Only Register
0	Transfer Information	TI
1	Source Address	SOURCE_AD
2	Destination Address	DEST_AD
3	Transfer Length	TXFR_LEN
4	2D Mode Stride	STRIDE
5	Next Control Block Address	NEXTCONBK
6-7	Reserved – set to zero.	N/A

Table 5.7: Control block data structure table. [Inc12h]

This needs to be in uncached memory.

*** Physical address = periph_base + Address offset + N * 0x100**
NOTE: N refers to number of DMA

Address Offset *	Register Name	Description
0x0	N_CS DMA	Channel N Control and Status
0x4	N_CONBLK_AD	DMA Channel N Control Block Address
0x8	N_TI DMA	Channel N CB Word 0 (Transfer Information)
0xC	N_SOURCE_AD	DMA Channel N CB Word 1 (Source Address)
0x10	N_DEST_AD	DMA Channel N CB Word 2 (Destination Address)
0x14	N_TXFR_LEN	DMA Channel N CB Word 3 (Transfer Length)
0x18	N_STRIDE	DMA Channel N CB Word 4 (2D Stride)
0x1C	N_NEXTCONBK	DMA Channel N CB Word 5 (Next CB Address)
0x20	N_DEBUG	DMA Channel N Debug

Table 5.8: DMA address map table. [Inc12i]

Sample code DMA Start

Listing 5.14: DMA start

```
void dma_start(void * src_addr, void * dest_addr,
uint32_t transfer_info, uint32_t transfer_len,
                           uint32_t DMA_CHANNEL) {
  // Prepare DMA control block.
  struct dma_cb * cb =
    (struct dma_cb*)malloc(sizeof(struct dma_cb));

  cb->info    = transfer_info;
```

```
  cb->src    = (uint32_t*)src_addr;
  cb->dst    = (uint32_t*)dest_addr;
  cb->length = transfer_len;
  cb->stride = 0;
  cb->next   = (uint32_t*)cb; // Loop itself

  struct dma_channel* channel =
    (struct dma_channel*)0x3F00700 + DMA_CHANNEL * 0x100;

  channel->cs |= DMA_CS_END;
  channel->cblock = (uint32_t*)cb;
  channel->cs = DMA_CS_PRIORITY |
    DMA_CS_PANIC_PRIORITY;
  channel->cs |= DMA_CS_ACTIVE;
}
```

Function creates DMA control block. Sets source and destination address. Sets transfer length. In this example the same DMA block loops itself, which means repeating same transfer until it is disabled. In control and status register is saved priority of channel and panic signal. Which means what to do when sending data and get outside sending zone. It also sets DMA active.

Listing 5.15: Shutdown DMA channel

```
dma_stop(uint32_t DMA_CHANNEL){
  struct dma_channel* channel =
    (struct dma_channel*)0x3F00700 + DMA_CHANNEL * 0x100;

  channel->cs |= DMA_CS_ABORT;
  msec_wait(100);
  channel->cs &= ~DMA_CS_ACTIVE;
  channel->cs |= DMA_CS_RESET;
}
```

5.7 Pulse width modulation (PWM)

Outputs bit stream with fixed frequency. It can be configured to output PWM stream or serialized version of 32-bit words. In this serialized mode it is configured to load data from FIFO storage block. This block can store up to eight 32-bit words. Modes are clocked by *clk_pwm*. Default clock is 100MHz. [Inc12n]

Physical base address: periph_base + 0x20C000

Address Offset	Register Name	Description	Size
0x0	CTL	PWM Control	32
0x4	STA	PWM Status	32
0x8	DMAC	PWM DMA Configuration	32
0x10	RNG1	PWM Channel 1 Range	32
0x14	DAT1	PWM Channel 1 Data	32
0x18	FIF1	PWM FIFO Input	32
0x20	RNG2	PWM Channel 2 Range	32
0x24	DAT2	PWM Channel 2 Data	32

Table 5.9: PWM address map table. [Inc12j]

Sample code PWM start

Listing 5.16: Enable pwm channel in assembly

```
PWM_START:
      mov w0,(periph_base + PWM_base) and $0000FFFF
      mov w1,(periph_base + PWM_base) and $FFFF0000
      orr w0,w0,w1
      mov w1,$RANGE_VAL
      str w1,[x0,RNG1]

      mov w1,PWM_USEF1 + PWM_PWEN1 + PWM_CLRF1
      str w1,[x0,CTL]
```

Function that starts PWM channel in assembly. Configuration about range is saved on *RNG1* and *RNG2*. Another configuration can be saved in *CTL* register. In given example are *PWM_USEF1* to use queue with FIFO (first in, first out). *PWM_PWEN1* to enable channel and *PWM_CLRF1* to clear FIFO.

5.8 Mailboxes

Mailbox interface. Used to communicate with GPU.

It has different channels. Channels have different formatting. Most useful mailbox channel is property channel with number 8. Another can be framebuffer with channel 1 used for screen view.

To use mailbox, we fill the mailbox array and then send it to GPU. Mailbox videocore register starts at (periph_base + 0xB880).

Register name	Address
MBOX_READ	0x0
MBOX_POLL	0x10
MBOX_SENDER	0x14
MBOX_STATUS	0x18
MBOX_CONFIG	0x1C
MBOX_WRITE	0x20

Table 5.10: Videocore register.

Sample codes

Function for sending and receiving message via mailbox.

Listing 5.17: Send message via Mailbox

```
unsigned int r =
(((unsigned int)((unsigned long)&mbox)&~0xF) | (ch&0xF));
do{asm volatile("nop");}while(*MBOX_STATUS & MBOX_FULL);
*MBOX_WRITE = r;
```

Example will wait until can write to the mailbox and write the message too channel identifier.

Listing 5.18: Receive a response

```
unsigned int r =
(((unsigned int)((unsigned long)&mbox)&~0xF) | (ch&0xF));

while(1){
do{asm volatile("nop");}while(*MBOX_STATUS & MBOX_EMPTY);
if(r == *MBOX_READ)
    // got mailbox data
}
```

Example loops until gets received a message and check if it is successful response.

In this examples:
MBOX_RESPONSE has value 0x80000000

MBOX_FULL is 0x80000000 and
MBOX_EMPTY is 0x40000000

If it is wanted for example to get serial number via mailbox. Mailbox would be filled like this:

Number of 32-bit register	Description	Value
0	Length of the message	36
1	Type of message (request)	0
2	Type of command (get serial number)	0x10004
3	Buffer size	8
4		8
5	Clear output buffer	0
6		0

Table 5.11: Sample filling mailbox array.

5.9 Memory management unit (MMU)

For using advanced features like caches it is needed to turn on MMU. This unit is already integrated in device. It just needs to be configured and launched.

Variable	Value	Description
PAGESIZE	4096	
PT_PAGE	0b11	granularity - 4k granule
PT_BLOCK	0b01	2M granule
	Accessibility	
PT_KERNEL	(0«6)	privileged, supervisor EL1 access only
PT_USER	(1«6)	unprivileged, EL0 access allowed
PT_RW	(0«7)	read-write
PT_RO	(1«7)	read-only
PT_AF	(1«10)	accessed flag
PT_NX	(1UL«54)	no execute
	Shareability	
PT_OSH	(2«8)	outter shareable
PT_ISH	(3«8)	inner shareable
	Defined in MAIR register	
PT_MEM	(0«2)	normal memory
PT_DEV	(1«2)	device MMIO
PT_NC	(2«2)	non-cachable
TTBR_CNP	1	

Table 5.12: MMU Specifications.

This example creates MMU translation tables.

Listing 5.19: Initialize memory management unit

```
unsigned long data_page = (unsigned long)&_data/PAGESIZE;
unsigned long r, b, *paging=(unsigned long*)&_end;

// setup L1 cache
paging[0]=(unsigned long)((unsigned char*)&_end+2*PAGESIZE) |
PT_PAGE | PT_AF | PT_USER | PT_ISH | PT_MEM;

// setup L2 cache, first 2M block
paging[2*512]=
(unsigned long)((unsigned char*)&_end+3*PAGESIZE) |
PT_PAGE | PT_AF | PT_USER | PT_ISH | PT_MEM;

// setup L2 cache, 2M blocks
b=periph_base>>21;
for(r=1;r<512;r++)
 paging[2*512+r]=(unsigned long)((r<<21)) |
 PT_BLOCK | PT_AF | PT_NX | PT_USER |
 (r>=b? PT_OSH|PT_DEV : PT_ISH|PT_MEM);

// setup L3 cache
```

```
for(r=0;r<512;r++)
 paging[3*512+r]=(unsigned long)(r*PAGESIZE) |
 PT_PAGE | PT_AF | PT_USER | PT_ISH |
 ((r<0x80||r>data_page)? PT_RW|PT_NX : PT_RO);

// kernel L1 cache
paging[512+511]=(unsigned long)((unsigned char*)&_end+4*PAGESIZE) |
 PT_PAGE | PT_AF | PT_KERNEL | PT_ISH | PT_MEM;

// kernel L2 cache
paging[4*512+511]=(unsigned long)((unsigned char*)&_end+5*PAGESIZE) |
 PT_PAGE | PT_AF |  PT_KERNEL | PT_ISH | PT_MEM;

// kernel L3 cache
paging[5*512]=(unsigned long)(MMIO_BASE+0x00201000) |
 PT_PAGE | PT_AF | PT_NX | PT_KERNEL | PT_OSH | PT_DEV;

// Memory Attributes array
r=  (0xFF << 0) | (0x04 << 8) | (0x44 <<16);
asm volatile ("msr␣mair_el1,␣%0" : : "r" (r));

// Mapping characteristics
r=  (0b00LL << 37) | (b << 32) | (0b10LL << 30) |
(0b11LL << 28) | (0b01LL << 26) | (0b01LL << 24) |
(0b0LL  << 23) | (25LL   << 16) | (0b00LL << 14) |
(0b11LL << 12) | (0b01LL << 10) | (0b01LL << 8) |
(0b0LL  << 7) | (25LL   << 0);
asm volatile ("msr␣tcr_el1,␣%0;␣isb" : : "r" (r));

// Save addresses of tables
asm volatile ("msr␣ttbr0_el1,␣%0" : : "r"
    ((unsigned long)&_end + TTBR_CNP));
asm volatile ("msr␣ttbr1_el1,␣%0" : : "r"
    ((unsigned long)&_end + TTBR_CNP + PAGESIZE));

// Enable page translation
asm volatile ("dsb␣ish;␣isb;␣mrs␣%0,␣sctlr_el1" : "=r" (r));
r|=0xC00800;
r&=~((1<<25) | (1<<24) | (1<<19) | (1<<12) | (1<<4) |
(1<<3) | (1<<2) | (1<<1));
r|=  (1<<0);
asm volatile ("msr␣sctlr_el1,␣%0;␣isb" : : "r" (r));
```

5.10 Serial Peripheral Interface (SPI)

Serial Peripheral Interface is serial synchronous communication. It is interface bus used to send data between microcontrollers. It can be also sensors, and SD cards. It has clock signal, so data are synchronous. Receiver device can be very simple against UART for example. [Gru]

Devices in SPI interface are divided between master and slave. Master provides the clock signal and slave only listens the clock. There are two wires for data transfer. In MOSI wire master is sending data and slave listens. In MISO wire master listens and slave is sending data. Last wires in interface are slave select wires. They are set to true all time. When they are set to false. Slave wakes up and do some action. Wires defined on Raspberry Pi are:

Pin	GPIO x=1	x=0	Description
SPIx_CE2_N	16	-	Slave select 2
SPIx_CE1_N	17	35	Slave select 1
SPIx_CE0_N	18	36	Slave select 0
SPIx_MISO	19	37	Master input, slave output
SPIx_MOSI	20	38	Master output, slave input
SPIx_SCLK	21	39	Clock signal

Table 5.13: SPI wires description table.[Inc12c]

SPI register map.

Address Offset	Register Name	Description
0x0	CS	SPI Master Control and Status
0x4	FIFO	SPI Master TX and RX FIFOs
0x8	CLK	SPI Master Clock Divider
0xC	DLEN	SPI Master Data Length
0x10	LTOH	SPI LOSSI mode TOH
0x14	DC	SPI DMA DREQ Controls

Table 5.14: SPI Address map table. [Inc12k]

5.11 Multicore applications

Raspberry Pi 3 has four processors. For use of more than one of them in terms of most possible independence, some changes needs to be done.

In linker file needs to be defined different sections for stack memory. This addresses have to be initialized. While core zero will execute main thread. Other cores needs to be set to listen for calling.

Another thing is to create global variables containing important data. For example which cores are ready. If core is not ready, it cannot execute function. Function is called with core number in register x0 and function address in register x1.

Listing 5.20: Core Execute

```
.globl core_launch
core_launch:
        ldr x3, =cores_ready
        ldr w2, [x3]
        cmp w0, w2
        bcs     CoreExecuteFail
        mov x6, #0
        mov w6, w0
        mov x5, #address_cpu0
        str x1, [x5, x6, lsl #3]
        dsb sy
        sev
        mov x0, #1
        ret
CoreExecuteFail:
        mov x0, #0
        ret
```

Every core except zero will be asleep and listens on address if it is been set. They created 2 bit mask of core Id. Loaded address *address_cpu0*, from which will be callen. It must be zeroed. To get processor to sleep, use instruction *wfe*. When address is set, it will wake and make a function call on that address.

5.11.1 Advanced applications

When goal is to process data in real time, I can use for example two cores. One will be constantly read data from periphery and save them to some public register. Another will always read the register and send it to desired output periphery.

Chapter 6

Screen output

In this chapter, I will focus on using HDMI output. First thing is change resolution and other properties with mailbox. Then send the values to GPU. We can use property channel or frame buffer channel. This will set screen to resolution 1024x768 with RGB and other properties.

To display something on the screen, we fill the specified pointer with wanted data.

Length of mailbox is 140 Bytes. Message is type request. On Address 20 is frame width, address 24 is frame height. On addresses 40 and 44 are virtual width and height. On address 48 is virtual offset. On address 60 and 64 is x and y offset. On address 68 is setting depth. On address 84 is setting pixel order. On address is color type, 1 for RGB. On address 112 is Framebuffer pointer.

Addr	Values
0	35*4
4	MBOX_REQUEST
8	0x48003
12	8
16	8
20	1024
24	768
28	0x48004
32	8
36	8
40	1024
44	768
48	0x48009
52	8
56	8
60	0
64	0
68	0x48005
72	4
76	4
80	32
84	0x48006
88	4
92	4
96	1
100	0x40001
104	8
108	8
112	4096
116	0
120	0x40008
124	4
128	4
132	0
136	MBOX_TAG_LAST

Table 6.1: Example of filling mailbox array to setup screen.

Sample code print picture

Listing 6.1: Print picture

```
void lfb_showpicture()
{
```

```
    int x,y;
    unsigned char *ptr=lfb;
    char *data=header_data, pixel[4];

    ptr += (frameHeight-height)/2*pitch + (frameWidth-width)*2;
    for(y=0;y<height;y++) {
        for(x=0;x<width;x++) {
            HEADER_PIXEL(data, pixel);
            *((unsigned int*)ptr)=*((unsigned int *)&pixel);
            ptr+=4;
        }
        ptr+=pitch-width*4;
    }
}
```

Part III

Experimental part

Chapter 7

Speed of GPIOs

7.1 Assignment

Find out the maximum safe speed of GPIOs. First I will test the maximum writing speed on GPIO pins.

Manufacturer is providing only maximum possible frequency of output GPIOs, that is not corresponding to its real value.

7.1.1 System properties

There are two parts, that can determine the speed of writing GPIO pin. First is time which processor is writing value to certain register. Second is periphery that is responsible to read the register number and accomplish the GPIO value change. This is done by flip-flop circuit.

By the datasheet of Broadcom **BCM2837** has the maximum GPIO pins frequency, which is $\approx 125MHz$ at 1.2V but is reduced if the pins are heavily loaded or have a capacitive load.

I set the frequency to **1.4GHz**, which was defaultly set to 700Mhz in config file. I will use in this example only one of four cores. I will not use graphic adapter in this example. Later I will use caches L1 & L2 & L3, because access time of RAM in Raspberry Pi slowing process as I will show in examples below.

7.1.2 Measure conditions

Device: Raspberry Pi 3 B+

- **CPU frequency:** 1.4GHz (ARM Cortex-A53)
- **GPIO Maximum clock:** 125MHz

Measuring osciloscope: Tektronix TDS1001B

- **Osciloscope frequency:** 40MHz

7.1.3 Measuring program

Switching between values HIGH (3.16V) and LOW (0V) which are values of transistor-transistor logic. I will rate quality of signal.

Program uses bare metal C language with possible inner blocks in assembly language. If blocks written in C cannot be written better in assembler, I'm using C language.

I'm using volatile pointers, so compiler wont optimize these variables. I'm compiling with $-O3$, $-O4$ or $-Ofast$ option for highest possible speed.

7.2 Measured tests

1. Measuring speed within predefined **function**
2. Measuring speed **without function**, only assign value to given register

3. Measuring speed with code from official Raspberry Pi webside (**RPi GPIO**)
4. Measuring speed with enabled **cache**

Speed of GPIOs within predefined function

In first example bare cycle with function that are switching the pin:

Listing 7.1: Example 1

```
main(){
    gpio_setup(N, GPIO_OUTPUT);
    gpio = N;
    while(1){
        gpio_output(gpio,!true);
        wait_cycles(m);
    }
}

void gpio_output (unsigned int pin, bool value)
{
    volatile unsigned int* p;
        unsigned int bit = 1 << (pin % 32);
        if (value) {
            p = (unsigned int*)(SET_REGISTER_ADDR);
        } else {
        p = (unsigned int*)(CLR_REGISTER_ADDR);
        }
        *p = bit;
        return;
}

void wait_cycles(unsigned int n)
{
    if(n) while(n--) { asm volatile("nop"); }
}
```

NOTE: In every example constant N stands for number of GPIO pin, which can be pin 0-53 available. Constant M stands for waiting number of cycles between switch.

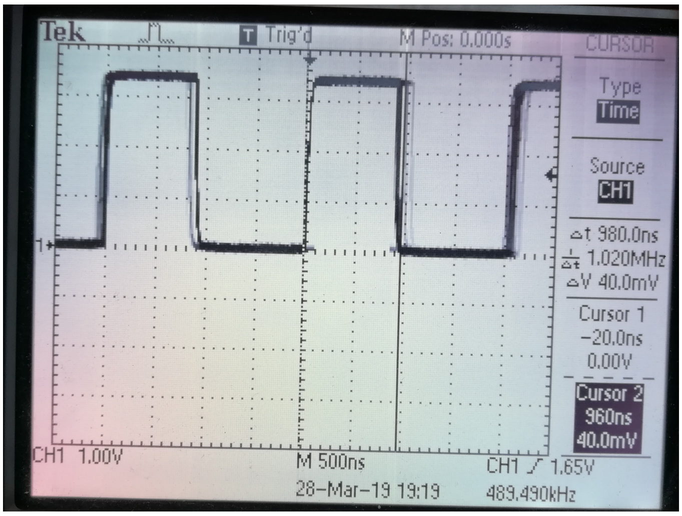

Figure 7.1: C Code (bare switching in functions).

In this first example speed is about **1MHz**.

Speed of GPIOs without predefined function

In second program I tried for even for higher speed get rid of functions, which are slowing down the process by few processor cycles.

Listing 7.2: Example 2

```
void main(){
    gpio_setup(N, GPIO_OUTPUT);

        volatile unsigned int* p_on;
    volatile unsigned int* p_off;
    unsigned volatile int gpio = N;
    unsigned volatile int bit = 1 << gpio;
    p_on = (unsigned volatile int*)(SET_REGISTER_ADDR);
    p_off = (unsigned volatile int*)(CLR_REGISTER_ADDR);

    while(1){
            *p_on = bit;
        *p_off = bit;
        }
        return;
}
```

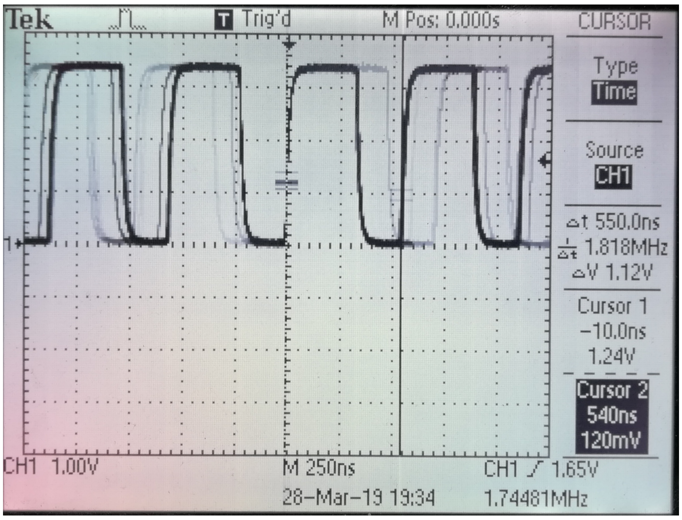

Figure 7.2: C Code (bare switching).

In second example the result is speed about **1.8MHz**.

Speed of GPIOs with code from official Raspberry Pi webside (RPi GPIO

Third example is using RPi GPIO Code Samples (link). This C code is edited to work on bare metal applications.

Listing 7.3: Example 3

```
volatile unsigned *gpio;

// GPIO setup macros
#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))
#define OUT_GPIO(g) *(gpio+((g)/10)) |=  (1<<(((g)%10)*3))

// sets   bits which are 1 ignores bits which are 0
#define GPIO_SET *(gpio+7)
// clears bits which are 1 ignores bits which are 0
#define GPIO_CLR *(gpio+10)

int main()
{
    int volatile g = N;
    int volatile gpio_value = 1<<g;
    gpio = 0x3F200000;
```

```
    INP_GPIO(g); // must use INP_GPIO before we can use OUT_GPIO
    OUT_GPIO(g);

    while(1){
        GPIO_SET = gpio_value;
        GPIO_CLR = gpio_value;
    }

    return 0;
} // main
```

In third example I'm using RPi GPIO Code Sample edited for bare metal function. The speed is exactly the same as in second example.

Enable caches

On bare metal application, we must enable caches L1 & L2 for not loading instruction and data from RAM. This can be made with assembler:

Listing 7.4: Turn on instruction and data cache in C with inline assembler commands

```
// Read System Control Register configuration data
asm ( "MRS X0, SCTLR_EL1" );
// Set [C] bit and enable data caching
asm ( "ORR X0, X0, #(1 << 2)" );
// Set [I] bit and enable instruction caching
asm ( "ORR X0, X0, #(1 << 12)" );
// Write System Control Register configuration data
asm ( "MSR SCTLR_EL1, X0" );
```

Note that memory management unit must be initialized at this moment.

After that we are getting around 30MHz.

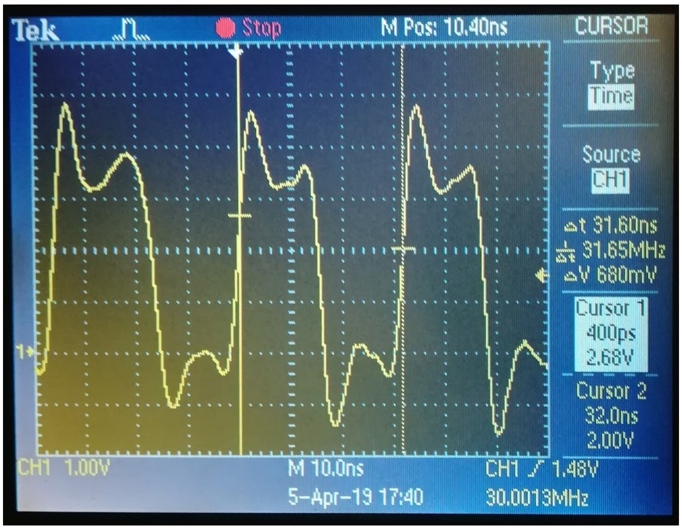

Figure 7.3: C Code (bare switching) with cache and MMU on (measured on 200MHz osciloscope).

7.3 Conclusion

All examples are made with bare metal applications.

	Speed [MHz]	Switch in functions	Cache on
(ex.1)	0.6	yes	no
(ex.2)	1.8	no	no
(ex.3) from RPi GPIO	1.8	no	no
(ex.4)	**30.3**	no	yes

Note: Frequencies are for every change of output signal per second.

Table 7.1: GPIO Speed table.

Speeds were very slow when we were not using L1&L2 caches, around 1.8MHz. When turning on cache, speed rise to 60 million changes per second (30MHz). Problem when turning on cache is that we must turn on memory management unit. Set up the paging array and we tell the CPU to use it.

Chapter 8

Speed of DA Converters

8.1 Assignment

Get the maximum speed that can produce DA Converter.

Goal is connect DA converter to Raspberry Pi and generate saw signal with highest speed. Later I can create different shape of signal like sine wave.

Secondary goal is to find out if the code is continuous, the saw signal would be flat. To know if processor is not operating something else like interrupt.

DA Converter properties

I'm using DA converter **Philips TDA8702**. It is 8 bit converter. It has clock input pin. I will call it **hold pin**. If it is turned on, it holds the output analog value. If it is turned off. The value on output is released.

My interpretation of connecting the converter to Raspberry Pi is using the shortest way between each circuits. Downside is that bit numbers on converter do not represent the same numbers of pins on Raspberry Pi.

Output voltage values of DA converter starts at **3.3V** and continue **to 5V**. So range is about 1.7V long.

Measure conditions

Device: Raspberry Pi 3 B+

- **CPU frequency:** 1.4GHz (ARM Cortex-A53)
- **GPIO Maximum clock:** 125MHz

Measuring osciloscope: Tektronix TDS1001B

- **Osciloscope frequency:** 40MHz

DA Converter: Philips TDA8702

- **Output voltage:** 3.3V - 5V
- **Converter clock max frequency:** 30MHz
- **Analog bandwidth** $\approx 3dB$ 150MHz

8.1.1 Measuring program

Program will generate saw increment value of 8 bit register. The output will be brought into 8 different GPIOs to DA Converter and analog value will be measured in oscilloscope.

First type of program will change output in function. I set the hold bit and it will hold the analog output. I set it to true at the start of change values of output pins and reset it after the change is done. This is because I would get false analog values, when it is not all set yet.

Note that I'm using the char datatype, that has 8 bit length. So I don't have to reset the value, but it will throw away carry bit and just reset the number to zero.

8.2 Measured tests

1. Measuring speed within predefined **function**
2. Measuring speed with enabled **cache**
3. Measuring speed with predefined **constant table**, with caches
4. Measuring speed with constant table, caches and using **hold bit** for fidelity
5. Measuring how quickly will converter change the **output voltage**
6. Generating **sine wave**

Measuring speed within predefined function

Listing 8.1: Example 1 - generate saw

```
// NOTE: all GPIOs are already set to output mode

char value = 0;
while(1){
    value++;
    gpio_output(16, true); // CLK - hold the value
    gpio_output(12, ( value & 0x80) ); // D0
    gpio_output(7 , ( value & 0x40) ); // D1
    gpio_output(5 , ( value & 0x20) ); // D2
    gpio_output(6 , ( value & 0x10) ); // D3
    gpio_output(20, ( value & 0x08) ); // D4
    gpio_output(21, ( value & 0x04) ); // D5
    gpio_output(13, ( value & 0x02) ); // D6
    gpio_output(19, ( value & 0x01) ); // D7
    gpio_output(16, false); // CLK - release the value
    }
```

The result of this example is measured in oscilloscope as follows:

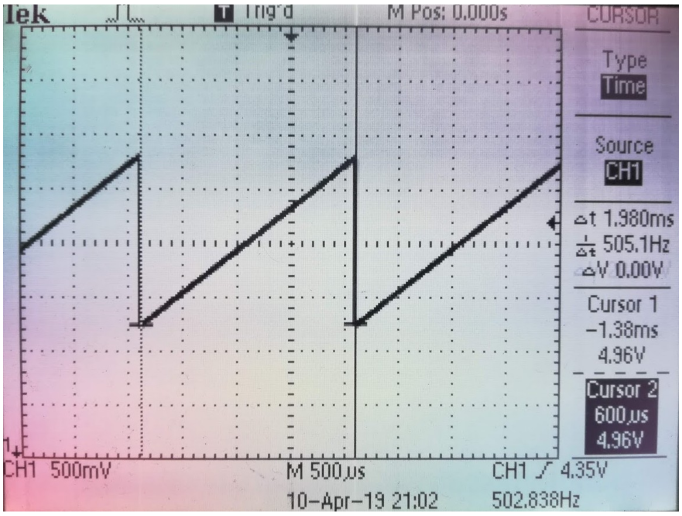

Figure 8.1: Saw generated via DA converter without cache.

As we can see frequency of saw cycles is about 500Hz. Now let's enable cache for higher speed.

Measuring speed with enabled cache

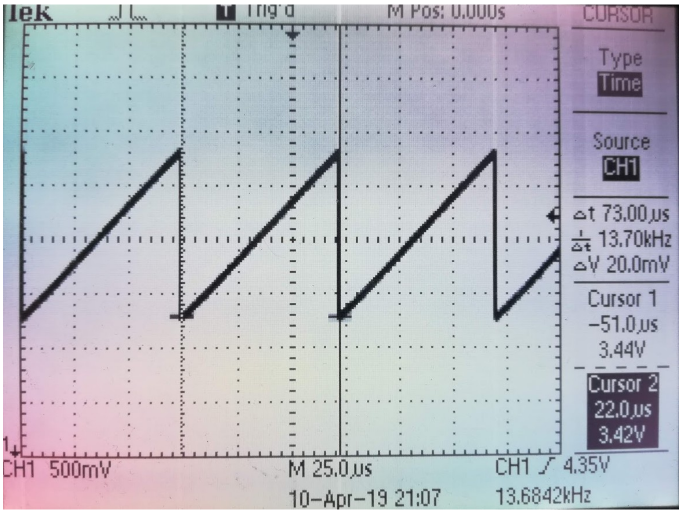

Figure 8.2: Saw generated via DA converter with cache.

Saw generated by DA converter with enabled cache has frequency about 13.7kHz.

Measuring speed with constant table

In third example I will use constant table for turning values of saw. This is because it will set all 8 output bits in one operation cycle instead of in 8 cycles. It will boost up saw speed about 8 times.

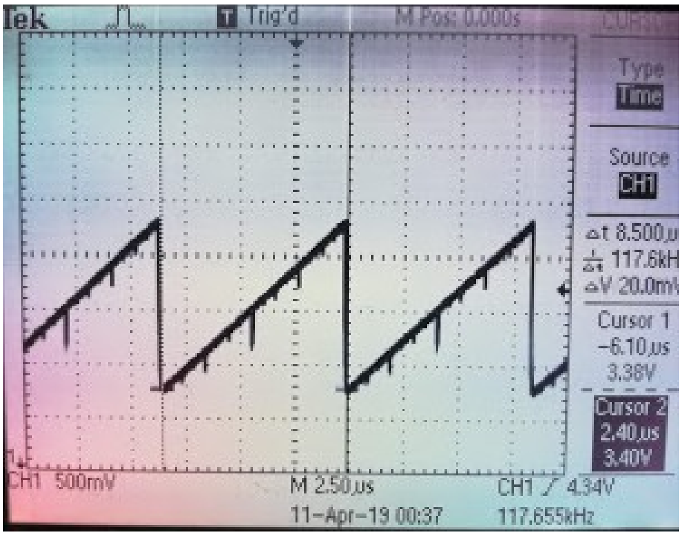

Figure 8.3: Saw generated via DA converter with constant table.

We can see error when changing to higher bits, because I didn't used hold pin due to get the highest speed. The frequency of saw is about 120kHz.

Measuring speed with managing the hold pin

When adding managing hold pin, I double instruction for writing the register memory from two instructions to four instructions. Speed is obviously divided by two. Frequency of saw is 60kHz.

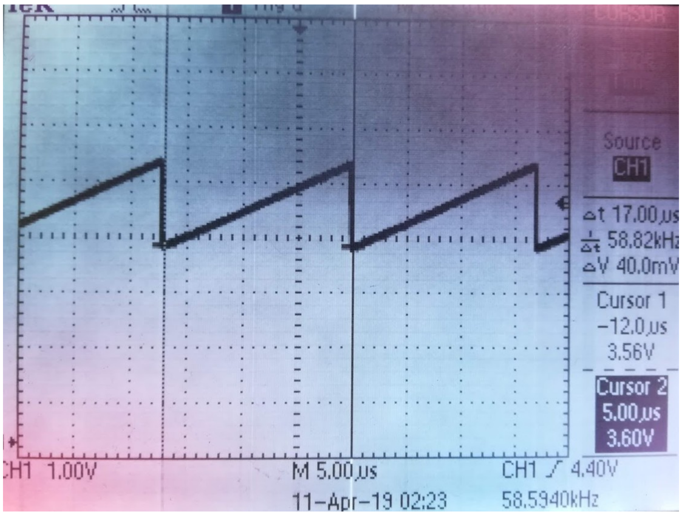

Figure 8.4: Saw generated via DA converter with managing hold pin.

Why I can't change the hold bit values in setting other values command is because hold bit must be set between setting other values.

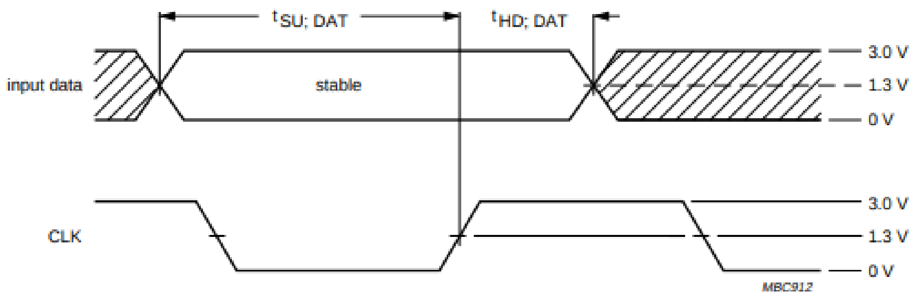

Figure 8.5: Data set-up and hold times (from Philips TDA8702 datasheet).

Measuring how quickly will converter change the output voltage

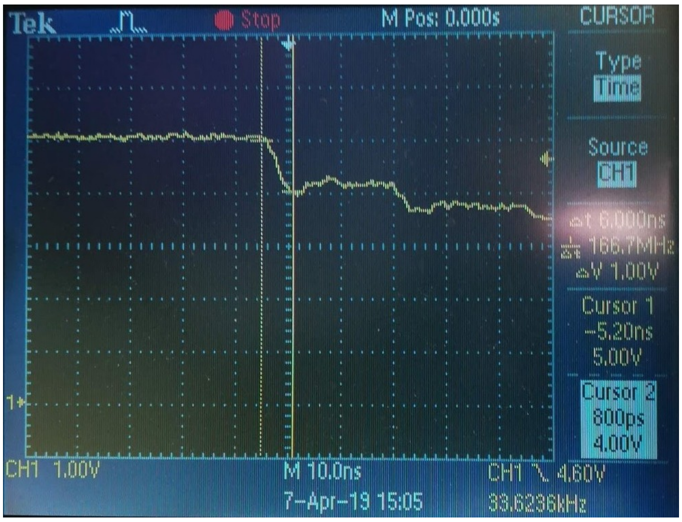

Figure 8.6: Change of output voltage (Measured on 200MHz oscilloscope).

The change of output voltage is converter capable about $1V$ every $6ns$. This means limit of speed comes from Raspberry Pi and hold pin on DA Converter. Which has frequency of 30 MHz and cannot be switched at the same time when switching input digital pins.

Sine wave

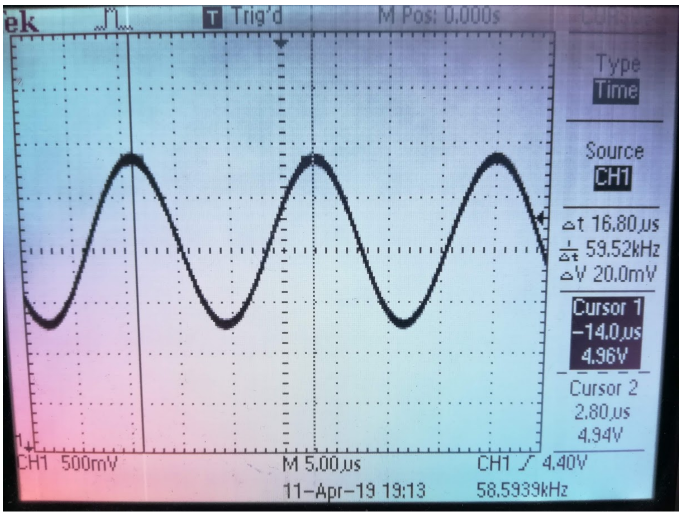

Figure 8.7: Generating sine wave with frequency 59.5kHz.

Using same set up as shown above with hold pin. Sine wave is generated in full range of output voltage capable by DA converter.

8.3 Conclusion

	Speed [kHz]	Functions	Cache	constant table	hold pin
(ex.1)	0.5	yes	no	no	yes
(ex.2)	13.7	yes	yes	no	yes
(ex.3)	**117.7**	no	yes	yes	no
(ex.4)	58.8	no	yes	yes	yes

Table 8.1: Speed of generating saw signal.

When switching output GPIOs with constant table, I can change every 8 bites with one instruction. This changing is capable of producing saw signal of 256 values (8 bit) with frequency 120kHz. However this producing error peaks, because it do not hold the value when it is not ready yet. To solve this I need to add two more instructions dividing twice the frequency to **60kHz**.

I did not register any interrupts between tests. Signal is linear when generating saw and can be used for producing signals like sine wave with fidelity. Up to 60kHz with tested 8 bit DA converter.

Appendices

Appendix A

Bibliography

[Ecl19] GNU MCU Eclipse, *How to install the arm toolchain*, `https://gnu-mcu-eclipse.github.io/toolchain/arm/install/`, May 2019, Accessed: 2019-05-20.

[Foua] Raspberry Pi Foundation, *The boot folder*, `https://www.raspberrypi.org/documentation/configuration/boot_folder.md`.

[Foub] ———, *config.txt*, `https://www.raspberrypi.org/documentation/configuration/config-txt/README.md`.

[Gri] Richard Grisenthwaite, *Armv8 technology preview*, `https://www.arm.com/files/downloads/ARMv8_Architecture.pdf`.

[Gru] Mike Grusin, *Serial peripheral interface (spi)*, `https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all`, Accessed: 2019-05-22.

[Hol] ARM Holdings, *The arm architecture*, `https://www.arm.com/files/pdf/ARM_Arch_A8.pdf`.

[Iba16] Roger Ferrer Ibanez, *Exploring aarch64 assembler*, `https://thinkingeek.com/2016/10/08/exploring-aarch64-assembler-chapter1/`, October 2016, Accessed: 2019-05-20.

[Inc12a] Broadcom Inc., *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 89.

[Inc12b] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 90.

[Inc12c] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 102.

[Inc12d] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 172.

[Inc12e] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 178.

[Inc12f] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 112.

[Inc12g] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 196.

[Inc12h] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 40.

[Inc12i] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 41.

[Inc12j] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 141.

[Inc12k] ———, *Bcm2835 arm peripherals*, `https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 152.

[Inc12l] ———, *Bcm2835 arm peripherals*, https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 109.

[Inc12m] ———, *Bcm2835 arm peripherals*, https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 38.

[Inc12n] ———, *Bcm2835 arm peripherals*, https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, February 2012, updated for version BCM2837, Accessed: 2019-03-02, p. 138.

[Joh11] Johan, *arm gcc toolchain as arm-elf or arm-none-eabi*, https://stackoverflow.com/questions/5961701/arm-gcc-toolchain-as-arm-elf-or-arm-none-eabi-what-is-the-difference, November 2011, Accessed: 2019-05-20.

[Mag] Magpi Magazine, *Raspberry pi 3: Specs, benchmarks testing*, https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/.

[Max] Bruce A. Maxwell, *A simple makefile tutorial*, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/.

[mod18] modexp, *A guide to arm64 / aarch64 assembly on linux with shellcodes and cryptography*, https://modexp.wordpress.com/2018/10/30/arm64-assembly/, October 2018, Accessed: 2019-05-20.

[Obe] Sophie Charlotte Oberschule, *Linker scripts*, http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html.

[Sho15] Chris Shore, *Armv8-a architecture overview*, https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/ARMv8_Overview.pdf, September 2015, Accessed: 2019-05-20.

[Sol16] Mark Solters, *Arm gcc toolchain in ubuntu*, http://marksolters.com/programming/2016/06/22/arm-toolchain-16-04.html, July 2016, Accessed: 2019-05-20.

[Sys] GNU Operating System, *Cross-compilation*, https://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html.

[Upt18] Eben Upton, *New product: Raspberry pi 3 model a+*, https://www.raspberrypi.org/blog/

new-product-raspberry-pi-3-model-a/, November 2018, Accessed: 2019-05-20.

[Wel] David Welch, *bootloader07*, https://github.com/dwelch67/raspberrypi/tree/master/boards/pi3/aarch64/bootloader07, Accessed: 2019-03-21.

Appendix B

List of Abbreviations

Shortcut	Meaning
IDE	Integrated development environment
BIOS	Basic Input-Output System
USB	Universal Serial Bus
UART	Universal asynchronous receiver-transmitter
PC	Personal computer
SPI	Serial Peripheral Interface
DMA	Direct Memory Access
pc	program counter
sp	stack pointer

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vanc** Jméno: **Petr** Osobní číslo: **465926**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Platforma Raspberry Pi bez použití operačního systému

Název bakalářské práce anglicky:

Raspberry Pi Platform without an Operating System

Pokyny pro vypracování:

Seznamte se s možnostmi vývoje programového vybavení a běhu programů na platformě Raspberry Pi bez použití operačního systému. Uvažujte deterministické procesy. Dostupné možnosti porovnejte. Vyberte vhodný způsob a realizujte ukázkové programy, zejména pro vstup a výstup dat v reálném čase v definovaných časových okamžicích a pro využití videovýstupu.

Seznam doporučené literatury:

[1] https://www.raspberrypi.org/forums/viewtopic.php?t=35207
[2] http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/
[3] https://www.raspberrypi.org/forums/viewforum.php?f=72
[4]https://archive.fosdem.org/2017/schedule/event/programming_rpi3/attachments/slides/1475/export/events/attachments/programming_rpi3/slides/1475/bare_metal_rpi3.pdf
[5] https://en.wikibooks.org/wiki/Bare-metal_Raspberry_Pi_Programming

Jméno a pracoviště vedoucí(ho) bakalářské práce:

prof. Ing. Pavel Zahradník, CSc., katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce:
do konce letního semestru 2019/2020

prof. Ing. Pavel Zahradník, CSc.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta