KU LEUVEN

FACULTEIT INDUSTRIELE INGENIEURSWETENSCHAPPEN

LUMEN programming language and game console

Lab Programmeertechnieken

Andries Gert-Jan
Dejager Xavier
Steen Nick

Master of Science in de industriële ingenieurswetenschappen:
Elektronica-ICT, optie Elektronica

Docent:
J.J. Vandenbussche

Academiejaar 2014 – 2015

Preface

LUMEN stands for Language For Ultraportable And Microcontroller-based Embedded eNvironments. It is an ultra lightweight programming language intended to design arcade games on embedded devices.
The core features of LUMEN are:

- Easy to understand
- Small memory footprint on any platform
- minimal number of keywords

Andries Gert-Jan
Dejager Xavier
Steen Nick

Contents

Abstract

The idea of the project is to build a basic game console. This console interprets the LUMEN programming language which is also part of the project. The project is divided in two separate parts.

The first part is developing an interpreted programming language, with an accompagnying IDE and compiler with sprite editor and map designer. The tools will be considered as already existing, since these do not contribute to embedded programming. The entire flow of the program, using inputs and providing output, is written in the interpreted code. The preprogrammed code in the PIC is merely a framework to enable the interpreter to execute the full range of functions available to the programmer.

In the second part, a game console is constructed. This console is built upon a PIC18F4620 microcontroller which runs an interpreter. A monochrome display is provided to give the game graphical output. Inputs are possible by two separate handheld controllers and thus multiplayer with up to 2 players is supported. The Games are stored in replaceable game cartridges to give a maximum flexibility.
The goal of the project is to make a new programming language and gaming console with the following features:

- An interpreter running on the PIC18F4620
- Graphical output to a monochrome display
- Input from two handheld controllers
- Multiplayer support with a maximum of two controllers
- Replaceable ROM game cards (cartridge system)

List of Abbreviations and Symbols

Abbreviations

CPU	Central Processing Unit
etc.	et cetera
FRS	Functional Requirements Specifications
IO	Input Output
kB	Kilo Byte
LCD	Liquid Crystal Display
LIFO	Last In First Out
LUMEN	Language For Ultraportable And Microcontroller-based Embedded eNvironments
MCU	Micro Controller Unit
MHz	Mega Hertz
NES	Nintendo Entertainment System
PIC	Peripheral Interface Controller
RAM	Random Access Memory
ROM	Read Only Memory
SPI	Serial Peripheral Interface
VRAM	Video Random Access Memory
TRAM	Text Random Access Memory

Chapter 1

The game console

1.1 Overview

The console has some specific features that will be described here. All of the aspects of hardware requirements, memory usage, input and output... will be explained. Based on the image below, all requirements and specifications will be defined, from the controller to the interpreter to the graphics control. Any of these blocks is a stand-alone configuration, but the interpreter and the graphics control compose the actual console.

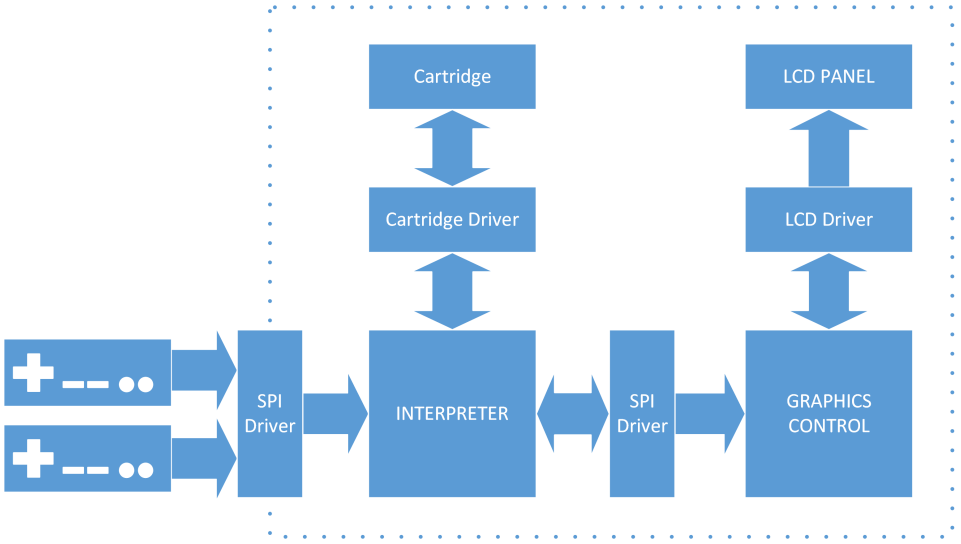
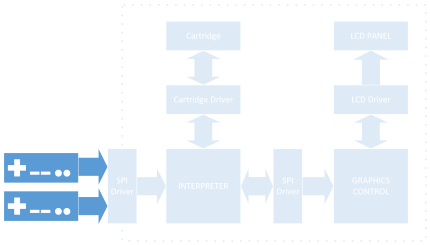


FIGURE 1.1: Console Overview

1.2 Controller

1.2.1 Overview

A total of 8 push buttons are present on the controller, each with a pressed and a released state. When the user presses a button the state changes form released to pressed and vice versa. The controller uses an SPI interface to allow communication with the game. Multiplayer is possible when connecting 2 controllers at once.



1.2.2 Buttons

D-Pad

Four of the buttons at the left side of the controller are formed as a D-Pad, a flat thumb-operated four-way directional control with one push button on each point, providing intuitive direction and steering capabilities, and should be used as such.

Select and start

In the center, the select and start button are located. These should be used to start the game, pause it, allow menu navigation etc. The start button should allow pausing the game at any point, pressing start again returns to the game.

A and B

Next to the select and start buttons, A and B are located. These two buttons are to be used in-game, providing interaction with objects on the screen. They should not be used to pause or unpause the game at any time.

1.2.3 Interface

The controller connects through a three-wire interface similar to SPI. A maximum of two controllers can be connected, addressed as 0 and 1. The timing can be seen in this view (clockperiod of 6μs, from top to bottom: Latch, Clock, Data):

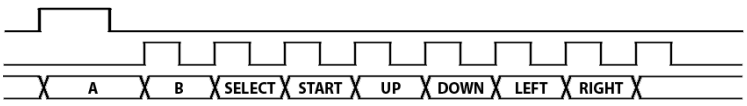


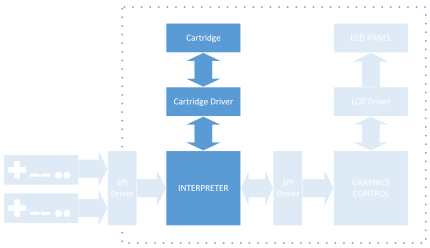
FIGURE 1.2: Interface timing of the controller

1.3 Interpreter

1.3.1 Overview

The interpreter is the main unit of the console. It executes the code and sends commands to the graphics controller to ensure the data displayed on screen is correct. Since the code is stored in external cartridge memory in the form of bytecode, specific requirement should be met to obtain accurate behaviour:

- MCU: PIC18F4620 (8-bit, 8 MHz)
- RAM: 3968 bytes - 256 bytes can be used by one game
- Input: NES controller (max 2)



1.3.2 Peripherals

Registers

Data transfer and manipulation is done via registers. These are memory locations that can be used to perform arithmetics on, used as delay counter... They can be used as data storage, but the RAM is a better place for this. In total, 16 8-bit wide registers can be used, yet one should be used with caution since it will be overwritten by some specific instructions. Registers 0 to 14 are free to use, and will only be overwritten when the code tells the interpreter to do so. The 15th register, called the HL register, is used by the interpreter as data buffer for e.g. reading a controller or timer.

Stack

The stack is used to temporarly store data when the interpreter executes a subroutine or a register should be stored for later use. It is a LIFO structure of 256 items long, each item being 8 bit wide. The stack pointer is 8 bit too, but cannot be manipulated by the user directly: push and pop instructions are supported, and are available through the interpreter.

Program counter

The program counter is a pointer to where the interpreter is in the code. It is a 16 bit unsigned short, starting at 0 for the first instruction. Each instruction or operand

increments this program counter. The program counter cannot be manipulated by the user directly, but should go through the jump, call or return instructions of the interpreter. A jump instruction replaces the program counter by a new value, while the call instruction pushes the current program counter to the stack and then replaces the program counter by the new one. Return pops the program counter from the stack at the end of a subroutine, and replaces the current program counter by this value, which results in a return to the point of calling the subroutine.

RAM

RAM can be used to store data during the execution of a program, it is 256 items long, each item being 8 bits wide. It will never be overwritten by the interpreter automatically, yet overwriting can happen when the code tells the interpreter to do so. It is possible to exchange data from a register to a RAM address and the other way around. Data that should be stored over a longer period of time can be placed in RAM, and copied back to a register for later manipulation.

1.3.3 Cartridge memory mapping

An important aspect of every game is the memory mapping on the cartridge. Every item (code, sprites, strings and maps) has its own specific start- and endaddress to establish a solid and correct performance. The mapping is as follows:

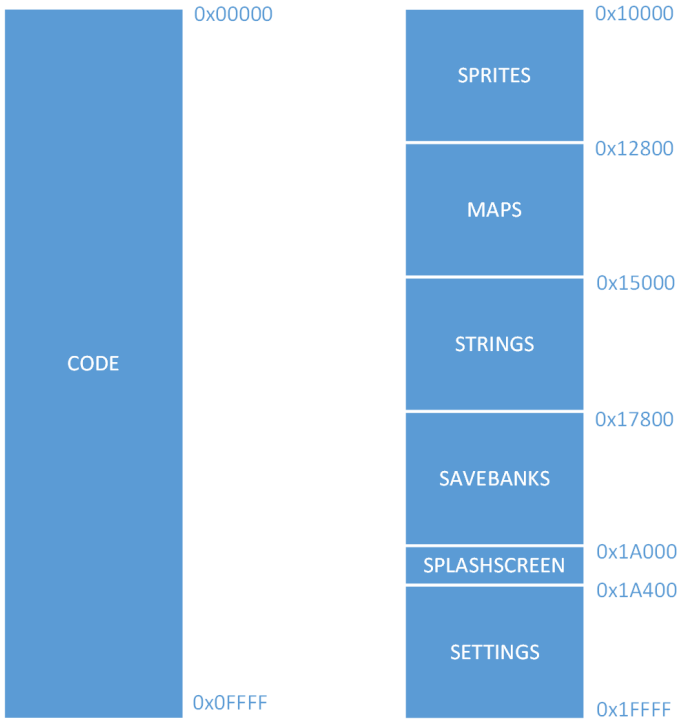


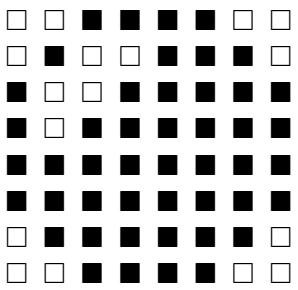
FIGURE 1.3: Memory map of the cartridge

Code

This is the actual bytecode that will be interpreted and executed. An in-depth look on the instructionset can be found in chapter 2.1. The maximum codesize is 64kB, or 65536 bytes. If no jump is performed at byte 65535, the program counter will overflow, resulting in a reset of the program counter to zero. Code examples can be found in the appendix on page 67.

Sprites

A sprite is an 8 by 8 pixels bitmap in black and white. Each bit represents one pixel, resulting in a size of 8 bytes per sprite. The addressing starts at the top left, and goes down per colum, this happens to be 8 bits, and can thus be packed in one byte. The first colum is the first byte, the second colum the second byte...



0	0	1	1	1	1	0	0
0	1	0	0	1	1	1	0
1	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	0
0	0	1	1	1	1	0	0

This sprite, representing a ball, has the binary representation in the right table. Since we address by colum, the first byte is 0b00111100, which corresponds to 0x3C. The second colum is 0b01001110, and thus 0x4E. When all pixels are calculated the bytes are placed after one another, resulting in a complete sprite, here 0x3C4E9FBFFFFF7E3C.

Maps

A map is a combination of sprites that can fill the background of the screen. Placing a map on the screen can be functional, e.g. in a platformer game, or aesthetically. It is in fact a tiled field containing the indexes of sprites in the onscreen memory, each tile being 8 by 8 pixels (equal to 1 sprite). For more information on maps, please refer to the chapter 1.4.3.

Strings

Strings can be used to be drawn on screen as guidelines or instructions to the user. They are to be stored in the cartrigde memory as null-terminated ascii encoded strings, and will be read as such. If a string is not null-terminated, the interpreter will read until the first null is encountered, which can result in unexpected behaviour. A maximun total size of 10kB of strings can be loaded into the cartridge, yet that amount can never be displayed at once on the screen, since the string buffer is limited to 256 bytes. For more information on strings, refer to the chapters 1.4.3 and 1.4.3.

Savebanks

The savebanks are 10 address locations of each 1kB that can be used freely by the game. If playernames, highscores, achievements... should be stored, this is the location to do so. The game will not check if the data is legitimate or not, this is up to the programmer. Settings should not be stored in savebanks, for this has its own address location.

Splash screen

A splash screen is the screen that should be shown while all data is loaded and transfered between the interpreter and the graphics control. It has to be a black and white bitmap addressed from the top left down each colum. Since the screen is 84x48, a fullscreen bitmap is 4032 bits, or 504 bytes.

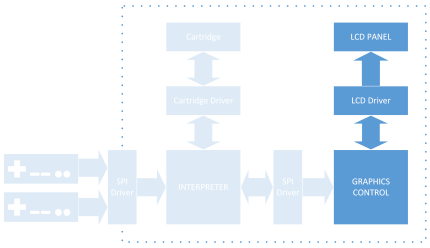
Settings

Settings can be stored in this address location. No restrictions are applied, and the content is up to the programmer.

1.4 Graphics control

1.4.1 Overview

The graphics control is located entirely in a separate microcontroller. Its tasks in a nutshell are to drive the actual LCD display and to provide a simple way of using graphics in a game. To do so it provides a set of commands to the programmer that can be used to initialize, draw, move and remove spites on the screen. It also has support for text on screen and for maps that can be scrolled over the screen.



1.4.2 The LCD display

The LCD display used in this setup has a resolution of 84x48 pixels. It has a PCD8544 chipset and is interfaced with SPI. The chipset does all the timing and scanning of the LCD. All we need to provide is a correct initialisation, fill the VRAM and send valid commands. This is where graphics control comes in. The graphics control translates SPI commands and data from the interpreter to SPI commands and data the LCD understands.

1.4.3 Memories of the graphics control

The graphics control has several sort of memories. Although most of it is directly in ram, their purposes are completely different:

Sprites in memory

This part of the memory is used to store the actual data of a sprite, or the data for the visual representation of that particular sprite. It can contain up to 32 different sprites and a sprite is 8 by 8 pixels or 8 bytes in size. For an in-depth look on the format of sprites, please refer to chapter 1.3.3. Every sprite in memory can be referenced by its number in memory, as there are 32 possible sprites valid numbers are from 1 up to 32.

Sprites on screen

This part of the memory tell the graphics control what to draw where concerning sprites. It can contain up to 32 record. A record is 3 bytes big and represents a sprite (first byte), a X position on the screen (second byte) and a Y position on the screen (third byte). From now on a sprite can be drawn on the screen. And a sprite on the screen is referenced by a list number, that number is the number of the record containing information of the sprite data and position. Note that multiple records can use the same sprite.

Maps

This part of the memory is used for background maps. Up to four maps can be used. A map exists of tiles and these tiles are in fact just sprites. One must use the the number of the spite in memory to use it in a map. A map measures 16 by 8 tiles, or 128 by 64 pixels. For tiling the map a sequence of references to sprites is needed. A map is tiled from left to right and than from top to bottom. When drawing a map on the screen, one can give a X and Y offset, resulting in a scrollable background.

Strings

This part of the memory is used to store pointers to string data. A string starts at the pointer location and ends when a null character is encountered. A maximum of 32 strings is allowed. The number of the string is used for referencing when drawing it to the screen.

String data

This part of the memory is used to store the actual data for the strings. It can contain up to 256 characters. Therefore all the strings (including null characters) together can have a maximum size of 256 bytes. A new string starts where the old string ended. Therefore it is important to initialize the strings at the beginning. Otherwise conflicts of overwriting strings is a ever present danger.

Font

This font memory contains the data for the actual graphical representation of an ascii character on screen. A character is made out of five bytes, and is mapped onto a 5x7 font. This part is also the only part not in RAM, it is hard coded in the flash region of the microcontroller.

VRAM

The VRAM is the graphical buffer for the screen. If needed, the map is drawn first, after which sprites are drawn on top. Every graphical element except for strings are drawn in this buffer, which is then sent to the display where it resides in the display's RAM until overwritten.

TRAM

The TRAM is the graphical buffer for the text to be drawn on screen. Through the font memory, characters are converted to bytes and draw in the TRAM. When the image is to be drawn to the display, the VRAM and TRAM buffers are XORed to ensure that twice a black pixel becomes white so that the image or character can still be seen.

1.4.4 The normal operation

To obtain a correct functionality, the graphics control is initialized using the provided intializer. Once this is completed, commands and data can be sent to it. This is done though a hardware SPI interface and will be regarded as a command, optionally followed by some arguments. Once a command is received the routine waits for the needed arguments, then calls the corresponding function. Correct commands are:

Load sprite: 0x10 Load a sprite in sprite memory
Load map: 0x12 Load a map in map memory
Load string: 0x14 Load a string in string memory
Set sprite: 0x16 Initialize a sprite in sprites on screen memory
Update sprite: 0x18 Draw a sprite on the screen at a certain position
Clear sprite: 0x1A Remove a sprite from the screen
Draw map: 0x1C Draw a map on the screen with a certain offset
Draw string: 0x1E Draw a string on the screen at a certain position
Clear all: 0x20 Clear both VRAM and TRAM
Clear graphics: 0x22 Clear VRAM
Clear strings: 0x24 Clear TRAM
Redraw: 0x26 Send the VRAM and TRAM to the LCD
All on: 0x28 Turn all the pixels on, draw them all black
All off: 0x2A Turn all the pixels of, draw them white
Invert: 0x2C Enable the inverted mode, black is white and vice versa
Normal: 0x2E Enable normal mode, to restore from inverted mode

Chapter 2

Instruction set

2.1 Overview

With this instructionset, the interpreter can be told to perform an action or operation. Each instruction performs a single action, but are not bound to an instruction or cycle speed. This means that it is hard to tell how long e.g. a subroutine will take, because there is no way to tell how long each instruction last. For this matter, a timer is implemented, so that can be checked how many milliseconds have passed. All instructions have the same format: a header, followed by its arguments. The header is a specific code, called opcode or operation code, and each instructions has its own. This means that, to call a function, all you need to do is write down the opcode, followed by it arguments. It looks like this: Most arguments are 8 bit wide,

Opcode Operand 1 Operand 2 ...

but some functions require a 16 bit argument, e.g. jumps. To correctly format these 16 bit arguments, the most significant byte (bit 8 to 15) should be first, followed by least significant byte (bit 0 to 7). This means that its hexadecimal representation can be ćopiedt́o the operands. To reduce size, and thus speed, instructions have an arbitrary length, and not a fixed one of e.g. 32 bits. This complicates jumps since not instructions should be counted, but bytes.

The instructions can be divided into seperate groups, depending on their function:

Variables Loading of values, register manipulations ... [page 10]
Mathematics Adding, subtracting, multiplying ... [page 15]
Logic Logic AND, OR, XOR and NOT operations. [page 23]
Jumps and calls Program counter manipulations. [page 27]
Stack Push and pop values from and to the stack. [page 30]
Graphics Draw sprites, strings ... [page 33]
Timer and delays Read and reset the timer and perform delays. [page 40]
Data read/writes Read or write from the cartridge data memory. [page 39]

2.2 Variables

Mnemonic (opcode)	NOP (0x00)
Function header	inline void ip_nop()
Summary	This function has exceptionally well performance at doing nothing
Used by	**Interpreter.c**

Mnemonic (opcode)	LD (0x01)
Function header	inline void ip_load(unsigned char address, unsigned char value)
Summary	Loads a fixed value into a specific register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	CPY (0x02)
Function header	inline void ip_copy(unsigned char destaddr, unsigned char srcaddr)
Summary	Copies te value from a register to another register. The value in the source register is retained.
Parameters	**destaddr:** The destaddr parameter requires an unsigned short containing an 8 bit address. **srcaddr:** The srcaddr parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	ST (0x03)
Function header	inline void ip_store(unsigned char destaddr, unsigned char srcaddr)
Summary	Store the value of a register to RAM.
Parameters	**destaddr:** The destaddr parameter requires an unsigned short containing an 8 bit address. **srcaddr:** The srcaddr parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	STI (0x04)
Function header	inline void ip_storeImmediate(unsigned char address, unsigned char value)
Summary	Store a fixed value to RAM.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	LDR (0x05)
Function header	inline void ip_loadRam(unsigned char destaddr, unsigned char srcaddr)
Summary	Load a value from RAM into a register.
Parameters	**destaddr:** The destaddr parameter requires an unsigned short containing an 8 bit address. **srcaddr:** The srcaddr parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	CMP (0x06)
Function header	inline void ip_compare(unsigned char firstaddress, unsigned char secondaddress)
Summary	Compare two values of registers. Two results are possible: 0 means not equal, not 0 means equal.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	GT (0x07)
Function header	inline void ip_greaterThan(unsigned char firstaddress, unsigned char secondaddress)
Summary	Check if the first register's value is greater than the second register's value. Two results are possible: 0 means register 2's value is bigger, not 0 means register 1's value is bigger.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	GTI (0x08)
Function header	inline void ip_greaterThanImmediate(unsigned char address, unsigned char value)
Summary	Check if a register value is greater than a fixed value Two results are possible: 0 means the fixed value value is bigger, not 0 means the register's value is bigger.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	LT (0x09)
Function header	inline void ip_lessThan(unsigned char firstaddress, unsigned char secondaddress)
Summary	Check if the first register's value is smaller than the second register's value. Two results are possible: 0 means register 2's value is smaller, not 0 means register 1's value is smaller.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	LTI (0x0A)
Function header	inline void ip_lessThanImmediate(unsigned char address, unsigned char value)
Summary	Check if a register value is smaller than a fixed value. Two results are possible: 0 means the fixed value is smaller, not 0 means the register's value is smaller.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	SWAP (0x0B)
Function header	inline void ip_swap(unsigned char firstaddress, unsigned char secondaddress)
Summary	Swap the values of two registers. Each register gets the value of the other register in one instruction.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	SWAPR (0x0C)
Function header	inline void ip_swapRam(unsigned char firstaddress, unsigned char secondaddress)
Summary	Swap the values of two RAM addresses. Each RAM address gets the value of the other RAM address in one instruction.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

2.3 Mathematics

Mnemonic (opcode)	ADD (0x0D)
Function header	inline void ip_add(unsigned char firstaddress, unsigned char secondaddress)
Summary	Add the values of two registers. The result is stored in the first register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	ADDI (0x0E)
Function header	inline void ip_addImmediate(unsigned char address, unsigned char value)
Summary	Add a fixed value to the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	SUB (0x0F)
Function header	inline void ip_subtract(unsigned char firstaddress, unsigned char secondaddress)
Summary	Subtract the values of two registers. The result is stored in the first register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	SUBI (0x10)
Function header	inline void ip_subtractImmediate(unsigned char address, unsigned char value)
Summary	Subtract a fixed value from the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	MUL (0x11)
Function header	inline void ip_multiply(unsigned char firstaddress, unsigned char secondaddress)
Summary	Multiply the values of two registers. The result is stored in the first register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	MULI (0x12)
Function header	inline void ip_multiplyImmediate(unsigned char address, unsigned char value)
Summary	Multiply a fixed value with the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	DIV (0x13)
Function header	inline void ip_divide(unsigned char firstaddress, unsigned char secondaddress)
Summary	Divide the value of the first register by the value of the second register. The result is stored in the first register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	DIVI (0x14)
Function header	inline void ip_divideImmediate(unsigned char address, unsigned char value)
Summary	Divide the value of a register by a fixed value. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	INC (0x15)
Function header	inline void ip_increment(unsigned char address)
Summary	Increment the value of a register by one. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	DEC (0x16)
Function header	inline void ip_decrement(unsigned char address)
Summary	Decrement the value of a register by one. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	POW (0x17)
Function header	inline void ip_power(unsigned char destaddr, unsigned char pwraddr)
Summary	Calculate the power of the value of a register. The power is the value of a register with the address in the second argument. The result is stored in the first register.
Parameters	**destaddr:** The destaddr parameter requires an unsigned short containing an 8 bit address. **pwraddr:** The pwraddr parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	POWI (0x18)
Function header	inline void ip_powerImmediate(unsigned char destaddr, unsigned char pwrval)
Summary	Calculate the power of the value of a register. The power is a fixed value given as the second argument. The result is stored in the register.
Parameters	**destaddr:** The destaddr parameter requires an unsigned short containing an 8 bit address. **pwrval:** The pwrval parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	SQRT (0x19)
Function header	inline void ip_squareRoot(unsigned char address)
Summary	Calculate the square root of the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	LOG2 (0x1A)
Function header	inline void ip_log2(unsigned char address)
Summary	Calculate the log (base 2) of the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	LOG10 (0x1B)
Function header	inline void ip_log10(unsigned char address)
Summary	Calculate the log (base 10) of the value of a register. The result is stored in the register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	RND (0x1C)
Function header	inline void ip_rand(unsigned char rangemin, unsigned char rangemax)
Summary	Generate a random value in a given range. The value is an unsigned char. The range is defined by the first and the second argument as addresses of registers.
Parameters	**rangemin:** The rangemin parameter requires an unsigned short containing an 8 bit address. **rangemax:** The rangemax parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	RNDI (0x1D)
Function header	inline void ip_randImmediate(unsigned char rangemin, unsigned char rangemax)
Summary	Generate a random value in a given range. The value is an unsigned char. The range is defined by the first and the second argument as fixed values.
Parameters	**rangemin:** The rangemin parameter requires an unsigned short containing an 8 bit value. **rangemax:** The rangemax parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

2.4 Logic

Mnemonic (opcode)	AND (0x1E)
Function header	inline void ip_bitwiseAnd(unsigned char firstaddress, unsigned char secondaddress)
Summary	Performs a bitwise AND on the values of two registers. The result is stored in the HL register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	ANDI (0x1F)
Function header	inline void ip_bitwiseAndImmediate(unsigned char address, unsigned char value)
Summary	Performs a bitwise AND on the value of a register and a fixed value. The result is stored in the HL register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	OR (0x20)
Function header	inline void ip_bitwiseOr(unsigned char firstaddress, unsigned char secondaddress)
Summary	Performs a bitwise OR on the values of two registers. The result is stored in the HL register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	ORI (0x21)
Function header	inline void ip_bitwiseOrImmediate(unsigned char address, unsigned char value)
Summary	Performs a bitwise OR on the value of a register and a fixed value. The result is stored in the HL register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	XOR (0x22)
Function header	inline void ip_bitwiseXor(unsigned char firstaddress, unsigned char secondaddress)
Summary	Performs a bitwise XOR on the values of two registers. The result is stored in the HL register.
Parameters	**firstaddress:** The firstaddress parameter requires an unsigned short containing an 8 bit address. **secondaddress:** The secondaddress parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	XORI (0x23)
Function header	inline void ip_bitwiseXorImmediate(unsigned char address, unsigned char value)
Summary	Performs a bitwise XOR on the value of a register and a fixed value. The result is stored in the HL register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	NOT (0x24)
Function header	inline void ip_bitwiseNot(unsigned char address)
Summary	Performs a bitwise NOT of the value of a registers. The result is stored in the HL register.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	SBIT (0x25)
Function header	inline void ip_setBit(unsigned char address, unsigned char position)
Summary	Sets a bit in a register's value, sepcified by the second argument.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **position:** The position parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	CBIT (0x26)
Function header	inline void ip_clearBit(unsigned char address, unsigned char position)
Summary	Clears a bit in a register's value, sepcified by the second argument.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address. **position:** The position parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

2.5 Jumps and Calls

Mnemonic (opcode)	JMP (0x27)
Function header	inline void ip_jump(unsigned short address)
Summary	Jumps inconditionally to the instruction address specified in the first argument.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	JMPR (0x28)
Function header	inline void ip_jumpRelative(unsigned short address)
Summary	Jumps inconditionally by incrementing the program counter with the value of the first argument. If the the program counter overflows, execution of the program is aborted.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	JMPZ (0x29)
Function header	inline void ip_jumpIfZero(unsigned short address)
Summary	Jumps if the HL register's value equals zero to the instruction address specified in the first argument.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	JMPRZ (0x2A)
Function header	inline void ip_jumpRelativeIfZero(unsigned short address)
Summary	Jumps if the HL register's value equals zero by incrementing the program counter with the value of the first argument. If the the program counter overflows, execution of the program is aborted.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	JMPNZ (0x2B)
Function header	inline void ip_jumpIfNotZero(unsigned short address)
Summary	Jumps if the HL register's value does not equal zero to the instruction address specified in the first argument.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	JMPRNZ (0x2C)
Function header	inline void ip_jumpRelativeIfNotZero(unsigned short address)
Summary	Jumps if the HL register's value does not equal zero by incrementing the program counter with the value of the first argument. If the the program counter overflows, execution of the program is aborted.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	CALL (0x2D)
Function header	inline void ip_call(unsigned short address)
Summary	Executes a subroutine at the address specified in the first argument. Starts by pushing the program counter to the stack. At the end of a subroutine, RET should be called to pop the program counter from the stack and return to the last execution point. If the the program counter overflows, execution of the program is aborted.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	CALLNZ (0x42)
Function header	inline void ip_callIfNotZero(unsigned short address)
Summary	Performs a call if and only if the HL register does not contain zero.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	CALLZ (0x43)
Function header	inline void ip_callIfZero(unsigned short address)
Summary	Performs a call if and only if the HL register contains zero.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	RET (0x2E)
Function header	inline void ip_ret()
Summary	Pops the program counter from the stack and returns to execution after a subroutine was called.
Used by	**Interpreter.c**

2.6 Stack

Mnemonic (opcode)	PUSH (0x2F)
Function header	inline void ip_push(unsigned char address)
Summary	Pushes a register's value to the stack.
Parameters	**address:** The address parameter requires an unsigned short containing an 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	PUSHI (0x30)
Function header	inline void ip_pushImmediate(unsigned char value)
Summary	Pushes a fixed value to the stack.
Parameters	**value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	POP (0x31)
Function header	inline void ip_pop(unsigned char address)
Summary	Pops a value from the stack to a register.
Parameters	**address:** The address parameter requires an unsigned short containing a 8 bit address.
Used by	**Interpreter.c**

2.7 Controller

Mnemonic (opcode)	BRD (0x32)
Function header	inline void ip_buttonRead(unsigned char id)
Summary	Reads a controller and places the input in the HL register. The id as the first argument is either 0 or 1, depending on the controller that is to be read.
Parameters	**id:** The id parameter requires an unsigned short containing a 8 bit id.
Used by	**Interpreter.c**

Mnemonic (opcode)	BRDS (0x33)
Function header	inline void ip_buttonReadSingle(unsigned char id, unsigned char btn)
Summary	Reads a controller and places the input of a specific button in the HL register. If the button is pressed, 1 is placed in HL, else 0. The id as the first argument is either 0 or 1, depending on the controller that is to be read. The button as the second argument is the button that is to be read, for more info, see the controller chapter.
Parameters	**id:** The id parameter requires an unsigned short containing a 8 bit id. **btn:** The btn parameter requires an unsigned short containing a 8 bit button id.
Used by	**Interpreter.c**

Mnemonic (opcode)	WAITF (0x34)
Function header	inline void ip_waitFor(unsigned char id, unsigned char btn)
Summary	Pauses execution of a program until a specific button or buttoncombination is pressed. As long as the buttoncombination does not equal the given one, the program halts.
Parameters	**id:** The id parameter requires an unsigned short containing a 8 bit id. **btn:** The btn parameter requires an unsigned short containing a 8 bit button id.
Used by	**Interpreter.c**

2.8 Graphics

Mnemonic (opcode)	SCRH (0x35)
Function header	inline void ip_screenHeight()
Summary	Places the screenheight in the HL register.
Used by	**Interpreter.c**

Mnemonic (opcode)	SCRW (0x36)
Function header	inline void ip_screenWidth()
Summary	Places the screenwidth in the HL register.
Used by	**Interpreter.c**

Mnemonic (opcode)	SSPR (0x37)
Function header	inline void ip_setSprite(unsigned char index, unsigned char indexonscreen)
Summary	Places a sprite from code to VRAM. The first argument is the index in the sprite array in code. The second argument is the index in the on-screen sprite array.
Parameters	**index:** The index parameter requires an unsigned short containing a 8 bit index. **indexonscreen:** The indexonscreen parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	USPR (0x38)
Function header	inline void ip_updateSprite(unsigned char indexonscreen, unsigned char x, unsigned char y)
Summary	Updates the position of the sprite on the screen. Positioning the sprite out of the bounds of the screen makes the sprite invisible. The second and third argument contain respectively the register with the x position and the y position.
Parameters	**indexonscreen:** The indexonscreen parameter requires an unsigned short containing a 8 bit index. **x:** The x parameter requires an unsigned short containing a 8 bit address. **y:** The y parameter requires an unsigned short containing a 8 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	USPRI (0x4A)
Function header	inline void ip_updateSpriteImmediate(unsigned char indexonscreen, unsigned char x, unsigned char y)
Summary	Updates the position of the sprite on the screen. Positioning the sprite out of the bounds of the screen makes the sprite invisible. The second and third argument contain respectively the x position and the y position.
Parameters	**indexonscreen:** The indexonscreen parameter requires an unsigned short containing a 8 bit index. **x:** The x parameter requires an unsigned short containing a 8 bit value. **y:** The y parameter requires an unsigned short containing a 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	CSPR (0x39)
Function header	inline void ip_clearSprite(unsigned char indexonscreen)
Summary	Removes a sprite from the onscreen sprite index.
Parameters	**indexonscreen:** The indexonscreen parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	SPRX (0x3A)
Function header	inline void ip_checkSpriteX(unsigned char firstindexonscreen, unsigned char secondindexonscreen)
Summary	Calculates the horizontal distance between two sprites. The distance (in pixels) is placed in the HL register.
Parameters	**firstindexonscreen:** The firstindexonscreen parameter requires an unsigned short containing a 8 bit index. **secondindexonscreen:** The secondindexonscreen parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	SPRY (0x3B)
Function header	inline void ip_checkSpriteY(unsigned char firstindexonscreen, unsigned char secondindexonscreen)
Summary	Calculates the vertical distance between two sprites. The distance (in pixels) is placed in the HL register.
Parameters	**firstindexonscreen:** The firstindexonscreen parameter requires an unsigned short containing a 8 bit index. **secondindexonscreen:** The secondindexonscreen parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	SMAP (0x3C)
Function header	inline void ip_setMap(unsigned char index)
Summary	Draws a map on the screen. The first argument is the index of the map in the map memory.
Parameters	**index:** The index parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	SSTR (0x3D)
Function header	inline void ip_setString(unsigned char index, unsigned char x, unsigned char y)
Summary	Draws a string on the screen at a specified position. The first argument is the index of the string in the string memory.
Parameters	**index:** The index parameter requires an unsigned short containing a 8 bit index. **x:** The x parameter requires an unsigned short containing a 8 bit value. **y:** The y parameter requires an unsigned short containing a 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	DON (0x44)
Function header	inline void allOn()
Summary	Sets all pixels on the screen to the on-state. The actual framebuffer is not affected. To get back to drawing the buffer, call normalMode(). No redraw is needed to change the state.
Used by	**Interpreter.c**

Mnemonic (opcode)	DOFF (0x45)
Function header	inline void allOff()
Summary	Sets all pixels on the screen to the off-state. The actual framebuffer is not affected. To get back to drawing the buffer, call normalMode(). No redraw is needed to change the state.
Used by	**Interpreter.c**

Mnemonic (opcode)	DINV (0x46)
Function header	inline void invertedMode()
Summary	Sets the screen to drawing the framebuffer inverted. The actual framebuffer is not affected, nor the operation of its states. A cleared pixel will be lit, and a lit pixel will be cleared on screen. No redraw is needed to change the state.
Used by	**Interpreter.c**

Mnemonic (opcode)	DNRM (0x47)
Function header	inline void normalMode()
Summary	Sets the screen to drawing the framebuffer. No redraw is needed to change the state.
Used by	**Interpreter.c**

Mnemonic (opcode)	DRDRW (0x48)
Function header	inline void redraw()
Summary	Redraws the framebuffer to the screen. If allOn or allOff has been called right before a redraw, it is needed to call normalMode to draw the framebuffer.
Used by	**Interpreter.c**

Mnemonic (opcode)	DCLR (0x49)
Function header	inline void clearAll()
Summary	Clears the entire framebuffer by resetting its pixelvalues to zero. Will only be seen after a redraw. After a clearAll, the framebuffer can be written to immediately to update the screen.
Used by	**Interpreter.c**

2.9 Data read/writes

Mnemonic (opcode)	RDAT (0x4B)
Function header	inline void ip_readDataMemory(address)
Summary	Reads a byte from the cartridge data memory to the HL register.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address.
Used by	**Interpreter.c**

Mnemonic (opcode)	WDAT (0x4C)
Function header	inline void ip_writeToDataMemory(unsigned short address, unsigned char value)
Summary	Writes a register's value to the cartridge data memory.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	WDATI (0x4D)
Function header	inline void ip_writeToDataMemoryImmediate(unsigned short address, unsigned char value)
Summary	Writes a fixed value to the cartridge data memory.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address. **value:** The value parameter requires an unsigned short containing an 8 bit value.
Used by	**Interpreter.c**

2.10 Timer and Delays

Mnemonic (opcode)	DLY (0x3E)
Function header	inline void ip_delay(unsigned char index)
Summary	Waits for an amount of milliseconds equal to the value of the register given in the first argument.
Parameters	**index:** The index parameter requires an unsigned short containing a 8 bit index.
Used by	**Interpreter.c**

Mnemonic (opcode)	DLYI (0x3F)
Function header	inline void ip_delayImmediate(unsigned char value)
Summary	Waits for an amount of milliseconds equal to the fixed value given in the first argument.
Parameters	**value:** The value parameter requires an unsigned short containing a 8 bit value.
Used by	**Interpreter.c**

Mnemonic (opcode)	CTMR (0x40)
Function header	inline void ip_clearTimer()
Summary	Clears the timer by resetting its value to zero.
Used by	**Interpreter.c**

Mnemonic (opcode)	RTMR (0x41)
Function header	inline void ip_readTimer()
Summary	Places the value of the timer in the HL register.
Used by	**Interpreter.c**

Chapter 3

Function overview

3.1 Interpreter

3.1.1 Interpreter.h

Interpreter starts and initilizes the actual interpreter, for this it provides:

- void ip_initInterpreter()
- void ip_startInterpreter()

It requires all the functions from IOcontrol to function correctly.

Function header	void ip_initInterpreter()
Summary	Inits the interpreter by resetting all its properties. This should be called before each ip_startInterpreter().
Used by	**Interpreter.c**, **Interpreter.h**, **Main.c**

Function header	void ip_startInterpreter()
Summary	Starts the actual interpreter. Before each ip_startInterpreter call, ip_initInterpreter should be called unless the state of the interpreter is to be saved. To be able to execute code, a RAM module should be attached to the controller containing the code, sprites, maps and strings. Define DEBUGASSERT at the top to enable assert functions to be used. These asserts can help debugging of written code by checking if the code does what it's supposed to do.
Used by	**Interpreter.c**, **Interpreter.h**, **Main.c**

3.1.2 IOcontrol.h

OIcontrol provides the functionality for input and output to the interpreter. For this, it provides the following functions:

- void io_initSprites(unsigned short startAddress, unsigned short amount)
- void io_initSlaveSelect()
- unsigned char io_readProgramMemory(unsigned short address)
- unsigned char io_readDataMemory(unsigned short address)
- void io_writeToDataMemory(unsigned short address, unsigned char data)
- unsigned char io_readController(unsigned char controllerNumber)
- void io_initIO()
- void io_allOn()
- void io_allOff()
- void io_invert()
- void io_setCursor(unsigned char x , unsigned char y)
- void io_updateSprite(unsigned char indVram,unsigned char x, unsigned char y)

It requires these functions from SoftSPI:

- void ss_initSPI()
- void ss_sendByte()

And these functions from RAMdriver:

- void rd_initRAM()
- void rd_setBank(unsigned char bank)
- void rd_writeToRAM(unsigned short address, unsigned char value)
- unsigned char rd_readFromRAM(unsigned short address)

Function header	void io_initSlaveSelect()
Summary	
Used by	**IOcontrol.c**, **IOcontrol.h**

Function header	unsigned char io_readProgramMemory(unsigned short address)
Summary	This function reads 1 byte of data from the external program RAM chip at a given address. The program RAM contains the program code that is executed by the interpreter.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address. The program address range is from 0x00000 to 0x0FFFF.
Returns	This method returns a unsigned char containing the data that was stored at the given address.
Used by	**IOcontrol.c**, **IOcontrol.h**

Function header	unsigned char io_readDataMemory(unsigned short address)
Summary	This function reads 1 byte of data from the external data RAM chip at a given address. The data RAM contains the sprites, maps, strings, settings and savebanks.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address. The data address range is from 0xF0000 to 0xFFFFF.
Returns	This method returns a unsigned char containing the data that was stored at the given address.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_writeToDataMemory(unsigned short address, unsigned char data)
Summary	This function writes 1 byte of data to the external data RAM at a given address.
Parameters	**address:** The address parameter requires an unsigned short containing a 16 bit address. The data address range is from 0xF0000 to 0xFFFFF.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	unsigned char io_readController(unsigned char controllerNumber)
Summary	This function reads the game controller buttons state for a given controller number.
Parameters	**ControllerNumber:** This parameter contains the controller number: Controller1 = 0x00 Controller2 = 0xff
Returns	The method returns a unsigned char containing the controller button state. The meaning of each bit: BIT 0: A BIT 1: B BIT 2: SELECT BIT 3: START BIT 4: DPAD UP BIT 5: DPAD DOWN BIT 6: DPAD LEFT BIT 7: DPAD RIGHT
Used by	**IOcontrol.c**, **IOcontrol.h**

Function header	void io_initIO()
Summary	This function sets up the IOcontrol interface. It performs the following actions: - RAM interface setup - SoftSPI interface setup and slave select initialisation - Loads the sprites into the video memory - Loads the strings into the video memory - Loads the maps into the video memory
Note	This function has a long execution time.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**, **Main.c**

Function header	void io_allOn()
Summary	This function lights up all the pixels of the LCD screen without overwriting the video memory.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**, **Main.c**

Function header	void io_allOff()
Summary	This function disables all the pixels of the LCD screen without overwriting the video memory.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**, **Main.c**

Function header	void io_invert()
Summary	This function inverts all the pixels of the LCD screen overwriting the video memory. This means that a black pixel turns white and a white pixel turns into black.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_setCursor(unsigned char x , unsigned char y)
Summary	This function sets the coordinates of the LCD display in the videobuffer. TOTO: EXPLAIN The coordinate (0,0) is located top left.
Parameters	**x:** This parameter of type unsigned char contains the x-coordinate of the cursor. **y:** This parameter of type unsigned char contains the y-coordinate of the cursor.
Used by	**IOcontrol.h**

Function header	void io_updateSprite(unsigned char indexVram,unsigned char x, unsigned char y)
Summary	This function updates the position of a sprite in the video memory.
Parameters	**indexVram:** This parameter of type unsigned char container the index of the sprite in the video RAM. **x:** This parameter of type unsigned char contains the x-coordinate of the cursor. **y:** This parameter of type unsigned char contains the y-coordinate of the cursor.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_setSprite(unsigned char index, unsigned char indexVram)
Summary	This function initialize a sprite in the video memory at a given position.
Parameters	**index:** This parameter of type unsigned char contains the index of the sprite in the cartridge memory. **indexVram:** This parameter of type unsigned char container the index of the sprite in the video RAM. **x:** This parameter of type unsigned char contains the x-coordinate of the cursor. **y:** This parameter of type unsigned char contains the y-coordinate of the cursor.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**, **Main.c**

Function header	void io_clearSprite(unsigned char index)
Summary	This function clears a sprite given by the index parameter in the video memory.
Parameters	**index:** This parameter of type unsigned char contains the index of the sprite in the cartridge memory.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_drawString(unsigned char index, unsigned char x, unsigned char y)
Summary	This function draws a string given by the index parameter on the screen at a given position.
Parameters	**index:** This parameter of type unsigned char contains the index of the string in the cartridge memory. **x:** This parameter of type unsigned char contains the x-coordinate of the cursor. **y:** This parameter of type unsigned char contains the y-coordinate of the cursor.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_drawMap(unsigned char index)
Summary	This function draws a map with a given index on the LCD screen.
Parameters	**index:** This parameter of type unsigned char contains the index of the map in the cartridge memory.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_clearAll()
Summary	This function clears al the elements on the LCD screen.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

Function header	void io_clearStrings()
Summary	This function clears the string buffer.
Used by	**IOcontrol.c**, **IOcontrol.h**

Function header	void io_clearGraphics()
Summary	This function clears the display buffer.
Used by	**IOcontrol.c**, **IOcontrol.h**

Function header	void io_redraw()
Summary	This function redraws the displaybuffer to the LCD screen.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**, **Main.c**

Function header	void io_normalMode()
Summary	This function sets the LCD screen back to normal mode.
Used by	**Interpreter.c**, **IOcontrol.c**, **IOcontrol.h**

3.1.3 RAMdriver.h

RAMdriver is the driver responsible for the communication of the interpreter with the cartridge. It provides the following functions:

- void rd_initRAM()
- void rd_setBank(unsigned char bank)
- unsigned char rd_getBank()
- void rd_writeToRAM(unsigned short address, unsigned char value)
- unsigned char rd_readFromRAM(unsigned short address)

Function header	void rd_initRAM()
Summary	
Used by	**IOcontrol.c**, **RAMdriver.c**, **RAMdriver.h**

Function header	void rd_setBank(unsigned char bank)
Summary	This function selects a memorybank.
Parameters	**bank:** The bank parameter contains the selected bank: PROGMEM = 0, DATAMEM = 1
Used by	**IOcontrol.c**, **RAMdriver.c**, **RAMdriver.h**

Function header	unsigned char rd_getBank()
Summary	Function to get the selected bank
Returns	The method returns a unsigned char containing the selected bank
Used by	**RAMdriver.c**, **RAMdriver.h**

Function header	void rd_writeToRAM(unsigned short address, unsigned char value)
Summary	Function to write data (1 byte) to the external RAM in a given address
Parameters	**address:** The address parameter requires an 16 bit address (unsigned short) **value:** Parameter value is helds the value that needs to be stored in the ROM
Used by	**IOcontrol.c**, **RAMdriver.c**, **RAMdriver.h**

Function header	unsigned char rd_readFromRAM(unsigned short address)
Summary	Function the read data (1 byte) from a given address
Parameters	**address:** The address parameter requires an 16 bit address (unsigned short).
Returns	The method returns a unsigned char containing the data in the address
Used by	**RAMdriver.c**, **RAMdriver.h**

3.2 Graphics controller

3.2.1 CommandControl.h

This file provides the functions needed to execute the sent commands.
It provides:

- cc_initCommandControl()
- cc_spiCallback()
- cc_control()

It requires these functions from HardSPI:

- eventhandler()
- hs_initSPI()
- hs_readByte()

And these from GraphicsControl:

- gc_loadSprite(unsigned char refNr, unsigned char *data)
- gc_setSprite(unsigned char refNr, unsigned char listNr, unsigned char x, unsigned char y)
- gc_clearSprite(unsigned char listNr)
- gc_updateSprite(unsigned char listNr, unsigned char x, unsigned char y)
- gc_loadMap(unsigned char mapNr)
- gc_drawMap(unsigned char mapNr, unsigned char xoff, unsigned char yoff)
- gc_loadString(unsigned char stringNr)
- gc_drawString(unsigned char x, unsigned char y, unsigned char stringNr)
- gc_redraw()
- gc_allOn()
- gc_allOff()
- gc_invertedMode()
- gc_normalMode()
- gc_clearAll()
- gc_clearGraphics()
- gc_clearStrings()

Function header	void cc_initCommandControl()
Summary	This initializes the command control, calls the needed methods to set some registers
Used by	**CommandControl.c**, **CommandControl.h**, **Graphics.c**

Function header	void cc_spiCallback()
Summary	Contains the code that must be executed on SPI interrupt This interrupt is triggered on receiving a full byte
Used by	**CommandControl.c**, **CommandControl.h**, **Eventhandler.c**

Function header	void cc_control()
Summary	This method does all the command handling It checks the receive buffer and calls the correct middleware methods
Used by	**CommandControl.c**, **CommandControl.h**, **Graphics.c**

3.2.2 GraphicsControl.h

This file provides the functionality to actually draw to the screen.
For this, it provides:

- gc_drawPixel(unsigned char x, unsigned char y, unsigned char color)
- gc_setPixel(unsigned char x, unsigned char y)
- gc_clearPixel(unsigned char x, unsigned char y)
- gc_drawByte(unsigned char x, unsigned char y, unsigned char data)
- gc_testbit(unsigned char data, unsigned char number)
- gc_loadSprite(unsigned char refNumber, unsigned char *data)
- gc_setSprite(unsigned char refNumber, unsigned char listNr, unsigned char x, unsigned char y)
- gc_clearSprite(unsigned char listNr)
- gc_updateSprite(unsigned char listNr, unsigned char x, unsigned char y)

- gc_loadMap(unsigned char mapNr)
- gc_drawMap(unsigned char mapNr, unsigned char xoffset, unsigned char yoffset)
- gc_loadString(unsigned char stringNr)
- gc_drawString(unsigned char x, unsigned char y, unsigned char stringNr)
- gc_initLCD()
- gc_resetCursor()
- gc_dataMode()
- gc_commandMode()
- gc_sendCommand()
- gc_sendData()
- gc_redraw()
- gc_allOn()
- gc_allOff()
- gc_invertedMode()
- gc_normalMode()
- gc_clearAll()
- gc_clearGraphics()
- gc_clearStrings()

It requires these functions from HardSPI:

- hs_readByte()
- hs_dataAvailable()

And these from SoftSPI:

- ss_initSPI()
- ss_sendByte(unsigned char data)

Function header	void gc_drawPixel(unsigned char xpos, unsigned char ypos, unsigned char color)
Summary	Draws a pixel with a specified color at a specified location on the screen
Parameters	**xpos:** Unsigned char denoting the x position on the screen **ypos:** Unsigned char denoting the y position on the screen **color:** Unsigned char denoting the color of the pixel
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_setPixel(unsigned char xpos, unsigned char ypos)
Summary	Draws a pixel with color = 1 at a specified location on the screen
Parameters	**xpos:** Unsigned char denoting the x position on the screen **ypos:** Unsigned char denoting the y position on the screen
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_clearPixel(unsigned char xpos, unsigned char ypos)
Summary	Draws a pixel with color = 0 at a specified location on the screen
Parameters	**xpos:** Unsigned char denoting the x position on the screen **ypos:** Unsigned char denoting the y position on the screen
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_drawByte(unsigned char xpos, unsigned char ypos, unsigned char data)
Summary	Draws 8 subsequent pixels starting at a specified location on the screen The folowing pixels are each drawn below the previous drawn pixel (ie. advancing in y position) If the end of the screen is reached, the rest of the pixels is not drawn
Parameters	**xpos:** Unsigned char denoting the starting x position on the screen **ypos:** Unsigned char denoting the starting y position on the screen **data:** Unsigned char containing the pixel data, is used lsb first
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	unsigned char gc_testbit(unsigned char data, unsigned char number)
Summary	Test the number bit of a certain byte data
Parameters	**data:** An unsigned char denoting the data byte we want a to test a bit from **number:** The nth bit from the data byte we want to test, with nth being equal to number
Returns	Unsigned char, in fact return is restricted to 0 or 1
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_loadSprite(unsigned char refNumber, unsigned char *data)
Summary	This method gets the pixel data for a sprite from the SPI receive buffer and puts it in the correct memory slot. The memory slot is given by the spritenumber.
Parameters	**refNumber:** The number of the sprite in memory. **data:** Takes a pointer to the start of the pixel data for the sprite, by definition a sprite has 8 bytes of pixeldata, they are found subsequent at the pointer location.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_setSprite(unsigned char refNumber, unsigned char listNumber, unsigned char xpos, unsigned char ypos)
Summary	Enables one to draw a sprite on the screen by taking a reference from the sprite in memory and mapping it to a sprite on the screen. It can be set at any location on the screen, if the sprite position is out of screen boundries nothing will be drawn. Correct sprite numbers are from 1 upto 32, which means a maximum of 32 sprites can be drawn on the screen at once. One sprite in memory can have multiple entries in the list with sprites to draw on the screen.
Parameters	**refNumber:** This is the number of the sprite in memory. This is also the reference to the actual sprite pixel data. **listNumber:** This is the number of the sprite on the screen. Can be different from the number of the sprite in memory. **xpos:** An unsigned char denoting the X postion on the screen. This means the left edge. **ypos:** An unsigned char denoting the Y postion on the screen. This means the top edge.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_clearSprite(unsigned char listNumber)
Summary	Removes a sprite from the list with sprites to be drawn on the screen.
Parameters	**listNumber:** The number of the sprite on the screen to be removed.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_updateSprite(unsigned char listNumber, unsigned char xpos, unsigned char ypos)
Summary	Changes a sprite to a new postion on the screen.
Parameters	**listNumber:** The number of the sprite on the screen from which the position will be changed. **xpos:** An unsigned char denoting the new X postion on the screen. This means the left edge. **ypos:** An unsigned char denoting the new Y postion on the screen. This means the top edge.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_loadMap(unsigned char mapNumber)
Summary	Method to receive the data for a map. A map exist of tiles who are in fact sprites. A map is 16 tiles wide and 8 tiles high, which correspond to 128 x 64 pixels. A map is tiled from left to right and then from top to bottom. This method expects that 128 bytes with a reference to a sprite are send over SPI, after entering the method.
Parameters	**mapNumber:** Denotes where to write the map data on receiving. The possible maps are from 0 upto 3.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_drawMap(unsigned char mapNumber, unsigned char xoffset, unsigned char yoffset); // offset in pixels
Summary	This method send the visible part of the map to the VRAM. Because the screen is only 84 x 48 pixels and the map is 128 x 64 pixels one needs to specify the visible part. The visible part is selected with the X and Y offsets.
Parameters	**mapNumber:** Tells the method from which map the part should be draw. **xoffset:** An unsigned char denoting the starting X postion on the map. This means the left edge of the screen. **yoffset:** An unsigned char denoting the starting Y postion on the map. This means the top edge of the screen.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_loadString(unsigned char stringNumber)
Summary	This method loads a string into the string memory. The stringnumber is the reference for future usage of the string. A string ends with a NULL character or when the string memory is full, which is 254 bytes.
Parameters	**stringNumber:** The stringnumber reference.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_drawString(unsigned char xpos, unsigned char ypos, unsigned char stringNumber)
Summary	This method draws a string onto the screen. It does so by writing the TRAM, so text can't conflict with graphics.
Parameters	**xpos:** An unsigned char denoting the X postion on the screen. The text starts at this position an continues to the right. **ypos:** An unsigned char denoting the Y postion on the screen. This means the top line of the text. **stringNumber:** Specifies the string from memory to be draw on the screen.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_initLCD()
Summary	Initializes the LCD module. This means sending commands to the LCD for setting backplane voltage, contrast ... It also initializes the software SPI internally.
Used by	**Graphics.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_resetCursor()
Summary	Sets X and Y address of VRAM to zero in the LCD module.
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_dataMode()
Summary	Selects the datamode of the LCD, in which pixeldata can be sent to it.
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_commandMode()
Summary	Selects the commandmode of the LCD, in which commands can be sent to it.
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_sendCommand(unsigned char command)
Summary	Sends a commandbyte to the lcd.
Parameters	**command:** An unsigned char denoting the command byte to send.
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_sendData(unsigned char data)
Summary	Sends a byte containing data for 8 pixels.
Parameters	**data:** An unsigned char denoting the pixel data to send.
Used by	**GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_redraw()
Summary	Sends the whole VRAM and TRAM from the controller to the LCD module.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_allOn()
Summary	Sends a command to turn on all the pixels of the LCD. Can be used to test the LCD for defects.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_allOff()
Summary	Sends a command to turn off all the pixels of the LCD.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_invertedMode()
Summary	Sends a command to enable the inverted pixelmode of the LCD. This inverts each and every pixel on the screen.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_normalMode()
Summary	Sends a command to disable the inverted pixelmode.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_clearAll()
Summary	Clears both the VRAM and TRAM by filling them with zeros.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_clearGraphics()
Summary	Clears the VRAM by filling it with zeros.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

Function header	void gc_clearStrings()
Summary	Clears the TRAM by filling it with zeros.
Used by	**CommandControl.c**, **GraphicsControl.c**, **GraphicsControl.h**

3.2.3 HardSPIdriver.h

This file provides the functions needed for the hardware SPI.

- hs_initSPI()
- hs_sendByte(unsigned char data)
- hs_readByte()
- hs_dataAvailable()
- hs_reset()

Function header	void hs_initSPI()
Summary	Initialisation of the hardware SPI pins. Sets all the registers needed for a correct working SPI peripheral Also sets the registers needed for interrupt callback
Used by	**CommandControl.c**, **HardSPIdriver.c**, **HardSPIdriver.h**

Function header	bool hs_sendByte(unsigned char data)
Summary	Sends a byte over SPI to the selected slave.
Parameters	**data:** The data argument is an unsigned char, the byte we want to send.
Returns	The method returns a boolean, this denotes the if the operation was succesfull.
Used by	**HardSPIdriver.c**, **HardSPIdriver.h**

Function header	unsigned char hs_readByte()
Summary	Reads a byte over SPI from the selected slave.
Returns	The data argument is an unsigned char, the byte we want to send.
Used by	**HardSPIdriver.c**, **HardSPIdriver.h**

Function header	bool hs_dataAvailable()
Summary	Check whether there is new data available since the last read or write operation.
Returns	Returns a boolean, denoting the availability of new data.
Used by	**GraphicsControl.c**, **HardSPIdriver.c**, **HardSPIdriver.h**

Function header	void hs_reset()
Summary	Method for resetting the SPI module after a collision or overflow has happened. Does not clear the buffer!
Used by	**HardSPIdriver.c**, **HardSPIdriver.h**

3.2.4 SoftSPIdriver.h

This file provides the functions needed for the software SPI.

- ss_initSPI()
- ss_sendByte(unsigned char data)
- ss_readByte()

Function header	void ss_initSPI()
Summary	Initialisation of the software SPI pins. Sets all the output pins low. Mind the tris register, this has to be set manually! The standard in this case is trisa = 0b00000100;
Used by	**GraphicsControl.c**, **SoftSPIdriver.c**, **SoftSPIdriver.h**

Function header	void ss_sendByte(unsigned char data)
Summary	Sends a byte over SPI to the selected slave.
Parameters	**data:** The data argument is an unsigned char, the byte we want to send.
Used by	**GraphicsControl.c**, **SoftSPIdriver.c**, **SoftSPIdriver.h**

Function header	unsigned char ss_readByte()
Summary	Reads a byte over SPI from the selected slave.
Returns	The method returns an unsigned char, the data received over SPI.
Used by	**SoftSPIdriver.c**, **SoftSPIdriver.h**

Appendices

Appendix A

Code example

With this example code, it is possible to move a sprite around on the display using the D-Pad of the first controller. This way it uses graphical initialization of a sprite, controller reading, mathematics, jumps and calls and should be enough to acquire a basic understanding of how the language is used.

```
1  #define spriteX 0
2  #define spriteY 1
3  #define controller 0
4  #define ballIndex 0
5
6  #define K_UP     0b10000000
7  #define K_DOWN   0b01000000
8  #define K_LEFT   0b00100000
9  #define K_RIGHT  0b00010000
10 #define K_SELECT 0b00001000
11 #define K_START  0b00000100
12 #define K_A 0b00000010
13 #define K_B 0b00000001
14
15 setup:
16   LD     spriteX, 38 ;Set sprite x pos to 38
17   LD     spriteY, 20 ;Set sprite y pos to 20
18   SSPR   ballIndex, ballIndex ;Copy sprite to vram index 0
19   SSTR   9,21 ;Draw string start in center of the screen
20   WAITF controller, K_START ;Wait for the user to press [start]
21   DNRM ;Set display to normal mode
22   DCLR ;Clear all content on the screen
23   DRDRW ;Redraw the screen
24
25 loop:
26   DCLR ;Clear all content on the screen
27   BRDS   controller, K_UP ;Check if the up key was pressed
28   CALLNZ incx ;Call incx if the up key was pressed
29   BRDS   controller, K_DOWN ;Check if the down key was pressed
30   CALLNZ decx ;Call decx if the down key was pressed
31   BRDS   controller, K_RIGHT ;Check if the right key was pressed
32   CALLNZ incy ;Call incy if the right key was pressed
33   BRDS   controller, K_LEFT ;Check if the left key was pressed
```

```
34    CALLNZ decy ;Call decy if the left key was pressed
35    USPR  ballIndex, spriteX, spriteY ;Update the sprite on screen
36    DRDRW ;Redraw the screen
37    JMP   loop ;Restart loop
38
39  incx:
40    INC spriteX ;Increment spriteX by one
41    RET ;Return to calling point
42
43  incy:
44    INC spriteY ;Increment spriteY by one
45    RET ;Return to calling point
46
47  decx:
48    DEC spriteX ;Decrement spriteX by one
49    RET Return to calling point
50
51  decy:
52    DEC spriteY ;Decrement spriteY by one
53    RET ;Return to calling point
```

This results in the following bytecode:

```
1   0x01 0x00 0x26
2   0x01 0x01 0x14
3   0x37 0x00 0x00
4   0x3D 0x09 0x15
5   0x34 0x00 0x80
6   0x47
7   0x49
8   0x48
9
10  0x49
11  0x33 0x00 0x80
12  0x42 0x00 0x51
13  0x33 0x00 0x80
14  0x42 0x00 0x57
15  0x33 0x00 0x80
16  0x42 0x00 0x54
17  0x33 0x00 0x80
18  0x42 0x00 0x60
19  0x38 0x00 0x00 0x01
20  0x48
21  0x27 0x00 0x18
22
23  0x15 0x00
24  0x2E
25  0x15 0x01
26  0x2E
27  0x16 0x00
28  0x2E
29  0x16 0x01
30  0x2E
```

Appendix B

Layers

This infograph shows the structure of the layers in this project. It consists of three software layers, plus one hardware layer.

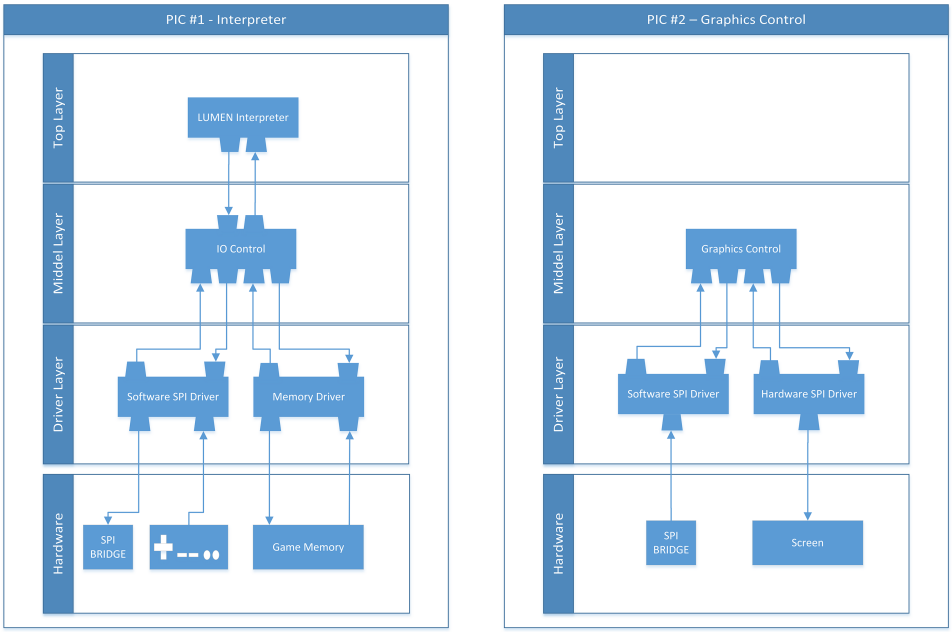


FIGURE B.1: Layer Overview