Index

Modules

Drivers

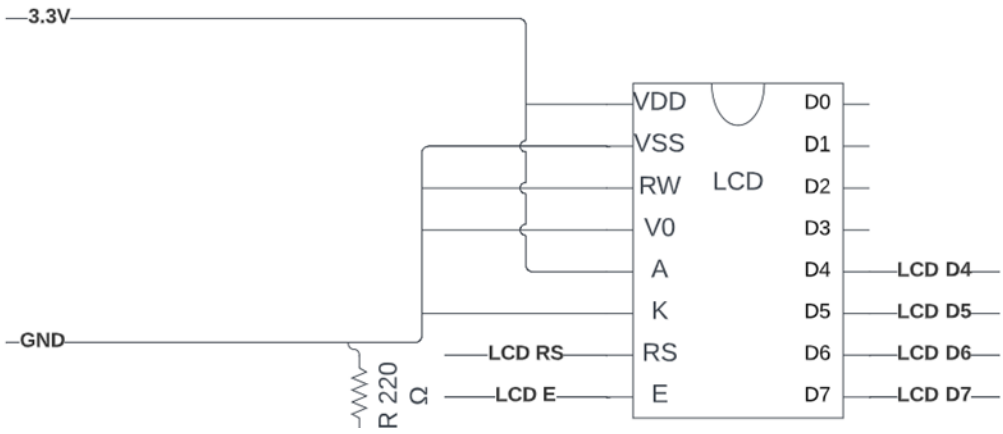
Here are grouped all the modules developed to provide a higher abstraction interface to the hardware components of the project.

As the majority of modules, the share information with the utilizer via pointers to variables given to the setup procedure.

LCD

Development

Used to show the serial number of the bomb and the remaining time to game over.



As shown in the picture on top, the LCD display needs to be connected to 3.3V trough VDD pin and to GND (ground) trough the VSS pin. The RW pin is connected to GND since we will always use it in writing mode that corresponds to logical low. V0 sets the contrast and is also connected to GND since it gives a good contrast. A and K power the led. RS and E are used to control the display:

- RS stands for register select, it is used to declare if I'm sending a command (low) or data (high)
- E stands for enable; it is used to tell the display when to listen to the data line (Dx)

These are managed by the driver as needed.

At last, the pins D4, D5, D6 and D7 are used to send the command/data. To use as less pins as possible the 4 pin mode is used, where the data is sent in a serial: first the upper nibble and then the lower, the exact transmission is addressed later.

To understand how to use the display the following documentation was used:

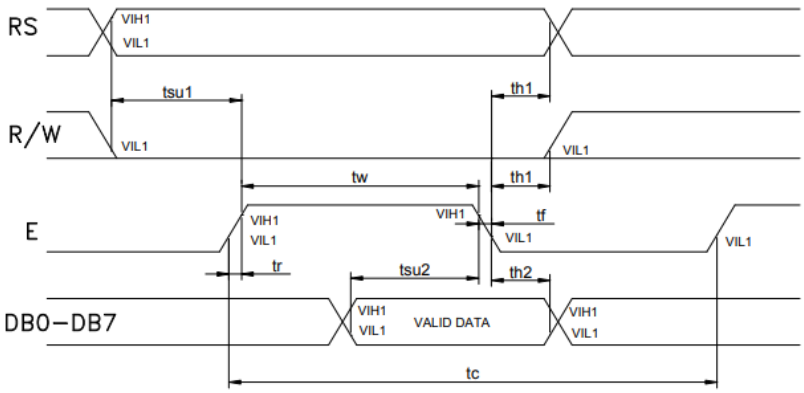
https://components101.com/sites/default/files/component_datasheet/16x2%20LCD%20Datasheet.pdf.

Here are referenced only the important bits.

Instruction	Instruction code										Description	Execution time (fosc= 270 KHZ
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write “20H” to DDRA and set DDRAM address to “00H” from AC	1.53ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to “00H” From AC and return cursor to Its original position if shifted. The contents of DDRAM are not changed.	1.53ms
Entry mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction And blinking of entire display	39us
Display ON/ OFF control	0	0	0	0	0	0	1	D	C	B	Set display (D), cursor (C), and Blinking of cursor (B) on/off Control bit.	
Cursor or Display shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display Shift control bit, and the Direction, without changing of DDRAM data.	39us
Function set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL: 8- Bit/4-bit), numbers of display Line (N: =2-line/1-line) and, Display font type (F: 5x11/5x8)	39us
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address Counter.	39us
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address Counter.	39us
Read busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal Operation or not can be known By reading BF. The contents of Address counter can also be read.	0us
Write data to Address	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43us
Read data From RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43us

The table contains the commands that can be sent to the display. From it we can also see how to write a character, the data sent is just the 8 bits of the char.

The following schema shows how to correctly send the data to the display in 8 pin mode:



Working with 4, instead, we need to use the E pin ad a clock:

- Set the 4 most significant bits of the data/command to send and the RS pin
- Set the E pin to high and then low
- Set the other 4 bits of the data/command
- Set the E pin to high and then low

The display will now put the two parts together and execute the command or display the character.

However, the display is set to 8-bit mode at startup. To modify it I have to send the command: Function set with DL at low, I must therefore send the bits: 00100000. We will therefore have to set our four ports to 0010, since pins D1-D4 are not connected, they are seen as 0.

One problem is that the display, for various reasons, may not have reset and therefore be in 4-bit mode, this would lead the display to think that 0010 is the upper nibble of the command I want to send, thus breaking all subsequent communications.

For this reason, I send the 0x02 command as if I were already in 4 bit mode, i.e. sending the upper first and then the lower nibble. This is because if the display is in 8-bit mode it will see two commands:

1) 00000000: which corresponds to a "NOP"
2) 00100000: which sends the display to 4-bit mode

If, on the other hand, the display is already in 4-bit mode, the interpreted command will be precisely 0x02 which corresponds to return home, which does not produce any unwanted effect during the setup procedure, as the clear display command is then sent.

To write on the display, the $displayLCD(rowSel\ row, char\ string[], int\ len)$ function is provided to pass an array of chars, the length of the string contained in it and on which row (ROW_1 or ROW_2) of t he display to display it.

Testing

To evaluate the correct functioning of the driver, two simple projects were written which sent the string "Hello world!" to the screen, reversing the order of the words between the lines, and one which mirrored the messages received via UART with scrolling.

Uart_d

To communicate with the raspberry pi I need to open a connection on the USB port. To do this you need to use uart and via the UART_A0 port.

Initialization

To initialize the communication it is necessary to fill the following structure:

```
typedef struct _eUSCI_eUSCI_UART_ConfigV1
{
    uint_fast8_t selectClockSource;
    uint_fast16_t clockPrescalar;
    uint_fast8_t firstModReg;
    uint_fast8_t secondModReg;
    uint_fast8_t parity;
    uint_fast16_t msborLsbFirst;
    uint_fast16_t numberofStopBits;
    uint_fast16_t uartMode;
    uint_fast8_t overSampling;
    uint_fast16_t dataLength;
} eUSCI_UART_ConfigV1;
```

You can get a detailed description of the fields of the structure in the definition of the UART_initModule function:

- selectClockSource selects Clock source. Valid values are:
    - EUSCI_A_UART_CLOCKSOURCE_SMCLK
    - EUSCI_A_UART_CLOCKSOURCE_ACLK
- clockPrescalar is the value to be written into UCBRx bits
- firstModReg is First modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device User Guide. This value is written into UCBRFx bits of UCAxMCTLW.
- secondModReg is Second modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device User Guide. This value is written into UCBRSx bits of UCAxMCTLW.
- parity is the desired parity. Valid values are:
    - EUSCI_A_UART_NO_PARITY [Default Value],
    - EUSCI_A_UART_ODD_PARITY,
    - EUSCI_A_UART_EVEN_PARITY

- msborLsbFirst controls direction of receive and transmit shiftregister. Valid values are:
  - EUSCI_A_UART_MSB_FIRST
  - EUSCI_A_UART_LSB_FIRST [Default Value]
- numberofStopBits indicates one/two STOP bits. Valid values are:
  - EUSCI_A_UART_ONE_STOP_BIT [Default Value]
  - EUSCI_A_UART_TWO_STOP_BITS
- uartMode selects the mode of operation. Valid values are:
  - EUSCI_A_UART_MODE [Default Value],
  - EUSCI_A_UART_IDLE_LINE_MULTI_PROCESSOR_MODE,
  - EUSCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR_MODE,
  - EUSCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION_MODE
- overSampling indicates low frequency or oversampling baud generation. Valid values are:
  - EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION
  - EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION
- dataLength indicates Character length. Selects 7-bit or 8-bit character length. Valid values are:
  - EUSCI_A_UART_8_BIT_LEN
  - EUSCI_A_UART_7_BIT_LEN

The parameters we use are:

```
const eUSCI_UART_ConfigV1 UARTConfig=
    {
            EUSCI_A_UART_CLOCKSOURCE_SMCLK,
            26,
            0,
            111,
            EUSCI_A_UART_NO_PARITY,
            EUSCI_A_UART_LSB_FIRST,
            EUSCI_A_UART_ONE_STOP_BIT,
            EUSCI_A_UART_MODE,
            EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION,
            EUSCI_A_UART_8_BIT_LEN
    };
```

Calculated using the 48Mhz clock.

communication

This module was developed relatively recently compared to the rest of the application, so we didn't want to be forced to develop the rest around sub-optimal communication. We therefore defined a simple communication protocol that was as generic as possible:

- The communication starts with N header characters (8 bits each) which define the size of the message to be sent, with N settable at the module set-up
- Send acceptance character "B" or refusal "R" of the communication (only for raspy→MSP)
- Followed by M characters, with M being the size specified in the header

The management of this protocol takes place within the IRQ of the UART, to notify the main process of the arrival of a message I use two pointers to external variables passed during setup, one points to an integer at 0 which represents the size of the message and the other to a pointer to char on which the address of the first character of the vector containing the message will be written. When the IRQ stops receiving a message, it sets the variable pointed to the size of the message.

To ensure I don't overwrite the message while the main program is processing it, the IRQ will not accept any more messages, sending the R character upon receipt of the header, until the message size is reset to 0, signalling the termination of the message. message processing.

The message vector is dynamically generated in the heap, the burden of freeing the memory is left to main which will therefore have to execute it before setting the size to 0.

Acceptance of communication is not performed by the MSP to raspberry pi as the latter uses the pySerial library which implements a character buffer, so I don't have the problem of losing a message.

Sending messages is performed through the function defined in the module $sendUART(char * data, unsigned\ int\ size)$ to which I have to pass the vector containing the message and its size.

Testing

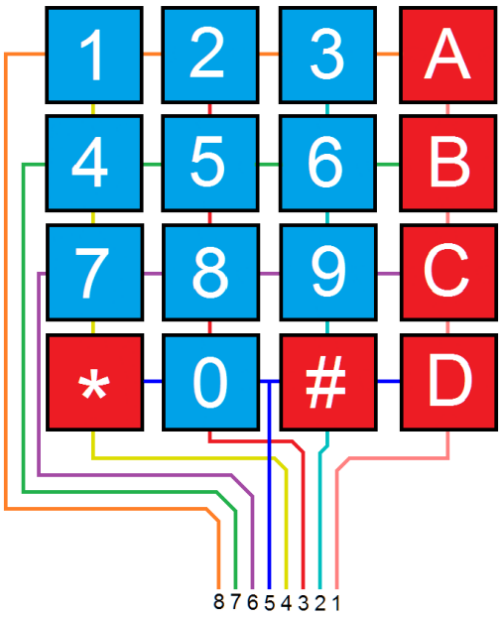
È stato scritto un programma che riceve via UART un JSON e risponde con la corretta ricezione ed elaborazione del messaggio.

È stato poi utilizzato integrato ad altri moduli per testare quest'ultimi quali *LCD* e *gameParser*.

Keypad

implementation

To understand how I use the keypad, I need to take a look at its circuit diagram available at page 2 of the following pdf: https://cdn.sparkfun.com/assets/f/f/a/5/0/DS-16038.pdf.



As I can see it doesn't have a VCC line to power it, I can't just listen to the 8 pins and wait for 2 (row and col) of them to be powered.

Let's solve it step by step.

To be able to detect a button press I set all columns as input pins and set their IRQs. To give them power I set all rows to output as high, so that when a button is pressed the column will be powered by that row calling the IRQ.

Using the interrupt flags, I are able to detect which one of the columns contain the pressed button. To detect the row, I need to selectively power off each row; when the column does not receive power anymore, I successfully identified the row that contains the button.

Given the row and column I can say which button was pressed.

A problem I encountered with this solution was the time the IRQ took to check every column and row. To solve this I rewrote the IRQ to only detect the button press and notify the main process about it, using the same variable that will contain the pressed button. The main process will then call the method $evaluateKeyPad()$ that will write to a memory position, containing a char passed at setup, the pressed button.

Another problem was button debouncing. After releasing the button, often the IRQ was wrongly called trying to find a button that wasn't being pressed. To solve this in $evaluateKeyPad()$ before evaluating the button I wait some time for the signal to be stable and then try to evaluate the button, if no button is detected then it was a bouncing signal and an error char "E" is written.

Testing

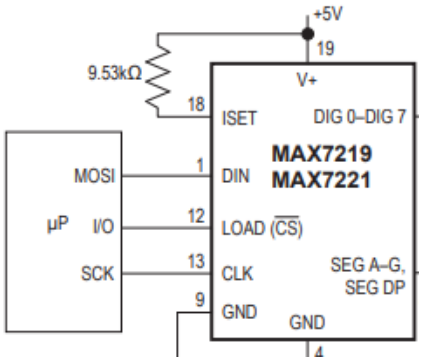
To test the correct functioning of this module I simply wrote a program that printed the detected button.

Dot matrix

SPI

The led matrix to operate uses the controller XXXXXX for 8 segment displays, which datasheet is available at the following link: https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf.

As I can see from the datasheet, I can use SPI to communicate with the controller. The circuit diagram of the controller is as follows:



I choose to connect the controller to EUSCI_B2, that maps to pin 3.5 for SCK and 3.6 for MOSI (master output slave input).

To set up the connection I need to populate the following struct:

```
typedef struct _eUSCI_SPI_MasterConfig
{
    uint_fast8_t selectClockSource;
    uint32_t clockSourceFrequency;
    uint32_t desiredSpiClock;
    uint_fast16_t msbFirst;
    uint_fast16_t clockPhase;
    uint_fast16_t clockPolarity;
    uint_fast16_t spiMode;
} eUSCI_SPI_MasterConfig;
```

Which parameters are explained as follows:

- selectClockSource selects clock source. Valid values are:
  - EUSCI_SPI_CLOCKSOURCE_ACLK
  - EUSCI_SPI_CLOCKSOURCE_SMCLK
- clockSourceFrequency is the frequency of the selected clock source
- desiredSpiClock is the desired clock rate for SPI communication
- msbFirst controls the direction of the receive and transmit shift register. Valid values are:
  - EUSCI_SPI_MSB_FIRST
  - EUSCI_SPI_LSB_FIRST [Default Value]
- clockPhase is clock phase select. Valid values are:
  - EUSCI_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT [Default Value]
  - EUSCI_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT
- clockPolarity is clock polarity select. Valid values are:
  - EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_HIGH
  - EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_LOW [Default Value]
- spiMode is SPI mode select. Valid values are:
  - EUSCI_SPI_3PIN [Default Value]
  - EUSCI_SPI_4PIN_UCxSTE_ACTIVE_HIGH
  - EUSCI_SPI_4PIN_UCxSTE_ACTIVE_LOW

I populated it as follows (I assume the clock speed is already set to 48 MHz when setting up this module):

```
const eUSCI_SPI_MasterConfig matrixConfig = {
                                EUSCI_SPI_CLOCKSOURCE_SMCLK,
                                48000000,
                                10000000,
                                EUSCI_SPI_MSB_FIRST,
                                EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT ,
                                EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_LOW,
                                EUSCI_SPI_3PIN
};
```

To send data (8-bit values) I just need to call the driverLib method:

```
SPI_transmitData(eUSCIPort, data);
```

Communication

To communicate with the controller, I need to refer to its datasheet. I can look at the following table:

REGISTER	ADDRESS					HEX CODE
	D15–D12	D11	D10	D9	D8	
No-Op	X	0	0	0	0	0xX0
Digit 0	X	0	0	0	1	0xX1
Digit 1	X	0	0	1	0	0xX2
Digit 2	X	0	0	1	1	0xX3
Digit 3	X	0	1	0	0	0xX4
Digit 4	X	0	1	0	1	0xX5
Digit 5	X	0	1	1	0	0xX6
Digit 6	X	0	1	1	1	0xX7
Digit 7	X	1	0	0	0	0xX8
Decode Mode	X	1	0	0	1	0xX9
Intensity	X	1	0	1	0	0xXA
Scan Limit	X	1	0	1	1	0xXB
Shutdown	X	1	1	0	0	0xXC
Display Test	X	1	1	1	1	0xXF

As I can see the communication needs to send 16 bits: the last 8 represents the command (of which only 4 are used) and the others are the "payload".

In case of *digit X* the payload will be what to show on the digit X (in our case row X) of the display, if decode mode is disabled the 8 bits will represent the 8 segments, otherwise the number sent will be interpreted and display the number itself (in our case it won't work since it's not attached to an 8 segment display).

The command *Decode Mode* activates and deactivates the homonymous mode, bit n of the payload defines the state of decode mode of digit n.

Scan limit is used to set the number of digits displayed.

Shutdown and *intensity* do as the names suggest. *Display test* turns on all led at the same time.

Set-up

To set up the matrix the following commands are sent:

```
sendCmdMat(MAT_OP_SCANLIMIT, 0x0f);
sendCmdMat(MAT_OP_INTENSITY, 0x0e);
sendCmdMat(MAT_OP_DECODEMODE, 0x00);
```

the first one tells the controller to display all numbers, the second sets the intensity and the last one deactivates the decode mode for all the digits.

To show something on the matrix I call the module function $sendMat(char\ mat[8])$ that uses one bit for every led state (8×8).

Testing

To test the module, I wrote a simple program that sends a configuration of LEDs to be shown:

```
char mat[8]={
                0b 0 0 1 0 0 1 0 0,
                0b 1 1 1 1 1 1 1 1,
                0b 1 1 1 1 1 1 1 1,
                0b 0 1 1 1 1 1 1 0,
                0b 0 1 1 1 1 1 1 0,
                0b 0 0 1 1 1 1 0 0,
                0b 0 0 0 1 1 0 0 0,
                0b 0 0 0 0 0 0 0 0
};
sendMat(mat);
```

ADC subscription

Since multiple modules will use the ADC, I either had to pack all the I/O into a single IRQ handler or do a subscription system as I choose.

This module let us "subscribe" to an ADC interrupt and have a function be called by the IRQ when it's triggered.

Since I assumed to not know which input and interrupt will be needed the module only initializes the core of ADC by calling the following functions:

```
//init module
ADC14_enableModule();
ADC14_initModule(ADC_CLOCKSOURCE_ADCOSC, ADC_PREDIVIDER_64, ADC_DIVIDER_8, 0);
```

Other modules will then be able to subscribe to the ADC by calling:

```
int registerADC(void (*intHandler)(void))
```

The information of this function is saved to an array and the index is returned to the caller. This can be later be used to temporarily deactivate or activate the interrupt (doesn't actually mask the interrupt but doesn't call the function).

Another significant note is that the register function does not set the pins mode nor the ADC memory map, those will have to be handled by the caller.

After setting all the mapping, $enableConvADC()$ can be called that sets the MultiSequenceMode for the first 6 (MEM0-MEM5) memories and starts the conversion.

Testing

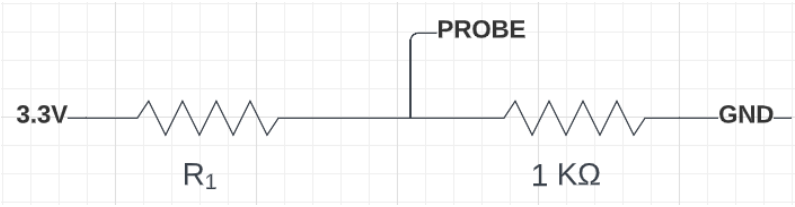
This module functions were tested using it to implement the modules "sonda", "accelerometer" and "joystick" that make use of it.

Sonda

circuit

This module manages an analog input and use it to discern between 9 different interactions using different voltage values for each.

To do this I had to decide 9 different voltage levels to use and calculate the correct resistors to use to be able to archive it. In general, the circuit for each "button" is as follows:



So, the voltage sensed by the probe will be:

$$V_p = V_{in} * \frac{R_1 + R_g}{R_g} = 3.3V * \frac{R_1 + 1000\Omega}{1000\Omega}$$

Inverting the formula, I obtain:

$$R_1 = \frac{V_p}{3.3V} * 1000\Omega - 1000\Omega$$

With V_p being the target voltage registered by the probe.

Using this formula, I created the following table:

3.3	3	2.481	1.98	1.65	1.416	1.1	0.66	0.3
0	100	330	660	1000	1330	2000	4000	10000

Detection

First, to be able detect which one of the "buttons" Ire pressed the module must subscribe to the ADC and set up the pin, memory, and interrupt. This means I'll have a function that is called every time the voltage is converted by the ADC.

Using this I can detect the voltage and return the sensed "button". Problem with this is the variability of the recorded input voltage, to avoid this I choose to evaluate the voltage only after sampling it NUM_SAMPLES times and calculate the mean on it.
The classification is then based on a KNN (with $K = 1$) using the calculated mean.

The result is then passed to the main using a pointer to a variable given during set-up.

Testing

To test this module, I wrote a program that prints the sensed "button".

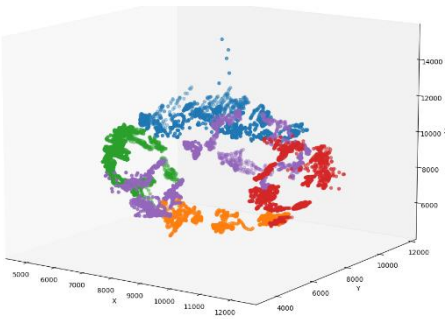
Cables

This module sets up the HW to be able to detect when a cable is disconnected. To communicate with the main program, it modifies an array passed to the module during setup.

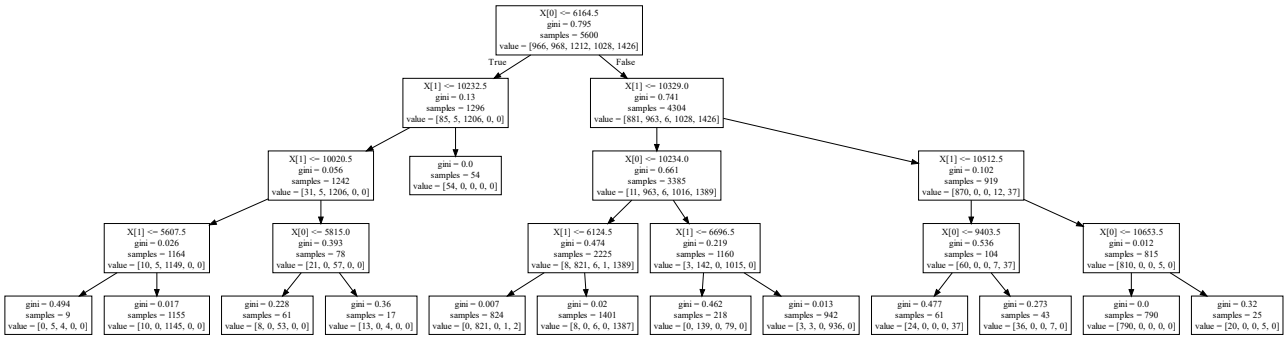
It sets up the pins as input with pull up resistor and registers their IRQ to the same function that checks the flags and modifies the shared array accordingly.

Accelerometer

It's always active, it decides who is the active module based on the X and Z acceleration. To do so a decision tree was created using data collected directly from the sensor:



To ease the implementation the depth of the tree was limited arbitrarily to 4. After training the following was obtained:



The nodes with an impurity higher than 0.45 were considered not reliable and in the implementation the previous prediction is returned.

This module acts as a scheduler, but the pre-emption is implemented inside the modules.

Joystick

Used in the Not not game, this module manages the booster pack joystick and detects one of the four states: U (up), D (down), L (left) and R (right).

To do so it maps the joystick voltages to the range 0.0 – 1.0 and subtract 0.5 from it, due to a shift in the X values I had to correct the voltage by subtracting 300 and dividing it by 16083 instead of 16383. This allows us to think of it as a circle of diameter 1 with its center in (0,0).

I then decided to ignore every input that doesn't extend the joystick more than 75% from the centre of the circle, in order to do so I just had to compute the distance using $x^2 + y^2$ and testing if it was greater than 0.33^2.

I now have a crown of valid inputs, and I want to subdivide it in 4 regions. To do so I have to test for $x > 0.30 \rightarrow R$ or $x < -0.30 \rightarrow L$ or $y > 0.3 \rightarrow U$ or $y < -0.3 \rightarrow D$.

Testing

This module was tested with a program that prints the detected input. It was also tested with "*sonda*" to ensure multiple modules that needs the ADC would work together.

Buzzer

This module uses the PWM to play a simple 8-bit square wave style music (taken from https://github.com/robsoncouto/arduino-songs/blob/master/bloodytears/bloodytears.ino). To archive it an array with the frequencies to play and another with the time unit to play each for is defined, the length is defined in constants.h as NUM_NOTES.
To help defining the music an header file containing all the notes and their corresponding frequency was defined. A single time unit is calculated using the TEMPO and GRID constants defined in constants.h using the following: $60000/(TEMPO * GRID)$ ms, this is then used to set up TIMER_A2.

This module is also used from the Control module to play some chimes as feedbacks for the user inputs. To do it the function $sendBuzzer(int\ freq[], int\ size)$ is called with an array containing the frequencies in Hz to be played and its size. After calling it the music will stop and each frequency will then play for 20ms to then resume the music.

To play the requested frequency the following configuration for TIMER_A0 is used:

```
/* Timer_A Compare Configuration Parameter  (PWM) */
Timer_A_CompareModeConfig compareConfig_PWM = {
TIMER_A_CAPTURECOMPARE_REGISTER_4,                  // Use CCR3
        TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,   // Disable CCR interrupt
```

```
        TIMER_A_OUTPUTMODE_TOGGLE_SET,              // Toggle output but
        compareValue                                // compareValue
        };

/* Timer_A Up Configuration Parameter */
Timer_A_UpModeConfig upConfig = {
        TIMER_A_CLOCKSOURCE_SMCLK,              // SMCLK = 48 MHz
        TIMER_A_CLOCKSOURCE_DIVIDER_12,         // SMCLK/12 = 4 MHz
        timerPeriod,                            //timerPeriod
        TIMER_A_TAIE_INTERRUPT_DISABLE,         // Disable Timer interrupt
        TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE,    // Disable CCR0 interrupt
        TIMER_A_DO_CLEAR                        // Clear value
        };
```

With:

$$upConfig.timerPeriod = \frac{4000000}{notesMusic[currentNote]}$$

$$compareConfig\_PWM.compareValue = \frac{upConfig.timerPeriod}{2}$$

Testing

Since this was a late addition to the project it was tested directly in the full implementation.

Others

Game parser

This module is used to translate the data transmitted by the raspberry pi about the game in JSON to the following data structures:

Testing

Tested locally using the “testing files\test_game_parse.c” program that sends a static JSON to parse.

Games

Control

This module manages the general game status: remaining time and number of errors.

It uses the LCD modules to work. The former one is set up inside the module. It also need to set up the pins for the error leds and

Since this is the only module that access the LCD, it also must write the serial number during the set up.

The module defines the function $addMistakeControl()$ that adds one error, turns on one error LED and speeds up the countdown.

Testing

To test this module a simple program with a fixed time to countdown from was used.

Generic Game

All the games follow the same fundamental architecture:

$$
\begin{aligned}
&void\ setupG(\{\\
&\quad //\ setups\ HW\ dependencies\\
&\quad //\ setups\ internal\ libraries\\
&\}\\
&void\ GameG\{\\
&\ //resets\ necessary\ variables\\
&\ //enables\ HW\ dependencies\ interrupts\\
&\ while(*\ currGame == G\ \&\&*\ time > 0)\{\\
&\quad //game\ loop\ and\ logic\\
&\ \}\\
&\ //disable\ HW\\
&\}
\end{aligned}
$$

Each will then implement each step accordingly.