ECE3073 Project Technical Report

Image Processing Techniques and Optimizations Based on RTOS

Date prepared: [24/05/2024]

Table of Contents

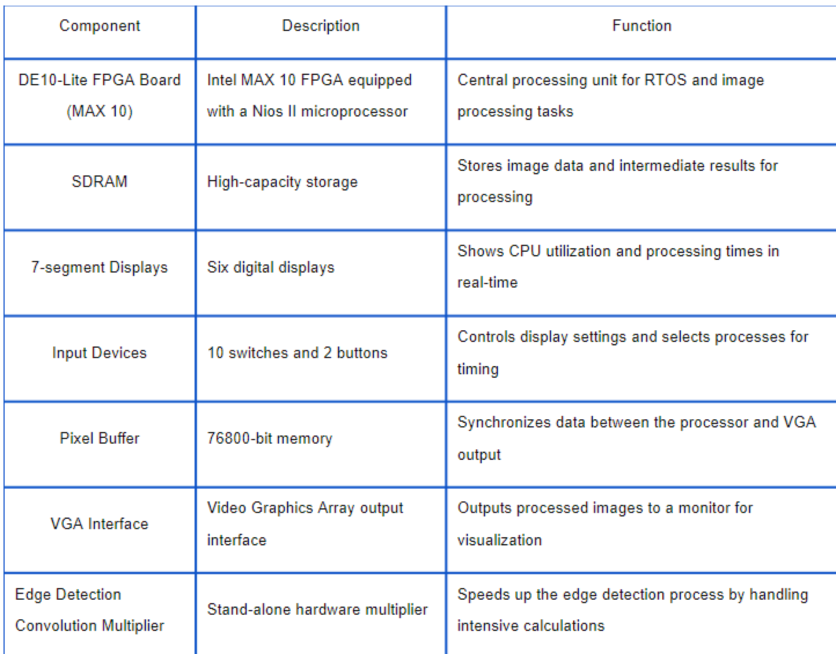
Introduction

This report outlines the development and optimization of a proof-of-concept computer system designed to display and process images through the use of a Real-Time Operating System (RTOS). The system is designed to display and perform calculations on a 160 by 120 grayscale image, where each pixel is represented by 4 bits. By integrating RTOS, the project focuses on improving the real-time performance capabilities and efficiency of both hardware and software components, specifically in the edge detection in an image. The discussions include setting up RTOS in an Eclipse environment, applying image processing techniques such as image flipping, blurring and edge detection, and implementing optimizations to boost throughput and performance. The goal is to demonstrate a practical application of RTOS in complex image processing and display tasks within an embedded systems framework.

System Components and Design

This project integrates various hardware and software elements to build a robust image processing system utilising a Real-Time Operating System (RTOS). Below is an overview of the key components:

Hardware Components



Component	Description	Function
DE10-Lite FPGA Board (MAX 10)	Intel MAX 10 FPGA equipped with a Nios II microprocessor	Central processing unit for RTOS and image processing tasks
SDRAM	High-capacity storage	Stores image data and intermediate results for processing
7-segment Displays	Six digital displays	Shows CPU utilization and processing times in real-time
Input Devices	10 switches and 2 buttons	Controls display settings and selects processes for timing
Pixel Buffer	76800-bit memory	Synchronizes data between the processor and VGA output
VGA Interface	Video Graphics Array output interface	Outputs processed images to a monitor for visualization
Edge Detection Convolution Multiplier	Stand-alone hardware multiplier	Speeds up the edge detection process by handling intensive calculations

Software Components

RTOS

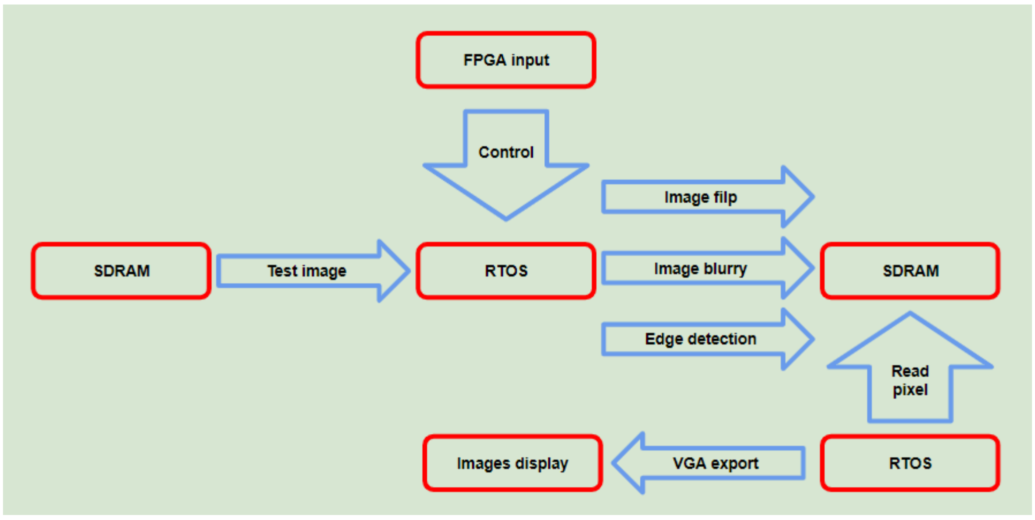
The RTOS manages task scheduling, prioritisation, and resource allocation, ensuring that image processing and display tasks are completed efficiently and in real time. The details for how tasks are designed and how they function are included in the next section.

Image Processing Algorithms

Key algorithms implemented include image flipping, blurring and edge detection, utilising convolution techniques optimised for performance on the FPGA.

Overview of Integration

The system initiates with the Nios II microprocessor booting up the RTOS on the DE10-Lite FPGA board. Upon startup, the RTOS configures tasks for mode switching, Interrupt Service Routines, image processing and display and benchmarking. Images are written into the SDRAM via the DE-10 Control Panel. The Nios II processor, managed by the RTOS, retrieves these images and applies image processing algorithms like blurring and edge detection. Processed images are then selected by the input interface and sent to a pixel buffer which is read by a VGA interface for display. User inputs through FPGA board controls allow for real-time adjustments and switching between display modes, facilitating dynamic interaction and visualisation of the image processing results. Two modes are available to the user. The user can choose to display a single chosen image, or 4 chosen images after downscaling on each of the 4 quadrants of a display. In the process, the time taken for each of the image processing algorithms are recorded and displayed on the 8 segment displays on the DE-10.



Development and Testing Methodology

Task Design

The development started by planning out all of the components of the software system. It is noted that the system required several distinct functions, where each can be allocated to an individual task. Improvements on the image processing speeds are done sequentially, where every major improvement, as measured by the benchmarking task, is documented.

Here, the system contained the following components:

1. ISR and Handlers: The system uses interrupts for switches and keys, which are handled by separate tasks using signalling semaphores. This includes the main task, which is used to handle the switch modes, and key mode handler task.
2. Display: The software initialises 4 tasks, each responsible for displaying to a quadrant of the display, after downscaling a 160 by 120 image. If single image mode is active, then the top left display task handles the display of the entire screen, and the other 3 tasks sleep. The screen display is refreshed every 250 ms.
3. Image Processing: A task that runs all image processing algorithms every 500 ms. The continuous processing of images intends to simulate the continuous processing of a video stream.
4. Benchmark: A task that reads and displays the time taken to run each of the image processing tasks and cpu utilisation.

Priority

In the creation of the tasks, priorities are given in order of importance. The interrupt mode handlers are given the highest priority, followed by the image processing task, display tasks and the benchmarking task. This ensures that the image processing runs mostly uninterrupted, and displays only ever giving correct inputs to the pixel buffer. All Mutually Exclusive semaphores, as introduced in the next sections, occupy priorities from 1 to 6 as required by the mutex to function.

Semaphores and RTOS system

During the design process, it was found that multiple tasks require access to shared resources such as images in the SDRAM. Mutually exclusive (Mutex) and signalling semaphores are used to guard these resources and prevent unwanted simultaneous accesses.

Specifically, the system used the following semaphores:

1. Mode Mutex: This is a semaphore used for guarding the display modes, including whether quad-image mode is on, and what image(s) are displayed. Effectively all tasks except image processing use this mutex.
2. Image Mutex: This is a collection of 4 Mutexes, where each mutex guards a single image inside the SDRAM. The design introduced this to prevent simultaneous reading and writing of the same image in SDRAM, ensuring that correct pixel values are delivered to the screen.

3. Time Mutex: This mutex guards the global array of time taken to perform each image processing algorithm, which is accessed by the image processing task and the benchmarking task.
4. Switch Signalling Semaphore: This semaphore is used to signal the switch mode handling task, such that the interrupt does not need to wait for any mutex and that the handling task is not constantly polling the switches.
5. Key Signalling Semaphore: This semaphore is used to signal the key mode handling task, such that the interrupt does not need to wait for any mutex and that the handling task is not constantly polling the switches.

User interface and display

The system accepts user input through switches and keys. Note that these 2 user inputs are handled through Hardware Interrupts, and are integrated into the RTOS system by 2 separate tasks dedicated to servicing each interrupt. Two display modes are available to the user: the single image mode where the user sees a full sized image of choosing, and quad image mode where the user can pick 4 images to be downscaled and displayed on each of the 4 quadrants of a display. It is worth noting that the 160 by 120 pixel image is downscaled to 80 by 60 when displaying in quad image mode. The 4 leftmost displays show the time taken to run various image processing algorithms. The 2 rightmost 7-segment displays show the cpu usage, refreshing every 250 ms. This user input scheme is summarised in the table below.

Input Device	Single Image Mode	Quad Image Mode
Key 0	None	Switch to single image mode
Key 1	Switch to quad image mode	None
Switch 0 and 1	Pick which image to display	Pick which image to display for top left quadrant
Switch 2 and 3	None	Pick which image to display for top right quadrant
Switch 4 and 5	None	Pick which image to display for bottom left quadrant
Switch 6 and 7	None	Pick which image to display for bottom right quadrant
Switch 8 and 9	00 -> display flip time 01 -> display blur time 10 -> display edge time 11 -> combined time	Same as single image mode

* For switches: 0 indicates the switch being off, and 1 is when the switch is on. 00 displays the normal image, 01 the flipped image, 10 the blurring image and 11 the edge detection image.

Image Processing Algorithms and Optimizations

This project demonstrates 3 distinct image processing algorithms: image flipping, blurring and Sobel Edge detection. An iterative methodology is taken to improve the runtime of the algorithms step by step, with a specific focus on the edge detection algorithm. With each step, the runtime for each algorithm is recorded. 3 rounds of optimization took place.

Removing Function Calls

After writing a functional implementation of each algorithm, the first optimization involves replacing small helper function calls to instead use macros. Examples of these function calls include aliases for reading and writing pixels, as well as calculating the offset address via the x and y coordinate of a pixel on an image. This simple change removed the frequent context switches for every small task that can be accomplished in a single line, resulting in approximately ⅓ speed up across all three algorithms.

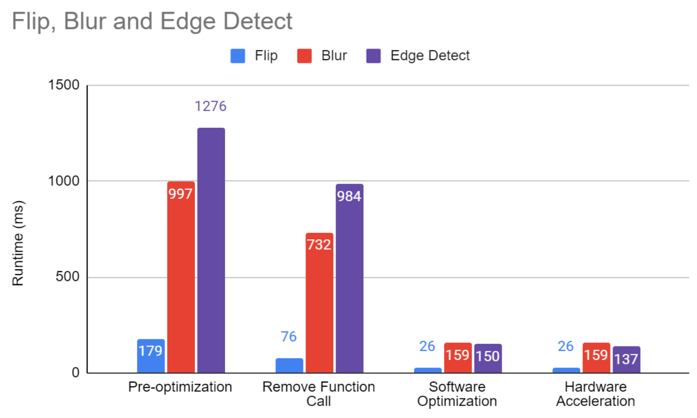
Software Optimization

Following that, overhaul of each of the algorithms is done via software. Each algorithm implementation is analysed and parts of redundant calculations are removed. For example, it is noticed that edge detection was reading each 9 pixels in a kernel region twice. Here, the number of reading operations from SDRAM for each pixel calculation is reduced from 18 to 8 as readings are done once and stored in a variable. Additionally, techniques such as unrolling loops are used to remove overhead in maintaining a counter. Some small if-else statements for checking whether a pixel is on the edge is isolated from the main for loop. This is done to remove branching in the loop, which can be costly if branch prediction fails in the CPU processing cycle. Flipping the image in both directions is now done in one cycle, instead of flipping in the x-direction first and the y-direction second. This optimization is most relevant to blurring and edge detection. This effectively reduced the processing time by over 70%.

Hardware Acceleration (for edge detection)

In the latest development version, a hardware multiplier is incorporated to facilitate convolution in the edge detection algorithm. Instead of multiplying the pixel values with the kernel value in software, all 8 pixels are sent to the multiplier and the result is read. It replaces software multiplication and addition with a single read and a single write instruction, improving the runtime.

Results



As seen in the difference in the first version, replacing function calls by macros dramatically improved the runtime. From the pre-optimization program, the image display became significantly more responsive, as the switch between display modes are serviced sooner. The software optimizations after removing significant amounts of redundant work. Hardware acceleration, being only implemented in the edge detection algorithm, resulted in smaller gain than expected, delivering approximately 8.67% reduction in the runtime.

The CPU utilisation reduced from 71% to 18% after the first optimization, and remained the same for subsequent steps.

Discussion

The project found a significant impact of software design to the runtime of programs. This is expected, as compared to the pre-optimization code, the newest version removed most of the redundant calculations which accounted for a large portion of the runtime before any optimization was made.

On the contrary, hardware acceleration seemed to have less than expected effects, as the runtime for edge detection only reduced by less than 10%. Compared to multiple instructions on the software side, the hardware multiplication is almost instant. Theoretically, 6 add and 2 multiply instructions are replaced by 1 read and 1 write instruction. This should reduce the effort within the loop by 75%. It is suspected here that the large discrepancy between the theory and the practical observations can be attributed to the overhead from other instructions that run in addition to the improved section, as well as the potential memory speed bottlenecks. Since hardware acceleration here gives little improvement at a greater cost of production, this technique is not recommended unless the 8.6% runtime improvement is critical to operation.

Recommendations

One major improvement that can be made is to utilise previously read data for calculations. For 2 adjacent pixels, currently 8 pixels are read for each pixel calculation. In these 16 reads, 4 pixels are read twice. If we can do edge detection for 2 adjacent pixels together, then only 16-4*2 = 8 pixels reads are needed. The more pixels we can do together, the greater the amount of savings we can achieve. By designing a better algorithm in performing convolution calculations, there is potential for runtime that is more than twice faster than the current version.

The current hardware multiplier is geared specifically towards edge detection. We can make use of this expensive hardware by turning it general for all convolution operations, such that image blurring or image downscaling can take advantage of it.

Conclusion

References

List any sources, references, or literature cited throughout the report. Ensure proper citation formatting according to APA guidelines or industry standards.

Appendix

Verilog Modules

Top Level

```
// *********************************
//
// Top level module for the integration of the VGA controller, pixel buffer and NIOS II processor.
//
// Author: Jiayi Gu  | ID: 33114404
//         Yulan Sui | ID: 31973647
// Last Edited: 13/04/2024
//
// *********************************


module integration(
    input CLOCK_50,
    input [9:0] SW,
    output [7:0] HEX0,
    output [7:0] HEX1,
    output [7:0] HEX2,
    output [7:0] HEX3,
    output [7:0] HEX4,
    output [7:0] HEX5,
    input [1:0] KEY,
    output [9:0] LEDR,

    // VGA
    output [3:0] VGA_R,
    output [3:0] VGA_G,
    output [3:0] VGA_B,
    output VGA_HS,
    output VGA_VS,

    // dram
    output DRAM_CLK,
    output [12:0] DRAM_ADDR,
    output [1:0] DRAM_BA,
    output DRAM_CAS_N,
    output DRAM_CKE,
    output DRAM_CS_N,
inout [15:0] DRAM_DQ,
output DRAM_RAS_N,
output DRAM_WE_N,
output DRAM_UDQM,
output DRAM_LDQM
);

// nios
milestone1 u0 (
.clk_clk            (CLOCK_50),            //         clk.clk
// SDRAM
.sdram_addr  (DRAM_ADDR),                  // sdram.addr
.sdram_ba    (DRAM_BA),                    //      .ba
.sdram_cas_n (DRAM_CAS_N),                 //      .cas_n
.sdram_cke   (DRAM_CKE),                   //      .cke
.sdram_cs_n  (DRAM_CS_N),                  //      .cs_n
.sdram_dq    (DRAM_DQ),                    //      .dq
.sdram_ras_n (DRAM_RAS_N),                 //      .ras_n
.sdram_we_n  (DRAM_WE_N),                  //      .we_n
.sdram_dqm   ({DRAM_UDQM, DRAM_LDQM}),     //      .dqm

.buttons_export     (KEY[1:0]),            //     buttons.export
.switches_export    (SW[9:0]),             //    switches.export
.red_leds_export    (LEDR[9:0]),           //    red_leds.export
.buffer_addr_export (write_addr),          // buffer_addr.export
.buffer_data_export (pixel_in),            // buffer_data.export
.hex_10_export      ({HEX1, HEX0}),
.hex_5432_export    ({HEX5, HEX4, HEX3, HEX2}),
.mult_in_export     (mult_in),
.mult_out_export    (mult_out),
);

wire [31:0] mult_in;
wire [7:0] mult_out;
mult_kernel mult_kernel_inst(
.in(mult_in),
.out(mult_out)
);

sdram_clk sdram_clk_inst (
    .inclk0 (CLOCK_50),
.c0 (DRAM_CLK)
);



// VGA clock
wire vga_clock;
vga_clk vga_clk_inst(
.inclk0(CLOCK_50),
.c0(vga_clock)
);

wire [3:0] pixel_in;

// for debugging
// assign pixel_in = SW[3:0];

wire [14:0] read_addr;
wire [14:0] write_addr;
wire [3:0] pixel_out;
pixel_buffer pixel_buffer_inst(
.clock(CLOCK_50),
.data(pixel_in),
.rdaddress(read_addr),
.wraddress(write_addr),
.wren(vga_clock),
.q(pixel_out)
);

// VGA controller
vga_controller vga_controller_inst(
.VGA_DATA(pixel_out),
.VGA_CLK(vga_clock),
.VGA_ADDR(read_addr),
.VGA_R(VGA_R),
.VGA_G(VGA_G),
.VGA_B(VGA_B),
.VGA_HS(VGA_HS),
.VGA_VS(VGA_VS)
);
endmodule
```

```
module mult_kernel(
input [31:0] in,
output reg [7:0] out // [x, y]
);
wire signed [4:0] n00, n01, n02, n10, n12, n20, n21, n22;
assign n00 = {1'd0, in[3:0]};
assign n01 = {1'd0, in[7:4]};
assign n02 = {1'd0, in[11:8]};
assign n10 = {1'd0, in[15:12]};
assign n12 = {1'd0, in[19:16]};
assign n20 = {1'd0, in[23:20]};
assign n21 = {1'd0, in[27:24]};
assign n22 = {1'd0, in[31:28]};

reg signed [7:0] gradX;
reg signed [7:0] gradY;
wire [6:0] absX;
wire [6:0] absY;
abs abs_x_inst(
.in(gradX),
.out(absX)
);
abs abs_y_inst(
.in(gradY),
.out(absY)
);
always @(*) begin
gradX <= n00 + 2*n10 + n20 - n02 - 2*n12 - n22;
gradY <= n00 + 2*n01 + n02 - n20 - 2*n21 - n22;
if (absX + absY > 8'd7) begin
out <= 8'd7;
end else begin
out <= absX + absY;
end
end
endmodule

module abs(
input signed [6:0] in,
output reg signed [6:0] out
);
```

```
always @(*) begin
    if (in < 0) begin
        out <= -in;
    end else begin
        out <= in;
    end
end
endmodule
```

VGA Controller

```
// **********************************
//
// VGA Controller Module
// Made for students to use during the ECE3073 Project
//
// Author: Ben Saunders
// Last Edited: 25/02/2024
//
// Copyright © 2024 Copyright, Monash University
//
// **********************************


module vga_controller (
    input [3:0] VGA_DATA,
    input VGA_CLK,

    output [18:0] VGA_ADDR, // This addres may vary, having it longer that what you use wont be an issue
    output [3:0] VGA_R,
    output [3:0] VGA_G,
    output [3:0] VGA_B,
    output VGA_HS,
    output VGA_VS

);

// Back and front does not seem to be used

// Maybe students should fill this out
parameter H_SYNC = 96;
parameter H_BACK = 48;
parameter H_FRONT = 16;
parameter H_CYCLE = 800;

parameter V_SYNC = 2;
parameter V_BACK = 33;
parameter V_FRONT = 10;
parameter V_CYCLE = 525;

reg [18:0] H_ADDR, V_ADDR;

always @(posedge VGA_CLK) // horizontal counter
begin
if (H_ADDR < H_CYCLE)
H_ADDR <= H_ADDR + 1;
else
H_ADDR <= 0;
end

always @(posedge VGA_CLK) // vertical counter
begin
if (H_ADDR == H_CYCLE)
begin
if (V_ADDR < V_CYCLE)
V_ADDR <= V_ADDR + 1;
else
V_ADDR <= 0;
end
end

assign VGA_HS = (H_ADDR < H_SYNC) ? 1:0; // hsync high for 96 counts
assign VGA_VS = (V_ADDR < V_SYNC) ? 1:0; // vsync high for 2 counts

assign pixelValid = (H_ADDR >= 144 && H_ADDR < 784) && (V_ADDR >= 35 && V_ADDR < 515);
assign VGA_R = pixelValid ? VGA_DATA : 4'd0;
assign VGA_G = pixelValid ? VGA_DATA : 4'd0;
assign VGA_B = pixelValid ? VGA_DATA : 4'd0;

wire [18:0] H_MEM_ADDR, V_MEM_ADDR;

assign H_MEM_ADDR = H_ADDR - 144;
assign V_MEM_ADDR = V_ADDR - 35;
```

```
// Scale the addresses to 160X120 instead of 640x480.
assign VGA_ADDR = ((H_MEM_ADDR>>2) + ((V_MEM_ADDR>>2) * 160));


endmodule
```

Software Components

Benchmarking

Header

```
/**
 *   Name: benchmark.h
 *   Description:
 *       This file contains the declaration for benchmark task and related functions
 *   Last edited: 17/05/2024
 *   Author: Jiayi Gu  | ID: 33114404
 *           Yulan Sui | ID: 31973647
 */
#ifndef BENCHMARK_H_
#define BENCHMARK_H_

/**
 * @brief benchmark task, used to display cpu usage and image processing time
 * @param pdata
 */
extern void benchmark(void *pdata);

/**
 * @brief write a number to the 7 segment display HEX0 and HEX1
 * @param num the number to write
 */
void write_to_hex_10(int num);

/**
 * @brief write a number to the 7 segment display HEX5 to HEX2
 * @param num the number to write
 */
void write_to_hex_5432(int num);

/**
```

```
* @brief convert a number to a 7 segment display value
* @param num the number to convert, in range 0-9
* @return the 7 segment display value
*/
int num_to_7seg(int num);


#endif /* BENCHMARK_H_ */
```

Source

```
/**
*   Name: benchmark.h
*   Description:
*       This file contains the definition for benchmark task and related functions
*           All function comments are in benchmark.h
*   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/
#include "display_tasks.h"
#include "benchmark.h"
#include "global_var.h"
#include "system.h"
#include "includes.h"
#include "io.h"
#include <stdio.h>

void benchmark(void *pdata) {
INT8U err;
while (1) {
// update the cpu usage
write_to_hex_10(OSCPUUsage);
// read switch and update time
OSMutexPend(mode_mutex, 0, &err);
OSMutexPend(time_mutex, 0, &err);
write_to_hex_5432(task_time[hex_time]);
OSMutexPost(time_mutex);
OSMutexPost(mode_mutex);
```

```
OSTimeDlyHMSM(0, 0, 0, DISPLAY_INTERVAL);
}
}

void write_to_hex_10(int num) {
int tens = num / 10;
IOWR_32DIRECT(HEX_10_BASE, 0, (num_to_7seg(tens)<<8)|num_to_7seg(num-tens*10));
}

void write_to_hex_5432(int num) {
int thousands = num / 1000;
int hundreds = (num - thousands*1000) / 100;
int tens = (num - thousands*1000 - hundreds*100) / 10;
int ones = num - thousands*1000 - hundreds*100 - tens*10;
IOWR_32DIRECT(HEX_5432_BASE, 0,
(num_to_7seg(thousands)<<24)|(num_to_7seg(hundreds)<<16)|(num_to_7seg(tens)<<8)|num_to_7seg(ones));
}


int num_to_7seg(int num){
switch(num){
case 0:
return 0b11000000;
case 1:
return 0b11111001;
case 2:
return 0b10100100;
case 3:
return 0b10110000;
case 4:
return 0b10011001;
case 5:
return 0b10010010;
case 6:
return 0b10000010;
case 7:
return 0b11111000;
case 8:
return 0b10000000;
case 9:
return 0b10010000;
```

```
default:
return 0b11111111;
}
}
```

Display Tasks

Header

```
/**
*   Name: display_tasks.h
*   Description:
*       This file contains the declaration for 4 display tasks and 1 image processing task
*   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/

#ifndef DISPLAY_TASKS_H_
#define DISPLAY_TASKS_H_

#define DISPLAY_INTERVAL 250
#define PROCESS_INTERVAL 500

#include "global_var.h"

/**
* @brief display the top left image, or the whole image in single image mode
*/
void tl_display(void *pdata);
/**
* @brief display the top right image
*/
void tr_display(void *pdata);
/**
* @brief display the bottom left image
*/
void bl_display(void *pdata);
/**
* @brief display the bottom right image
*/
void br_display(void *pdata);
/**
```

```
* @brief process the image, flip, blur and edge detection
*/
void process_image(void *pdata);

#endif /* DISPLAY_TASKS_H_ */
```

Source

```
/**
*   Name: display_tasks.h
*   Description:
*       This file contains the definition for 4 display tasks and 1 image processing task
*   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/
#include "display_tasks.h"
#include "image_processing.h"
#include <stdio.h>

void tl_display(void *pdata) {
int starting_x = 0, starting_y = 0;
INT8U err;
while (1) {
OSMutexPend(mode_mutex, 0, &err);

OSMutexPend(image_mutex[image_displayed[TL]], 0, &err);

// handle both quad and single, so that other tasks don't care about single mode
display_image(image_displayed[TL], mode, starting_x, starting_y);
OSMutexPost(image_mutex[image_displayed[TL]]);
OSMutexPost(mode_mutex);
OSTimeDlyHMSM(0, 0, 0, DISPLAY_INTERVAL);
}
}

void tr_display(void *pdata) {
int starting_x = 80, starting_y = 0;
INT8U err;
```

```
while (1) {
OSMutexPend(mode_mutex, 0, &err);

if (mode == QUAD) {
OSMutexPend(image_mutex[image_displayed[TR]], 0, &err);

display_image(image_displayed[TR], mode, starting_x, starting_y);
OSMutexPost(image_mutex[image_displayed[TR]]);
}
OSMutexPost(mode_mutex);
OSTimeDlyHMSM(0, 0, 0, DISPLAY_INTERVAL);
}
}



void bl_display(void *pdata) {
int starting_x = 0, starting_y = 60;
INT8U err;



while (1) {
OSMutexPend(mode_mutex, 0, &err);

if (mode == QUAD) {
OSMutexPend(image_mutex[image_displayed[BL]], 0, &err);

display_image(image_displayed[BL], mode, starting_x, starting_y);
OSMutexPost(image_mutex[image_displayed[BL]]);
}
OSMutexPost(mode_mutex);
OSTimeDlyHMSM(0, 0, 0, DISPLAY_INTERVAL);
}
}

void br_display(void *pdata) {
int starting_x = 80, starting_y = 60;
INT8U err;

while (1) {
OSMutexPend(mode_mutex, 0, &err);
```

```
if (mode == QUAD) {
OSMutexPend(image_mutex[image_displayed[BR]], 0, &err);

display_image(image_displayed[BR], mode, starting_x, starting_y);
OSMutexPost(image_mutex[image_displayed[BR]]);
}
OSMutexPost(mode_mutex);
OSTimeDlyHMSM(0, 0, 0, DISPLAY_INTERVAL);
}
}

void process_image(void *pdata) {
INT8U err;
int time_diff = 0;
int time_sum = 0;
while (1) {

OSMutexPend(time_mutex, 0, &err);

OSMutexPend(image_mutex[NORM], 0, &err);


OSMutexPend(image_mutex[FLIPPED], 0, &err);
time_sum = 0;
time_diff = OSTimeGet();
flip_image();
time_diff = OSTimeGet() - time_diff;
time_sum += time_diff;
task_time[FLIP] = time_diff;
OSMutexPost(image_mutex[FLIPPED]);

OSMutexPend(image_mutex[BLURRED], 0, &err);
time_diff = OSTimeGet();
blur_image();
time_diff = OSTimeGet() - time_diff;
time_sum += time_diff;
task_time[BLUR] = time_diff;
OSMutexPost(image_mutex[BLURRED]);
```

```
OSMutexPend(image_mutex[EDGE], 0, &err);
time_diff = OSTimeGet();
edge_image();
time_diff = OSTimeGet() - time_diff;
time_sum += time_diff;
task_time[EDGE_DETECT] = time_diff;
OSMutexPost(image_mutex[EDGE]);

task_time[COMBINED] = time_sum;

OSMutexPost(time_mutex);
OSMutexPost(image_mutex[NORM]);

OSTimeDlyHMSM(0, 0, 0, PROCESS_INTERVAL);
}
}
```

Global Variables

Header

```
/**
 *   Name: global_var.h
 *   Description:
 *       This file contains the declaration of all global variables, enums and semaphores
 *   Last edited: 17/05/2024
 *   Author: Jiayi Gu  | ID: 33114404
 *           Yulan Sui | ID: 31973647
 */

#ifndef GLOBAL_VAR_H_
#define GLOBAL_VAR_H_
#include "includes.h"

#define TASK_STACKSIZE 512

/* Definition of Sem/Mutex Prios */
#define MODE_MUTEX_PRIORITY 1
#define TIME_MUTEX_PRIORITY 2
#define IMAGE_MUTEX_PRIORITY 3  // 3~6
```

```
/* Definition of Task Priorities */
#define MAIN_PRIORITY 10
#define TL_PRIORITY 12
#define TR_PRIORITY 13
#define BL_PRIORITY 14
#define BR_PRIORITY 15
#define PROCESS_IMAGE_PRIORITY 11
#define BENCHMARK_PRIORITY 16
#define KEY_PRIORITY 17

/* Definition of Task Stacks */
extern OS_STK main_stk[TASK_STACKSIZE];
extern OS_STK tl_display_stk[TASK_STACKSIZE];
extern OS_STK tr_display_stk[TASK_STACKSIZE];
extern OS_STK bl_display_stk[TASK_STACKSIZE];
extern OS_STK br_display_stk[TASK_STACKSIZE];
extern OS_STK process_image_stk[TASK_STACKSIZE];
extern OS_STK benchmark_stk[TASK_STACKSIZE];
extern OS_STK key_stk[TASK_STACKSIZE];

// enums
typedef enum {
    NORM = 0,
    FLIPPED = 1,
    BLURRED = 2,
    EDGE = 3
} image_type;

typedef enum {
    SINGLE,
    QUAD
} display_mode;

typedef enum {
    TL = 0,
    TR = 1,
    BL = 2,
    BR = 3
} image_location;

typedef enum {
```

```c
FLIP = 0,
BLUR = 1,
EDGE_DETECT = 2,
COMBINED = 3
} display_time;

display_mode mode;
extern display_time hex_time;
extern image_type image_displayed[4];
extern OS_EVENT *mode_mutex;

extern OS_EVENT *image_mutex[4];

extern int switch_state;
extern OS_EVENT *switch_signal;

extern int key;
extern OS_EVENT *key_siganl;

extern int task_time[4];
extern OS_EVENT *time_mutex;


#endif /* GLOBAL_VAR_H_ */
```

Source

```c
/**
*   Name: global_var.c
*   Description:
*       This file contains the definition of all global variables, enums and semaphores
*   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/
#include "global_var.h"
#include "includes.h"

// Task Stacks
OS_STK main_stk[TASK_STACKSIZE];
OS_STK tl_display_stk[TASK_STACKSIZE];
OS_STK tr_display_stk[TASK_STACKSIZE];
OS_STK bl_display_stk[TASK_STACKSIZE];
OS_STK br_display_stk[TASK_STACKSIZE];
OS_STK process_image_stk[TASK_STACKSIZE];
OS_STK benchmark_stk[TASK_STACKSIZE];
OS_STK key_stk[TASK_STACKSIZE];

// Semaphores
display_mode mode = QUAD;
display_time hex_time = 0;
image_type image_displayed[4] = {0};
OS_EVENT *mode_mutex;

OS_EVENT* image_mutex[4];

int switch_state = 0;
OS_EVENT *switch_signal;

int key = 0;
OS_EVENT *key_siganl;

int task_time[4] = {0};
OS_EVENT *time_mutex;

// acquire order
// Any signalling semaphore
// 1. mode_mutex
// 2. time_mutex
// 3. image_mutex
```

Image Processing

Header

```
/**
 *   Name: image_processing.h
 *   Description:
 *       This file contains the declaration of all image processing functions, including 3 versions of edge detection, blur and flip
 *   Last edited: 17/05/2024
 *   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/

#ifndef IMAGE_PROCESSING_H_
#define IMAGE_PROCESSING_H_

#include "io.h"

// define functions
#define read_pixel(address, offset) (IORD(SDRAM_BASE, (address+offset))>>24)
#define write_pixel(address, offset, pixel) (IOWR(SDRAM_BASE, (address + offset), (pixel)<<24))
#define flatten_address(x, y) (((y) * MAX_X) + x)

// Constants for screen size
#define MAX_X 160
#define MAX_Y 120
#define PIXEL_COUNT 19200

// image offsets
#define NORM_OFFSET 0
#define FLIPPED_OFFSET 19200
#define BLURRED_OFFSET 38400
#define EDGE_OFFSET 57600

#define MAX_DOWNSCALED_X 80
#define MAX_DOWNSCALED_Y 60
#define DOWNSCALED_PIXEL_COUNT 4800

#define MAX_PIXEL_VALUE 15

void display_image(int type, int mode, int starting_x, int starting_y);

void flip_image(void);

void blur_image(void);

void edge_image(void);

int applySobelKernel(int x, int y, int kernel[3][3]);

void sobelEdgeDetection_v1();
void sobelEdgeDetection_v2();


#endif /* IMAGE_PROCESSING_H_ */
```

Source

```
/**
 *   Name: image_processing.c
 *   Description:
 *       This file contains the definitions of all image processing functions, including 3 versions of edge detection, blur and flip
 *   Last edited: 17/05/2024
 *   Author: Jiayi Gu  | ID: 33114404
 *           Yulan Sui | ID: 31973647
 */

#include "image_processing.h"
#include "global_var.h"
#include <stdlib.h>
#include "system.h"

void flip_image(void) {
int x, y;
for (x = 0; x < MAX_X; x++) {
  for (y = 0; y < MAX_Y; y++) {
write_pixel(flatten_address(x, y), FLIPPED_OFFSET, read_pixel(flatten_address(MAX_X - x - 1, MAX_Y - y - 1),
NORM_OFFSET));
    }
  }
}

void blur_image(void) {
int sum, x, y;
// unrolling these loops save 10ms
// process y=0 and y=MAX_Y-1
for (x = 1; x < MAX_X - 1; x++) {
    sum = 0;
sum += read_pixel(flatten_address(x - 1, 0), NORM_OFFSET);
sum += read_pixel(flatten_address(x, 0), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, 0), NORM_OFFSET);
sum += read_pixel(flatten_address(x - 1, 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x, 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, 1), NORM_OFFSET);
write_pixel(flatten_address(x, 0), BLURRED_OFFSET, sum / 6);
sum = 0;
sum += read_pixel(flatten_address(x - 1, MAX_Y - 2), NORM_OFFSET);
sum += read_pixel(flatten_address(x, MAX_Y - 2), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, MAX_Y - 2), NORM_OFFSET);
sum += read_pixel(flatten_address(x - 1, MAX_Y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x, MAX_Y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, MAX_Y - 1), NORM_OFFSET);
write_pixel(flatten_address(x, MAX_Y - 1), BLURRED_OFFSET, sum / 6);
}
// process x=0 and x=MAX_X-1
for (y = 1; y < MAX_Y - 1; y++) {
sum = 0;
sum += read_pixel(flatten_address(0, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(0, y), NORM_OFFSET);
sum += read_pixel(flatten_address(0, y + 1), NORM_OFFSET);
sum += read_pixel(flatten_address(1, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(1, y), NORM_OFFSET);
sum += read_pixel(flatten_address(1, y + 1), NORM_OFFSET);
write_pixel(flatten_address(0, y), BLURRED_OFFSET, sum / 6);
sum = 0;
sum += read_pixel(flatten_address(MAX_X - 2, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(MAX_X - 2, y), NORM_OFFSET);
sum += read_pixel(flatten_address(MAX_X - 2, y + 1), NORM_OFFSET);
sum += read_pixel(flatten_address(MAX_X - 1, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(MAX_X - 1, y), NORM_OFFSET);
sum += read_pixel(flatten_address(MAX_X - 1, y + 1), NORM_OFFSET);
write_pixel(flatten_address(MAX_X - 1, y), BLURRED_OFFSET, sum / 6);
}
// handle the edges
write_pixel(flatten_address(0, 0), BLURRED_OFFSET, (read_pixel(flatten_address(0, 1), NORM_OFFSET) +
read_pixel(flatten_address(1, 0), NORM_OFFSET) + read_pixel(flatten_address(1, 1), NORM_OFFSET)) / 4);
write_pixel(flatten_address(MAX_X - 1, 0), BLURRED_OFFSET, (read_pixel(flatten_address(MAX_X - 1, 1),
NORM_OFFSET) + read_pixel(flatten_address(MAX_X - 2, 0), NORM_OFFSET) + read_pixel(flatten_address(MAX_X - 2, 1),
NORM_OFFSET)) / 4);
write_pixel(flatten_address(0, MAX_Y - 1), BLURRED_OFFSET, (read_pixel(flatten_address(0, MAX_Y - 2),
NORM_OFFSET) + read_pixel(flatten_address(1, MAX_Y - 1), NORM_OFFSET) + read_pixel(flatten_address(1, MAX_Y - 2),
NORM_OFFSET)) / 4);
```

```
write_pixel(flatten_address(MAX_X - 1, MAX_Y - 1), BLURRED_OFFSET, (read_pixel(flatten_address(MAX_X - 1, MAX_Y -
2), NORM_OFFSET) + read_pixel(flatten_address(MAX_X - 2, MAX_Y - 1), NORM_OFFSET) +
read_pixel(flatten_address(MAX_X - 2, MAX_Y - 2), NORM_OFFSET)) / 4);
// process the rest
for (x = 1; x < MAX_X - 1; x++) {
for (y = 1; y < MAX_Y - 1; y++) {
sum = 0;
// unroll the loop, saved 600ms from 1.3s to 0.621s
sum += read_pixel(flatten_address(x - 1, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x - 1, y), NORM_OFFSET);
sum += read_pixel(flatten_address(x - 1, y + 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x, y), NORM_OFFSET);
sum += read_pixel(flatten_address(x, y + 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, y - 1), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, y), NORM_OFFSET);
sum += read_pixel(flatten_address(x + 1, y + 1), NORM_OFFSET);
write_pixel(flatten_address(x, y), BLURRED_OFFSET, sum / 9);
}
}
}

void edge_image(void) {
int gradX, gradY, mult_in;
int threshold = (MAX_PIXEL_VALUE >> 1);
// handle the first and last column
// handle the first and last column
for (int y=0; y<MAX_Y; y++) {
mult_in = 0;
mult_in |= read_pixel(flatten_address(0, y - 1), NORM_OFFSET);
mult_in |= read_pixel(flatten_address(1, y - 1), NORM_OFFSET) << 4;
mult_in |= read_pixel(flatten_address(0, y), NORM_OFFSET) << 8;
mult_in |= read_pixel(flatten_address(1, y), NORM_OFFSET) << 12;
mult_in |= read_pixel(flatten_address(0, y + 1), NORM_OFFSET) << 16;
mult_in |= read_pixel(flatten_address(1, y + 1), NORM_OFFSET) << 20;
IOWR(MULT_IN_BASE, 0, mult_in);
write_pixel(flatten_address(0, y), EDGE_OFFSET, IORD(MULT_OUT_BASE, 0));

mult_in = 0;
mult_in |= read_pixel(flatten_address(MAX_X - 2, y - 1), NORM_OFFSET);
mult_in |= read_pixel(flatten_address(MAX_X - 1, y - 1), NORM_OFFSET) << 4;
mult_in |= read_pixel(flatten_address(MAX_X - 2, y), NORM_OFFSET) << 8;
mult_in |= read_pixel(flatten_address(MAX_X - 1, y), NORM_OFFSET) << 12;
mult_in |= read_pixel(flatten_address(MAX_X - 2, y + 1), NORM_OFFSET) << 16;
mult_in |= read_pixel(flatten_address(MAX_X - 1, y + 1), NORM_OFFSET) << 20;
IOWR(MULT_IN_BASE, 0, mult_in);
write_pixel(flatten_address(MAX_X-1, y), EDGE_OFFSET, IORD(MULT_OUT_BASE, 0));
}

// handle the first and last row
for (int x=0; x<MAX_X; x++) {
mult_in = 0;
mult_in |= read_pixel(flatten_address(x - 1, 0), NORM_OFFSET);
mult_in |= read_pixel(flatten_address(x, 0), NORM_OFFSET) << 4;
mult_in |= read_pixel(flatten_address(x + 1, 0), NORM_OFFSET) << 8;
mult_in |= read_pixel(flatten_address(x - 1, 1), NORM_OFFSET) << 12;
mult_in |= read_pixel(flatten_address(x, 1), NORM_OFFSET) << 16;
mult_in |= read_pixel(flatten_address(x + 1, 1), NORM_OFFSET) << 20;
IOWR(MULT_IN_BASE, 0, mult_in);
write_pixel(flatten_address(x, 0), EDGE_OFFSET, IORD(MULT_OUT_BASE, 0));

mult_in = 0;
mult_in |= read_pixel(flatten_address(x - 1, MAX_Y - 2), NORM_OFFSET);
mult_in |= read_pixel(flatten_address(x, MAX_Y - 2), NORM_OFFSET) << 4;
mult_in |= read_pixel(flatten_address(x + 1, MAX_Y - 2), NORM_OFFSET) << 8;
mult_in |= read_pixel(flatten_address(x - 1, MAX_Y - 1), NORM_OFFSET) << 12;
mult_in |= read_pixel(flatten_address(x, MAX_Y - 1), NORM_OFFSET) << 16;
mult_in |= read_pixel(flatten_address(x + 1, MAX_Y - 1), NORM_OFFSET) << 20;
IOWR(MULT_IN_BASE, 0, mult_in);
write_pixel(flatten_address(x, MAX_Y-1), EDGE_OFFSET, IORD(MULT_OUT_BASE, 0));
}

// handle the rest
for (int y = 1; y < MAX_Y - 1; y++) {
for (int x = 1; x < MAX_X - 1; x++) {
mult_in = 0;
mult_in |= read_pixel(flatten_address(x - 1, y - 1), NORM_OFFSET);
mult_in |= read_pixel(flatten_address(x, y - 1), NORM_OFFSET) << 4;
mult_in |= read_pixel(flatten_address(x + 1, y - 1), NORM_OFFSET) << 8;
mult_in |= read_pixel(flatten_address(x - 1, y), NORM_OFFSET) << 12;
mult_in |= read_pixel(flatten_address(x, y), NORM_OFFSET) << 16;
mult_in |= read_pixel(flatten_address(x - 1, y + 1), NORM_OFFSET) << 20;
mult_in |= read_pixel(flatten_address(x, y + 1), NORM_OFFSET) << 24;
mult_in |= read_pixel(flatten_address(x + 1, y + 1), NORM_OFFSET) << 28;
IOWR(MULT_IN_BASE, 0, mult_in);
write_pixel(flatten_address(x, y), EDGE_OFFSET, IORD(MULT_OUT_BASE, 0));
}
}
}

void display_image(int type, int mode, int starting_x, int starting_y) {
int x, y, pixel, offset;

switch (type) {
case NORM:
offset = NORM_OFFSET;
break;
case FLIPPED:
offset = FLIPPED_OFFSET;
break;
case BLURRED:
offset = BLURRED_OFFSET;
break;
case EDGE:
offset = EDGE_OFFSET;
break;
default:
offset = NORM_OFFSET;
break;
}

if (mode == SINGLE) {
for (x = 0; x < MAX_X; x++) {
for (y = 0; y < MAX_Y; y++) {
pixel = read_pixel(flatten_address(x, y), offset);
IOWR_32DIRECT(BUFFER_ADDR_BASE, 0, flatten_address(x, y));
IOWR_32DIRECT(BUFFER_DATA_BASE, 0, pixel);
}
}
} else {
int sum;
for (x = 0; x < MAX_X; x += 2) {
for (y = 0; y < MAX_Y; y += 2) {
sum = 0;
sum += read_pixel(flatten_address(x, y), offset);
sum += read_pixel(flatten_address(x + 1, y), offset);
sum += read_pixel(flatten_address(x, y + 1), offset);
sum += read_pixel(flatten_address(x + 1, y + 1), offset);
sum = sum >> 2;
IOWR_32DIRECT(BUFFER_ADDR_BASE, 0, flatten_address((x >> 1) + starting_x, (y >> 1) + starting_y));
IOWR_32DIRECT(BUFFER_DATA_BASE, 0, sum);
// IOWR_32DIRECT(BUFFER_ADDR_BASE, 0, flatten_address((x>>1)+starting_x, (y>>1)+starting_y));
// IOWR_32DIRECT(BUFFER_DATA_BASE, 0, read_pixel(flatten_address(x, y), offset));
}
}
}
}

int applySobelKernel(int x, int y, int kernel[3][3]) {
int sum = 0;
int edge = 1;

for (int ky = -edge; ky <= edge; ky++) {
for (int kx = -edge; kx <= edge; kx++) {
int px = x + kx;
int py = y + ky;
px = (px < 0) ? 0 : (px >= MAX_X ? MAX_X - 1 : px);
py = (py < 0) ? 0 : (py >= MAX_Y ? MAX_Y - 1 : py);

int pixel = read_pixel(flatten_address(px, py), NORM_OFFSET);
sum += pixel * kernel[ky + edge][kx + edge];
}
}
return sum;
}

void sobelEdgeDetection_v1() {
int sobelX[3][3] = {
{-1, 0, 1},
{-2, 0, 2},
{-1, 0, 1}};
int sobelY[3][3] = {
{-1, -2, -1},
{0, 0, 0},
{1, 2, 1}};
```

```
for (int y = 0; y < MAX_Y; y++) {
for (int x = 0; x < MAX_X; x++) {

int gradX = abs(applySobelKernel(x, y, sobelX));

int gradY = abs(applySobelKernel(x, y, sobelY));

int edgeStrength = gradX + gradY;
write_pixel(flatten_address(x, y), EDGE_OFFSET, edgeStrength);
}
}
int threshold = MAX_PIXEL_VALUE << 1;
int pixel_value;
for (int y = 0; y < MAX_Y; y++) {
for (int x = 0; x < MAX_X; x++) {
pixel_value = read_pixel(flatten_address(x, y), EDGE_OFFSET);
if (pixel_value > threshold) {
write_pixel(flatten_address(x, y), EDGE_OFFSET, threshold);
}
}
}
}


void sobelEdgeDetection_v2() {
int gradX, gradY;
int surround[3][3];
int threshold = (MAX_PIXEL_VALUE >> 1);
// handle the first and last column
for (int y=0; y<MAX_Y; y++) {
surround[0][0] = read_pixel(flatten_address(0, y - 1), NORM_OFFSET);
surround[0][1] = read_pixel(flatten_address(1, y - 1), NORM_OFFSET);
surround[1][0] = read_pixel(flatten_address(0, y), NORM_OFFSET);
surround[1][1] = read_pixel(flatten_address(1, y), NORM_OFFSET);
surround[2][0] = read_pixel(flatten_address(0, y + 1), NORM_OFFSET);
surround[2][1] = read_pixel(flatten_address(1, y + 1), NORM_OFFSET);
gradX = abs(surround[0][0] + 2*surround[1][0] + surround[2][0] - surround[0][1] - 2*surround[1][1] - surround[2][1]);
gradY = abs(-surround[0][0] - 2*surround[0][1] - surround[0][2] + surround[2][0] + 2*surround[2][1] + surround[2][2]);
gradX += gradY;
if (gradX > threshold) {
write_pixel(flatten_address(0, y), EDGE_OFFSET, threshold);
} else {
write_pixel(flatten_address(0, y), EDGE_OFFSET, gradX);
}
surround[0][1] = read_pixel(flatten_address(MAX_X - 2, y - 1), NORM_OFFSET);
surround[0][2] = read_pixel(flatten_address(MAX_X - 1, y - 1), NORM_OFFSET);
surround[1][1] = read_pixel(flatten_address(MAX_X - 2, y), NORM_OFFSET);
surround[1][2] = read_pixel(flatten_address(MAX_X - 1, y), NORM_OFFSET);
surround[2][1] = read_pixel(flatten_address(MAX_X - 2, y + 1), NORM_OFFSET);
surround[2][2] = read_pixel(flatten_address(MAX_X - 1, y + 1), NORM_OFFSET);
gradX = abs(surround[0][1] + 2*surround[1][1] + surround[2][1] - surround[0][2] - 2*surround[1][2] - surround[2][2]);
gradY = abs(-surround[0][1] - 2*surround[0][2] - surround[0][2] + surround[2][1] + 2*surround[2][2] + surround[2][2]);
gradX += gradY;
if (gradX > threshold) {
write_pixel(flatten_address(MAX_X-1, y), EDGE_OFFSET, threshold);
} else {
write_pixel(flatten_address(MAX_X-1, y), EDGE_OFFSET, gradX);
}
}
// handle the first and last row
for (int x=0; x<MAX_X; x++) {
surround[0][0] = read_pixel(flatten_address(x - 1, 0), NORM_OFFSET);
surround[0][1] = read_pixel(flatten_address(x, 0), NORM_OFFSET);
surround[0][2] = read_pixel(flatten_address(x + 1, 0), NORM_OFFSET);
surround[1][0] = read_pixel(flatten_address(x - 1, 1), NORM_OFFSET);
surround[1][1] = read_pixel(flatten_address(x, 1), NORM_OFFSET);
surround[1][2] = read_pixel(flatten_address(x + 1, 1), NORM_OFFSET);
gradX = abs(surround[0][0] + 2*surround[0][1] + surround[0][2] - surround[2][0] - 2*surround[2][1] - surround[2][2]);
gradY = abs(-surround[0][0] - 2*surround[1][0] - surround[2][0] + surround[0][2] + 2*surround[1][2] + surround[2][2]);
gradX += gradY;

if (gradX > threshold) {
write_pixel(flatten_address(x, 0), EDGE_OFFSET, threshold);
} else {
write_pixel(flatten_address(x, 0), EDGE_OFFSET, gradX);
}
surround[1][0] = read_pixel(flatten_address(x - 1, MAX_Y - 2), NORM_OFFSET);
surround[1][1] = read_pixel(flatten_address(x, MAX_Y - 2), NORM_OFFSET);
surround[1][2] = read_pixel(flatten_address(x + 1, MAX_Y - 2), NORM_OFFSET);
surround[2][0] = read_pixel(flatten_address(x - 1, MAX_Y - 1), NORM_OFFSET);
surround[2][1] = read_pixel(flatten_address(x, MAX_Y - 1), NORM_OFFSET);
surround[2][2] = read_pixel(flatten_address(x + 1, MAX_Y - 1), NORM_OFFSET);
```

```
gradX = abs(surround[1][0] + 2*surround[1][1] + surround[1][2] - surround[2][0] - 2*surround[2][1] - surround[2][2]);
gradY = abs(-surround[1][0] - 2*surround[1][1] - surround[1][2] + surround[2][0] + 2*surround[2][1] + surround[2][2]);
gradX += gradY;

if (gradX > threshold) {
write_pixel(flatten_address(x, MAX_Y-1), EDGE_OFFSET, threshold);
} else {
write_pixel(flatten_address(x, MAX_Y-1), EDGE_OFFSET, gradX);
}
}

// handle the rest
for (int y = 1; y < MAX_Y - 1; y++) {
for (int x = 1; x < MAX_X - 1; x++) {
surround[0][0] = read_pixel(flatten_address(x - 1, y - 1), NORM_OFFSET);
surround[0][1] = read_pixel(flatten_address(x, y - 1), NORM_OFFSET);
surround[0][2] = read_pixel(flatten_address(x + 1, y - 1), NORM_OFFSET);
surround[1][0] = read_pixel(flatten_address(x - 1, y), NORM_OFFSET);
surround[1][2] = read_pixel(flatten_address(x, y), NORM_OFFSET);
surround[2][0] = read_pixel(flatten_address(x - 1, y + 1), NORM_OFFSET);
surround[2][1] = read_pixel(flatten_address(x, y + 1), NORM_OFFSET);
surround[2][2] = read_pixel(flatten_address(x + 1, y + 1), NORM_OFFSET);

// x gradient
gradX = abs(surround[0][0] + 2*surround[1][0] + surround[2][0] - surround[0][2] - 2*surround[1][2] - surround[2][2]);
// y gradient
gradY = abs(-surround[0][0] - 2*surround[0][1] - surround[0][2] + surround[2][0] + 2*surround[2][1] + surround[2][2]);
gradX += gradY;
if (gradX > threshold) {
write_pixel(flatten_address(x, y), EDGE_OFFSET, threshold);
} else {
write_pixel(flatten_address(x, y), EDGE_OFFSET, gradX);
}
}
}
}
```

Interrupt Handler

Header

```
/**
 *   Name: interrupt_handler.h
 *   Description:
 *       This file contains the declaration of ISRs and key mode handler
 *   Last edited: 17/05/2024
 *   Author: Jiayi Gu  | ID: 33114404
 *           Yulan Sui | ID: 31973647
 */

#ifndef INTERRUPT_HANDLER_H_
#define INTERRUPT_HANDLER_H_


/**
 * Handles key interrupts.
 */
void key_handler();

/**
 * Handles switch interrupts.
 */
void switch_handler();

/**
 * Handles key mode interrupts.
 */
void key_mode_handler(void *pdata);


#endif /* INTERRUPT_HANDLER_H_ */
```

Source

```
/**
 *   Name: interrupt_handler.c
 *   Description:
 *       This file contains the declaration of ISR and key mode handler
 *   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/

#include "interrupt_handler.h"
#include "global_var.h"
#include "system.h"
#include "io.h"
#include <stdio.h>

void key_handler() {
int edgeCapture = IORD_32DIRECT(KEY_BASE, 3 << 2);
key = edgeCapture & 0b11;
IOWR_32DIRECT(KEY_BASE, 3 << 2, edgeCapture);
OSSemPost(key_siganl);
}

void switch_handler() {
IOWR_32DIRECT(SW_BASE, 3 << 2, IORD_32DIRECT(SW_BASE, 3 << 2));
OSSemPost(switch_signal);
}

void key_mode_handler(void *pdata) {
INT8U err;
while (1) {
OSSemPend(key_siganl, 0, &err);
OSMutexPend(mode_mutex, 0, &err);
if (key == 0b01) {
mode = SINGLE;
} else if (key == 0b10) {
mode = QUAD;
}
OSMutexPost(mode_mutex);
}
}
```

Main

Header

No Header present for this component.

Source

```
/**
*   Name: main.c
*   Description:
*       This file contains the main function that creates all tasks and starts the OS
*       Note that main task handles switches.
*   Last edited: 17/05/2024
*   Author: Jiayi Gu  | ID: 33114404
*           Yulan Sui | ID: 31973647
*/

#include "benchmark.h"
#include "display_tasks.h"
#include "global_var.h"
#include <includes.h>
#include "interrupt_handler.h"
#include "image_processing.h"
#include <stdio.h>
#include "io.h"
#include "system.h"

#define OS_MUTEX_EN 1
#define OS_TASK_STAT_EN 1

// For interrupt handling
void *context;
void *flags;

void main_task(void *pdata) {
OSStatInit();
INT8U err;
err = OSTaskCreateExt(tl_display,
(void *)0,
(OS_STK *) &tl_display_stk[TASK_STACKSIZE - 1],
TL_PRIORITY,
TL_PRIORITY,
(OS_STK *) &tl_display_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

```
err = OSTaskCreateExt(tr_display,
(void *)0,
(OS_STK *) &tr_display_stk[TASK_STACKSIZE - 1],
TR_PRIORITY,
TR_PRIORITY,
(OS_STK *) &tr_display_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


err = OSTaskCreateExt(bl_display,
(void *)0,
(OS_STK *) &bl_display_stk[TASK_STACKSIZE - 1],
BL_PRIORITY,
BL_PRIORITY,
(OS_STK *) &bl_display_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


err = OSTaskCreateExt(br_display,
(void *)0,
(OS_STK *) &br_display_stk[TASK_STACKSIZE - 1],
BR_PRIORITY,
BR_PRIORITY,
(OS_STK *) &br_display_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


err = OSTaskCreateExt(process_image,
(void *)0,
(OS_STK *) &process_image_stk[TASK_STACKSIZE - 1],
PROCESS_IMAGE_PRIORITY,
PROCESS_IMAGE_PRIORITY,
(OS_STK *) &process_image_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


err = OSTaskCreateExt(benchmark,
(void *)0,
(OS_STK *) &benchmark_stk[TASK_STACKSIZE - 1],
BENCHMARK_PRIORITY,
BENCHMARK_PRIORITY,
(OS_STK *) &benchmark_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


err = OSTaskCreateExt(key_mode_handler,
(void *)0,
(OS_STK *) &key_stk[TASK_STACKSIZE - 1],
KEY_PRIORITY,
KEY_PRIORITY,
(OS_STK *) &key_stk[0],
TASK_STACKSIZE,
(void *)0,
OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);


while (1) {
OSSemPend(switch_signal, 0, &err);
switch_state = IORD_32DIRECT(SW_BASE, 0) & 0x3FF;
OSMutexPend(mode_mutex, 0, &err);
hex_time = (switch_state >> 8) & 0b11;
image_displayed[TL] = switch_state & 0b11;
image_displayed[TR] = (switch_state >> 2) & 0b11;
image_displayed[BL] = (switch_state >> 4) & 0b11;
image_displayed[BR] = (switch_state >> 6) & 0b11;
OSMutexPost(mode_mutex);
}
}

/* The main function creates two task and starts multi-tasking */
int main(void) {
```

```
printf("Starting...");
// register interrupts
alt_ic_isr_register(KEY_IRQ_INTERRUPT_CONTROLLER_ID, KEY_IRQ, key_handler, &context, 0x0);
IOWR_32DIRECT(KEY_BASE, 3 << 2, 0b11);
IOWR_32DIRECT(KEY_BASE, 2 << 2, 0b11); // enable interrupts for both keys
IOWR_32DIRECT(KEY_BASE, 3 << 2, 0b11);

alt_ic_isr_register(SW_IRQ_INTERRUPT_CONTROLLER_ID, SW_IRQ, switch_handler, &context, 0x0);
IOWR_32DIRECT(SW_BASE, 3 << 2, 0x3FF);
IOWR_32DIRECT(SW_BASE, 2 << 2, 0x3FF); // enable interrupts for all 10 switches
IOWR_32DIRECT(SW_BASE, 3 << 2, 0x3FF);
INT8U err;
mode_mutex = OSMutexCreate(MODE_MUTEX_PRIORITY, &err);

for (int i = 0; i < 4; i++){
image_mutex[i] = OSMutexCreate(IMAGE_MUTEX_PRIORITY + i, &err);
}

switch_signal = OSSemCreate(0);

key_siganl = OSSemCreate(0);

time_mutex = OSMutexCreate(TIME_MUTEX_PRIORITY, &err);


err = OSTaskCreateExt(main_task,
(void *)0,
(OS_STK *) &main_stk[TASK_STACKSIZE - 1],
MAIN_PRIORITY,
MAIN_PRIORITY,
(OS_STK *)&main_stk[0],
TASK_STACKSIZE,
(void *)0,
0);

OSStart();
return 0;
}
```