Battle Tetris

20111493 김준영 (Jun Yeong Kim)
20111526 원재성 (Jae Seung Won)

1. **제목 :** Battle Tetris

2. 목표

(1) 10주간의 실험을 통해 배운 내용을 응용 및 조합하여 하나의 Application을 만든다.
(2) 제한된 조건 내에서 최대한 기능을 설계하는 과정을 통해 설계 능력을 높인다.

3. 관련 이론

설계 과정에서 Interrupt, GPIO, Timer를 이용하였다.

(1) Interrupt

1-1) Interrupt and Exception

Interrupt와 exception은 프로세서의 처리 관점에서 보면 본질적으로 같다. 예기치 않은 시점에서 발생하는 사건 또는 상태를 의미한다. Interrupt는 시스템 설계자 또는 프로그래머에 의해 사전에 설정된 하드웨어 입출력 장치 또는 소프트웨어 명령으로 발생한다. Exception은 정상적인 경우라면 발생하지 않아야 하지만 전체 시스템에 영향을 미칠만한 예상치 못한 오류/문제로 인해 발생한다.

현재 수행 중인 프로그램에서 한 instruction을 수행 하던 중 interrupt/exception이 발생하였다면, 우선 instruction을 마저 수행한다. 그 다음 instruction을 수행 하지 않고 프로그램의 수행을 멈춘다. 프로세서는 예외적으로 발생한 interrupt/exception에 대한 조치를 취한다. 이러한 조치는 해당 interrupt/exception에 적절한 interrupt service routine(또는 interrupt handler)을 수행하면서 이루어진다. 그 처리를 마친 후 인터럽트가 발생할 당시에 수행했던 명령어 바로 다음 명령어로 복귀한다. subroutine의 수행과 상황이 비슷하지만 언제 걸릴지 모른다는 점에서 차이가 있다.

1-2) Vector Table

인터럽트가 발생하면 프로세서는 이를 처리하기 위한 ISR의 시작 주소를 알 수 있어야 한다. 이 정보는 메모리상에 정의되는 Vector Table에 저장된다. 인터럽트가 발생하면 프로세서는 Vector Table에서 ISR의 시작 주소를 Fetch한다. 우리가 사용하는 stm32f103xx device에서는 15번까지가 exception, 16번부터 인터럽트가 정의되어 있다. Vector Table은 일반적으로 flash 또는 ROM에 저장되어 있는데, 프로그램 수행 중에 필요에 따라 그 내용을 변경시키기 위해서 Vector Table의 위치를 RAM으로 변경시키는 relocation을 수행할 수 있다.

1-3) Interrupt sequence

1) Interrupt 발생

인터럽트가 발생하면 다음과 같은 과정이 진행된다.

Interrupt가 발생하면 stacking, vector fetch, update register의 세 가지 과정이 진행된다.

i) Stacking - 인터럽트가 발생하면 r0-r3, r12, lr, pc, psr 8개 레지스터의 내용을 stack에 push한다.

ii) Vector Fetch

DBUS(Data bus)를 이용하여 8개의 Register를 stacking하면서 동시에 Vector table에서 ISR의 시작 주소를 IBUS(Instruction bus)를 이용하여 Fetch한다.

iii) SP, PC, LR update

8개의 register가 stack에 push 되므로 sp가 update된다. Vector fetch과정으로 불러온 ISR의 시작 주소를 PC에 load한다. LR에는 EXC_RETURN가 저장되는데, 하위 4bit가 ISR로부터의 복귀에 관련된 정보를 제공한다.

2) Instruction return

ISR의 수행이 끝나고 main program으로 돌아가는 과정은 다음과 같다.

i) LR에 저장되어 있던 EXC_RETURN을 PC에 Load한다. EXC_RETURN값에 따른 동작은 다음과 같다.

Value	Condition
0xFFFFFFF1	Return to handler mode
0xFFFFFFF9	Return to thread mode and on return use the main stack
0xFFFFFFFD	Return to thread mode and on return use the process stack

EXC_RETURN CASES

ii) Unstacking

stack에 저장되어 있던 레지스터들을 복구되고, SP도 인터럽트 이전 값으로 복원된다.

iii) NVIC Register Update : Active bit이 해제된다. 외부 인터럽트의 경우 외부 신호가 유지되고 있다면 pending bit이 다시 설정되고 ISR이 다시 수행된다.

1-4) NVIC and Interrupt Control

NVIC는 Cortex-M3 프로세서에 내장된 인터럽트 처리를 위한 controller이다. 인터럽트뿐만 아니라 SYSTICK timer 제어를 위한 레지스터도 포함하고 있다. NVIC는 외부 인터럽트 NMI등을 지원한다. 다음 표는 NVIC의 여러 Register의 동작을 나타내고 있다.

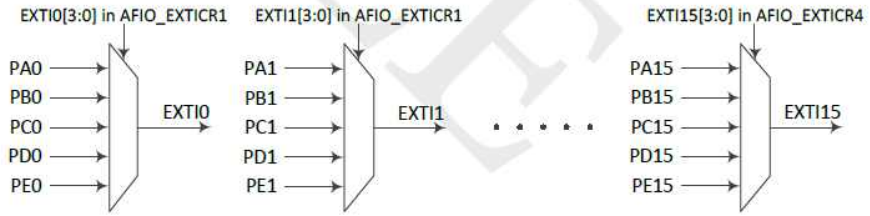
NVIC		내용
NVIC_ISERx	Interrupt set enable register	특정 인터럽트 사용여부 지정
NVIC_ICERx	Interrupt ser clear register	특정 인터럽트 무시 지정
NVIC_ISPRx	Interrupt set pending register	인터럽트 pending 상태 지정
NVIC_ICPRx	Interrupt clear pending register	인터럽트 pending 상태 해제
NVIC_IABRx	Interrupt active bit register	인터럽트의 active 상태
NVIC_IPRx	Interupt priority register	인터럽트의 priority

1-5) EXTI (External Interrupt/event Controller)

대부분의 인터럽트는 외부 소자/장치에 연결된다. 따라서 외부 인터럽트를 처리하기 위한 연결 관계 및 그 동작에 대해 알아야 한다. EXTI는 다른 소자 혹은 외부 장치로부터 발생하는 19개의 신호를 감지하여 인터럽트를 요청할 수 있게 해준다. 아래 표는 EXTI Register의 동작을 나타내고 있다.

Register		내용
EXTI_RTSR	rising trigger selection register	rising edge 로 동작할 interrupt 지정
EXTI_FTSR	falling trigger selection register	falling edge 로 동작할 interrupt 지정
EXTI_IMR	interrupt mask register	처리할 Interrupt 지정
EXTI_PR	pending register	1을 입력하면 Pending 해제
EXTI_SWIER	software interrupt event register	software적으로 interrupt 발생

19개의 EXTI 중 16개는 GPIO를 통해 연결된다. 5개의 port(PA, PB, PC, PD, PE)의 80개의 신호로부터 16개가 연결된다. 연결 형태는 아래의 그림과 같다.



EXTI-GPIO mapping

AFIO_EXTICR 레지스터에서 해당 EXTI의 bit 설정을 통해 5개 포트 중 하나의 신호를 선택 가능하다. (00:PA 01:PB 10:PC 11:PD 100:PE) 위와 같이 구성된 EXTI 신호들은 vector table에서 ISR과 어떻게 연결되는지 확인할 수 있다. EXTI-GPIO Mapping을 끝내고 EXTI_IMR 레지스터에 해당 EXTI 번호에 대한 bit을 1로 설정하고 NVIC_ISER 레지스터에 인터럽트 번호에 맞게 1로 설정해 주면 외부 포트 PA~PE를 인터럽트로 활용할 수 있다.

(2) GPIO

2-1) GPIO

마이크로프로세서를 이용하여 시스템을 구현하는 경우, 다양하고 많은 수의 입출력 장치/소자의 연결이 필요하다. 이에 반해서, 이러한 일반적인 입출력 연결을 위한 소자들을 칩에 내장하는 마이크로콘트롤러는 사용자에 따라 다양한 구조를 고려하여 설계되었다. 많은 수의 기본적인 입출력 포트 구조를 제공하되 이들을 프로그램에 의해 선택적으로 설정(configuration)할 수 있도록 사용자가 목적에 맞추어 동작시킬 수 있도록 해준다. 프로세서는 외부와의 연결을 위한 가장 기본적인 소자로 GPIO(general purpose inputs and outputs)를 꼽을 수 있다.

2-2) GPIO Functional Description

1) GPIO 입•출력 형식

GPIO에 속한 핀들은 프로그램을 통해 어떻게 설정(configuration)하는가에 따라 다양한 형식의 입력 혹은 출력으로 사용할 수 있다. 선택 가능한 입출력 형식을 정리하면 다음과 같다. stm32f10x 계열 프로세서에서 5개의 GPIO port는 APB2 Bus에 연결되어 있다.

•Input floating
•Input pull-up
•Input pull-down
•Analog
•Output open-drain
•Alternate function push-pull
•Alternate function open-drain

교재에서 사용하는 프로세서 Cortex-M3에서는 5개의 GPIO ports(GPIOA, GPIOB, GPIOC, GPIOD, GPIOE)가 제공된다. 각 port는 16bits로 구성되며, GPIOx의 신호를 Px[15:0]과 같은 형식으로 표시한다. 예를 들어, PA6은 port A의 bit 6을 의미한다. 따라서, 최대 80개의 GPIO port를 사용할 수 있다.

2) Registers of GPIO

port의 각 핀마다 선택적으로 기능과 동작형식을 정의하여 사용하기 위해 각 GPIO port 마다 다음과 같은 레지스터들이 마련되었다.

•두 개의 32-bit configuration registers (GPIOx_CRL, GPIOx_CRH)
•두 개의 32-bit data registers (GPIOx_IDR, GPIOx_ODR)
•한 개의 32-bit set/reset register (GPIOx_BSRR)
•한 개의 16-bit reset register (GPIOx_BRR)
•한 개의 32-bit locking register (GPIOx_LCKR)

각 port의 bit들의 동작을 자유롭게 설정하기 위해서 레지스터들의 access에는 32bit word단위로만 할 수 있다. 또한, Peripherals 영역도 bit-band alias가 제공되므로 bit-band operation을 통해서 레지스터들의 bit에 대한 set/reset을 atomic하게 수행할 수 있다.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOx_CRL	CNF7 [1:0]		MODE7 [1:0]		CNF6 [1:0]		MODE6 [1:0]		CNF5 [1:0]		MODE5 [1:0]		CNF4 [1:0]		MODE4 [1:0]		CNF3 [1:0]		MODE3 [1:0]		CNF2 [1:0]		MODE2 [1:0]		CNF1 [1:0]		MODE1 [1:0]		CNF0 [1:0]		MODE0 [1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x04	GPIOx_CRH	CNF 15 [1:0]		MODE1 5 [1:0]		CNF 14 [1:0]		MODE1 4 [1:0]		CNF 13 [1:0]		MODE1 3 [1:0]		CNF 12 [1:0]		MODE1 2 [1:0]		CNF 11 [1:0]		MODE1 1 [1:0]		CNF 10 [1:0]		MODE1 0 [1:0]		CNF 9 [1:0]		MODE9 [1:0]		CNF 8 [1:0]		MODE8 [1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x08	GPIOx_IDR	Reserved																IDR[15:0]I																
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	GPIOx_ODR	Reserved																ODR[15:0]																
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	GPIOx_BSRR	BR[15:0]																BSR[15:0]																
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	GPIOx_BRR	Reserved																BR[15:0]																
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	GPIOx_LCKR	Reserved															LCKK	LCK[15:0]																
	Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIO register map

a) configuration registers (GPIOx_CRL, GPIOx_CRH)

GPIO의 기능을 설정하는 레지스터이다. 각 pin의 기능은 configuration register의 CNFy[1:0]와 MODEy[1:0]에 의해 설정된다. 다음 표에 따라서 Configuration mode가 결정된다.

Configuration mode		CNF1	CNF0	MODE1	MODE0
Input	Analog	0	0	00	
	Input floating		1		
	Input with pull-up/pull-down	1	0		
General purpose output	Push-pull	0	0	max. output speed	
	Open-drain		1	01	10 MHz
Alternate Function output	Push-pull	1	0	10	2 MHz
	Open-drain		1	11	50 MHz

Port bit configuration table

b) data register (GPIOx_IDR, GPIOx_ODR)

한 pin이 출력으로 동작하도록 설정되면 output data register에 쓰여지는 값이 해당 pin에 출력된다. input data register는 입출력 pin에 나타나는 신호를 APB2 클락 사이클 마다 저장한다.

c) set/reser register (GPIOx_BSRR, GPIOx_BRR)

GPIO port에 데이터를 bit 단위로 쓸 때 이로 인한 인터럽트 발생을 염려하거나 관심 bit 이외의 bit에 영향을 미치는 것을 방지하면서 한 bit 또는 다수의 bit에 한 번의 atomic write를 수행할 수 있도록 해주는 레지스터이다.

Bit Set/Reset register(GPIOx_BSRR)의 관심 bit에 1을 쓰면 그 위치의 GPIOx_ODR 데이터가 set(또는 reset)된다. 1을 쓰지 않은 (0을 쓴) bit의 위치에 해당하는 GPIOx_ODR의 데이터는 영향을 받지 않는다. GPIOx_BRR은 해당 데이터 bit를 reset만 시킨다는 점에서 BSRR의 부분적 기능을 수행한다.

d) locking register (GPIOx_LCKR)

GPIO의 동작설정이 끝난 후 어떤 경우에도 그 설정내용이 변경되는 것을 방지하고 싶으면 GPIOx_LCKR 레지스터의 LCK[15:0]의 해당 bit에 1을 쓴다. GPIOx_LCKR의 LCKK bit가 0일 때에만 LCK[15:0]의 내용을 변경할 수 있다.

2-3) Alternate function I/O (AFIO)

한 프로세서가 서로 다른 pin 수와 배열을 갖는 package로 제작될 수 있는데, 이 경우 특정 주변연결 소자의 입출력 신호를 특정 GPIO pin으로 기본배정 (default)하는 것이 불가능한 경우가 있다. 이 경우에 일부 Alternate Function에 대하여 기본배정과 다른 pin으로 연결하여 사용할 수 있게 해준다. 이를 위한 레지스터를 AFIO라고 하며, 설정에 의해 기본배정과 다르게 변경할 수 있다. pin-Remapping을 통해 Alternate Function 역시 다르게 할 수 있다. remapping과 연관된 레지스터는 AFIO_MAPR이다.

(3) Timer

3-1) 타이머 구조

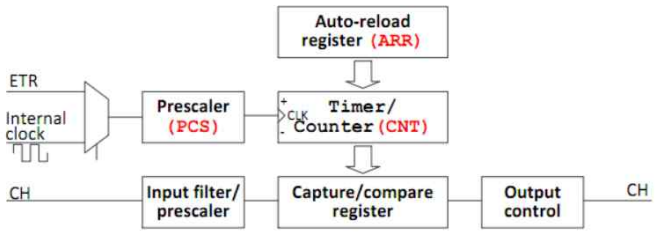


Figure 9.1: Basic function block of timer/counter.

타이머의 Clock으로 외부신호(ETR)와 프로세서 내부 clock(INT_CLK) 중에서 선택할 수 있다. 선택된 clock 신호는 prescaler (PSC)를 통해 counter에 연결되는데, 설정에 따라 prescaler는 입력되는 신호의 주파수를 분주하게 된다. Timer/Counter는 설정에 따라 up/down counting을 선택적으로 수행할 수 있다. auto-reload register (ARR)에 특정 값을 쓰면 이 값이 타이머의 초기값으로 사용된다. ARR에서 downcounting하여 0에 도달하여 underflow가 발생하거나 0에서 upcounting으로 ARR까지 도달하여 overflow가 발생하는지의 여부를 capture/compare register에서 확인할 수 있다.

3-2) 타이머 종류

우리가 사용하는 STM32F10X도 서로 목적을 달리하는 7개의 타이머를 내장하고 있다.

•SysTick Timer : OS를 위해 사용된다.

•Indepentdent watchdog : 장치에 문제가 발생했을 때 초기화하거나 프로그램의 timeout 관리 목적에 이용된다.

•Window watchdog : 장치에 문제가 발생했을 때 초기화 하는데 이용된다.

•Advanced-control timer (TIM1) and General-purpose timers (TIMx) : 한 개의 advanced-control 타이머 (TIM1)와 세 개의 범용 타이머 (TIM2-4)가 내장되어 있다. 일부 기능을 제외하면 TIM1은 TIM2-4와 그 기능이 동일하다. 이 타이머를 어떻게 설정하는가에 따라 매우 다양한 기능을 제공한다.

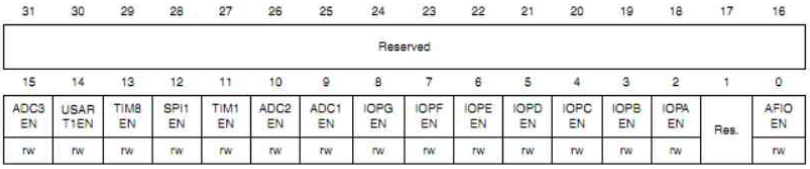
3-3) Timer Functional Description

1) RCC (Reset and Clock Control)

마이크로컨트롤러에 포함된 주변장치연결소자(Peripherals)에는 clock이 제공되어야 하는데 타이머도 마찬가지이다. 연결 소자들에는 주로 APB1과 APB2로 불리는 내부 버스를 통해 연결된다. 따라서, 사용하려는 연결소자들에 clock을 공급하기 위해서는 RCC_APB1ENR과 RCC_APB2ENR 두 레지스터를 통한 설정이 필요하다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DAC EN	PWR EN	BKP EN	Res.	CAN EN	Res.	USB EN	I2C2 EN	I2C1 EN	UART5E N	UART4 EN	USART 3EN	USART 2EN	Res.
Res.		rw	rw	rw	Res.	rw	Res.	rw	rw	rw	rw	rw	rw	rw	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWD GEN	Reserved					TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw	Res.		rw	Res.					rw	rw	rw	rw	rw	rw

RCC_APB1ENR Register map



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART1EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

RCC_APB2ENR Register map

이전 주차 실험에서 GPIO를 이용하기 위해 RCC_APB2ENR에 0x00000071을 Load했는데, Register map을 통해 그 이유를 이해할 수 있다. (RCC_APB2ENR의 LSB AFIO는 칩의 핀 개수 부족으로 alternate function을 이용하기 때문에 1로 한 것이다.) 우리는 TIM1-4를 이용하므로 RCC_APB2ENR의 bit11, RCC_APB1ENR의 bit0-2를 1로 해야 한다.

2) Clock Input Sources

Counter에 제공되는 Clock 신호를 다음과 같이 정리하였다.

Internal Clock	External Clock Mode1	External Clock Mode2
RCC에 의해 제공되는 내부 Clock에 의해 동작	외부 clock신호가 타이머의 구동에 사용. (외부 Clock 신호가 타이머의 Clock신호로 이용됨)	외부 Trigger 신호인 ETR pin이 타이머의 입력신호로 사용됨 (외부 Trigger 신호 = Counter의 Enable신호)

각 Clock 신호를 사용하기 위해서 Configuration Register의 설정이 필요하다.

3-4) Time Base Generator

• Counter Register (TIMx_CNT) : Counter의 clock신호는 prescaler의 출력인 CK_CNT에 의해 제공되는데, 이 신호는 TIMx_CR1의 CEN을 1로 했을 때 동작한다.

• Prescaler Register (TIMx_PSC) : 공급되는 clock 신호의 주파수를 16-bit 레지스터인 TIMx_PSC의 내용에 따라 분주한다.

• Auto-Reload Register (TIMx_ARR) : Event 발생 때 마다 Counter의 내용을 초기화하는 값이 저장되어 있다.

Clock신호, PSC, ARR 레지스터의 설정에 따라 UEV(Update Event)를 발생시킬 수 있다. Event의 발생 주기는 다음 식에 따른다

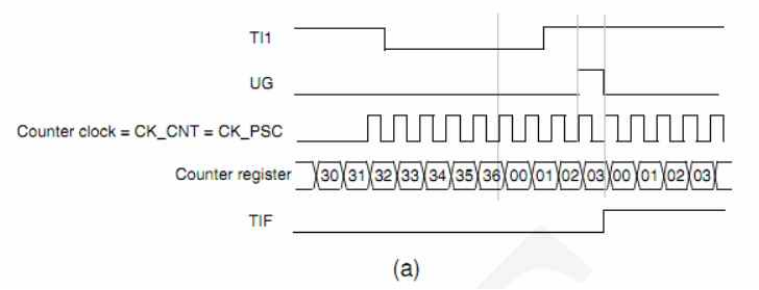
Update_Event = TIM_CLK / ((PSC + 1) * (ARR + 1) * (RCR + 1))

TIM_CLK는 프로세서에 장착된 clock으로 72MHz이다. RCR은 TIM1에만 제공된다.

3-5) Timers and external trigger synchronization

TIMx 타이머는 External Trigger를 이용하여 여러 모드에서 동작할 수 있다.

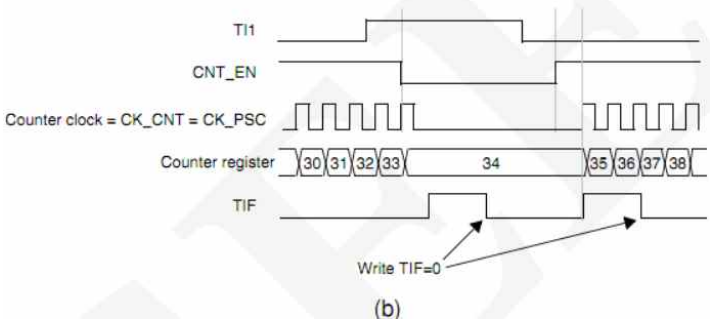
• Reset Mode



(a)

Reset Mode는 Trigger Input에 의해 Counter가 Reset된다.

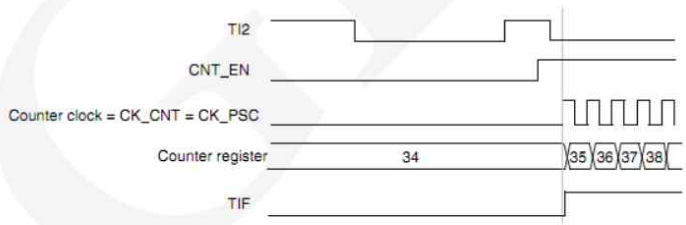
• Gated Mode



(b)

Gated mode는 Trigger Input의 Rising Edge와 Falling Edge에서 모두 TIF (Trigger Interrupt Flag)가 발생한다. Rising Edge의 TIF에서 CEN=0이 되어 Counter는 멈추고 Falling Edge의 TIF에서 CEN=1이 되어 Counter는 다시 동작한다.

• Trigger Mode



(c)

Trigger Mode는 Trigger Input에 의해 TIF가 발생하면 멈춰 있던 Counter에서 CEN=1이 되어 Counter가 다시 동작한다. 즉, External Trigger가 Counter를 다시 동작하게 하는 Trigger역할을 한다.

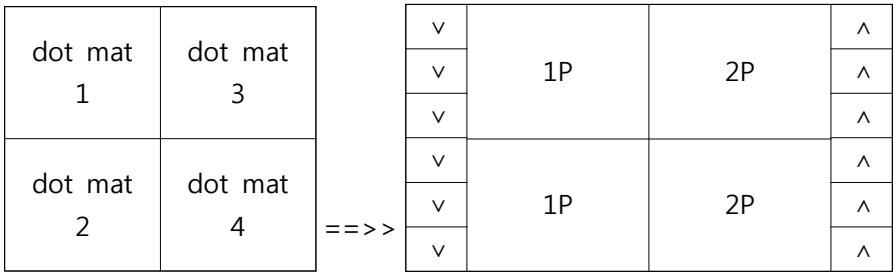
4. 설계 과정

(1) Hardware

1) 사용한 부품 List

부품	수량	부품	수량
STM32VLDISCOVERY	1	4x4 Keypad	1
2 color 8x8 dot matrix	4	PCB	1

아래와 같이 4개의 dot matrix중 2개씩 세로로 배치하여 1p와 2p에 대한 tetris stage를 나타내도록 하였다.



Keypad를 이용하여 1p와 2p의 동작을 나타내도록 하였다. 동작은 일반적인 Tetris 게임과 같이 left, right, down, rotate 4가지로 구분되어 있다.

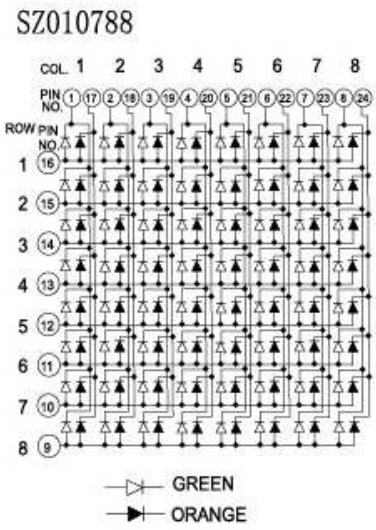
	Connector to Board				
1p left	4	3	2	1	2P rotate
1p down	8	7	6	5	2P right
1p right	12	11	10	9	2P down
1p rotate	16	15	14	13	2P left

즉 1p는 4, 8, 12, 16번 버튼을 사용하고, 2p는 1, 5, 9, 13번 버튼을 사용한다.

2) dot matrix 구성

8x8 dot matrix 4개를 정사각형 모양으로 붙여서 16x16꼴을 만들었다. 각 8x8 dot matrix는 가로 제어에 8개의 GPIO pin, 세로 제어에 8개의 GPIO pin이 필요해서 총 64개(16X4=64)의 GPIO pin이 필요하나, Latch를 사용하지 않고 16개의 GPIO pin을 이용해서 가로를 제어하고, 16개의 GPIO pin을 이용해서 세로를 제어하도록 설계 하여 총 32개의 pin으로 제어하도록 설계하였다.

먼저 8x8 dot matrix의 pin 구성이다.



Dot Matrix 후면 pin			
24	9	8	상표
23	10	7	
22	11	6	
21	12	5	
20	13	4	
19	14	3	
18	15	2	
17	16	1	
GREEN	ROW	RED	

8x8 dot matrix에서 row에 1을 write하고 column의 Green이나 red에 0을 쓰면 해당 색이 출력되는 형식이다.

dot mat 1	dot mat 3
dot mat 2	dot mat 4

앞서 다음과 같이 dot matrix를 배치했는데, 1과 3, 2와 4의 row pin을 이어주고, 1과 2, 3과 4의 column pin을 이어주면, 16개의 GPIO pin으로 row를 제어할 수 있고, 16개의 GPIO pin으로 column을 제어할 수 있다.

정리하면 다음과 같다.

Dot Matrix 1			Dot Matrix 3		
Green8	Row1	Red8	Green16	Row1	Red16
Green7	Row2	Red7	Green15	Row2	Red15
Green6	Row3	Red6	Green14	Row3	Red14
Green5	Row4	Red5	Green13	Row4	Red13
Green4	Row5	Red4	Green12	Row5	Red12
Green3	Row6	Red3	Green11	Row6	Red11
Green2	Row7	Red2	Green10	Row7	Red10
Green1	Row8	Red1	Green9	Row8	Red9
Dot Matrix 2			Dot Matrix 4		
Green8	Row9	Red8	Green16	Row9	Red16
Green7	Row10	Red7	Green15	Row10	Red15
Green6	Row11	Red6	Green14	Row11	Red14
Green5	Row12	Red5	Green13	Row12	Red13
Green4	Row13	Red4	Green12	Row13	Red12
Green3	Row14	Red3	Green11	Row14	Red11
Green2	Row15	Red2	Green10	Row15	Red10
Green1	Row16	Red1	Green9	Row16	Red9

이렇게 16개의 GPIO pin으로 16Rows를 제어하고, 16개의 GPIO pin으로 16 Columns를 제어하도록 하였다. (1P – Green / 2P – Red)

3) pin mapping

Debugging을 위한 jtag pin (PA13-15, PB3-4)를 제외하고 나머지 핀을 최대한 이용하였다. 먼저 row, column에 대한 pin map은 다음과 같다.

fn	row0	row1	row2	row3	row4	row5	row6	row7
pin	PA1	PA2	PA3	PA4	PA5	PA6	PA7	PA8
fn	row8	row9	row10	row11	row12	row13	row14	row15
pin	PA9	PA9	PA9	PA9	PB5	PB6	PB7	PB8
fn	col0	col1	col2	col3	col4	col5	col6	col7
pin	PC0	PC1	PC2	PC3	PC4	PC5	PC6	PC7
fn	col8	col9	col10	col11	col12	col13	col14	col15
pin	PC8	PC9	PC10	PC11	PC12	PC13	PB2	PB9

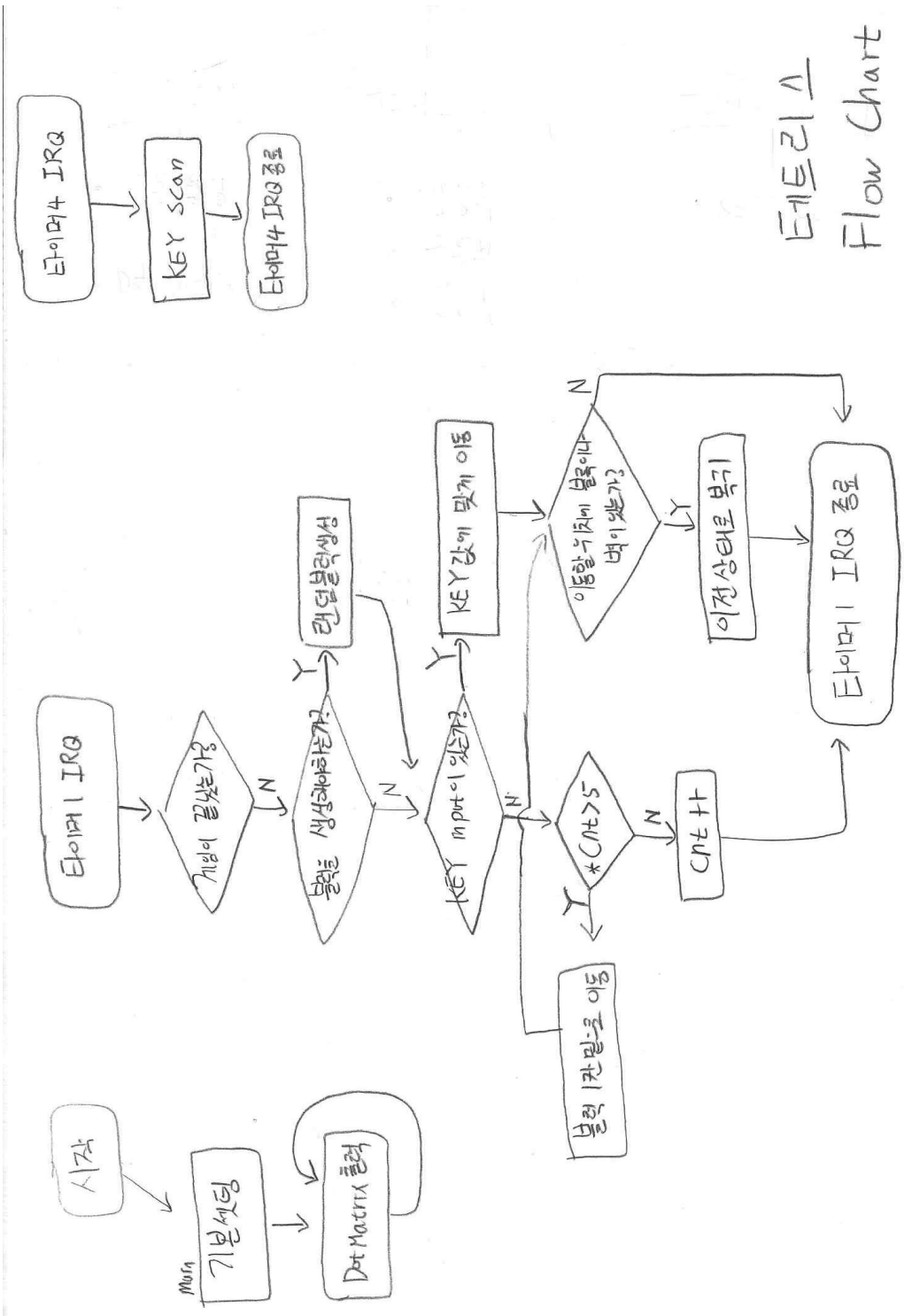
다음은 TIMER이다. key scan을 위해서 TIM1, tetris 게임 진행을 위해서 TIM4를 이용했는데, 2개의 TIMER는 최대한 보드에서 외부로 노출되지 않은 핀을 이용하였다. TIM1의 구동에 PE7-15, TIM4의 구동에 PD12-15를 이용하였다. 이를 위해서 AFIO_MAPR 레지스터에서 TIM1과 TIM4에 대하여 Remap을 설정하였다.

Key Scan을 위해서 4개의 GPIO output을 통해 Keyscan의 row에 공급해 주고, 누른 버튼의 출력을 4개의 GPIO input으로 받도록 한다. 정리하면 다음과 같다.

fn	row0	row1	row2	row3	col0	col1	col2	col3
pin	PB0	PB1	PB10	PB11	PB12	PB13	PB14	PB15
I/O	output				input			

(2) Software

1) Flow Chart



2) 기본 System

기본적으로 메인 Sytsem 에 사용된 것들은 다음과 같다.

Main 부문에서는 pin mapping에 맞추어 기본적인 GPIO와 타이머 설정을 하고 while 문에서 Display를 담당하도록 하였다.

TIM4 update interrupt handler에서 Key Scan을 담당한다.

TIM1 update interrupt에서 Tetris 관련 동작(생성, 구동, 연산)을 수행하도록 하였다.

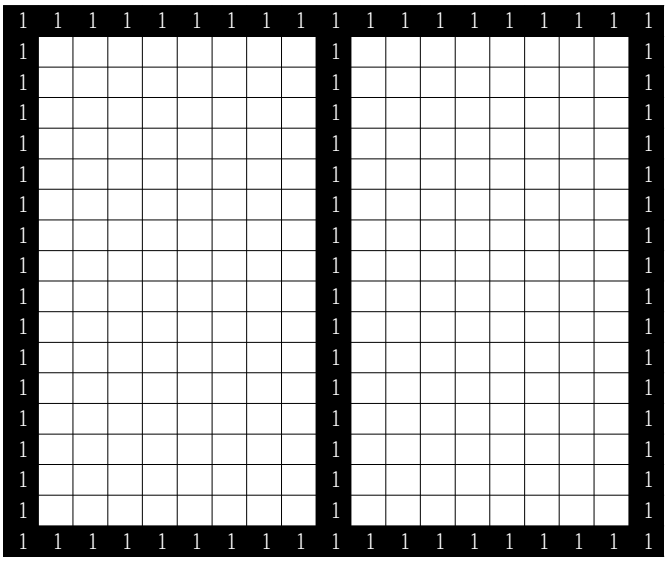
```
#include <stm32f10x.h>

u16 sdata, cdata, cdata_temp;
int TData [19][18], makenew,autodown,dectemp;
int Tblk1[3][3], Tblk11[4][4], Tx1, Txt1, Ty1, Tyt1, Tmove1, Ttype1, Trot1,Talert1,Tdel1, Tline1, Tline2;
int Tblk2[3][3], Tblk22[3][4], Tx2, Txt2, Ty2, Tyt2, Tmove2, Ttype2, Trot2, Talert2, Tdel2, p1end, p2end,p1c, p1r, p2c, p2r;
int p1e1, p2e1;
int autodown1, autodown2;
int Tline_tmp1, Tline_tmp2, Tline1,Tline2,Tline_attack1,Tline_attack2;

u16 scanpat[4] = { 0xFFFE,0xFFFD,0xFBFF, 0xF7FF};
u16 colpat[4] = {0x0001, 0x0002, 0x0004, 0x0008};
u8 key;
int ii,jj,blk,blk2,e,f,p,q,delay,delay2,keydelay, temp1, temp2, temp3, temp4;
        u8 c;

int main (void) {
{
        int i,j;
        //GPIO & AFIO Setup

  RCC->APB2ENR = 0x000000FD;
        GPIOA->CRL = 0x33333333;
        GPIOA->CRH = 0x33333333;
        GPIOB->CRL = 0x33333333;
        GPIOB->CRH = 0x88883333;
        GPIOB->ODR |= 0xFC03;
        GPIOC->CRL = 0x33333333;
        GPIOC->CRH = 0x33333333;
        GPIOD->CRL = 0x33333333;
        AFIO->MAPR |= 0x000010C0;

        //Tim1, Tim4 Setup
         RCC->APB1ENR |= 0x00000004;
        RCC->APB2ENR |= 0x00000800;
        TIM1->CR1 =0x00;
        TIM1->CR2 = 0x00;
        TIM1->PSC = 0x009F;
        TIM1->ARR = 0x116F;
        TIM4->CR1 =0x00;
        TIM4->CR2 = 0x00;
        TIM4->PSC = 0x009F;
        TIM4->ARR = 0x124D;

        TIM4->DIER |= 0x0001;
        TIM1->DIER |= 0x0001;
        NVIC->ISER[0] |= (1<<30);
        NVIC->ISER[0] |= (1<<25);

        TIM1->CR1 |= 0x0001;
        TIM4->CR1 |= 0x0001;

        GPIOA->ODR=0x0000;
        GPIOB->ODR=0xFE1F;
        GPIOC->ODR=0xFFFF;
//초기 셋팅
        Tmove1=100;
        Tmove2=100;
        p1c=0;
        p2c=0;
        Tline1=0;
        Tline2=0;
        p1e1=0;
        p2e1=0;
        Tline_attack2 =0;
        Tline_tmp1 = 0;
        Tline_attack1 =0;
        Tline_tmp2 = 0;
// 가장자리 바운더리 셋팅
        for(i=0;i<17;i++)
        {
                for(j=0;j<19;j++)
                {
                        TData[j][i]=0;
        }
}
        for(i=0;i<18;i++)
        {
                TData[i][0]=1;
                TData[0][i]=1;
                TData[9][i]=1;
                TData[18][i]=1;
                TData[i][17]=1;
        }
}
```

3) 입출력 System – 출력 Part

i) dot matrix 4개를 이용하기 때문에 좌표계를 도입하였다. (16x16) 좌표계를 이용하기 위해서 16x16 Array를 이용하였다. 이 16*16 Array 에 뭔가를 기록하면 그거를 매번 읽어서 그대로 Dot-Matrix 에 출력하는 방식으로 설계를 했으며, 0이면 Dot-Matrix 가 꺼져있고, 1 이면 켜지는 방식으로 기본 시스템을 설계하였다.

ii) Matrix 출력을 고려하면 16*16의 Array를 이용하면 된다. 테트리스 알고리즘을 고려해 양쪽과 상 하단에 가상의 벽을 만들어 18*19 Array 로 수정하였다. 실제로 사용된 Array는 다음과 같으며 벽으로 사용될 부분에 미리 1을 입력하였다.

이때 Dot Matrix 에 출력되는 부분은 1에 둘러쌓인 왼쪽 8*16 과 오른쪽 8*16 이다.
이를 합한 16*16이 Dot Matrix 에 출력된다.



iii) 초기 설계 과정에서 Matrix 의 출력부는 Timer를 이용하여 주기적으로 출력했으나, dot matrix의 밝기 문제 때문에 main의 while문에서 dot matrix출력하도록 설정하였다. Dot-Matrix에 전력을 계속 공급하는 것이 아니라 한 줄당 60hz 이상의 속도로 공급하다보니 전류가 부족하여 흐려지는 증상이 보였고, 이를 해결하기 위해 인터럽트가 걸렸을 때를 제외하고 Dot-Matrix 출력하도록 하였다. 만약 Dot Matrix 에 추가적인 전류를 공급해줄 Driver 가 존재했다면 60Hz의 타이머를 이용하여 구성하는 것이 최적이었을 것이다. 하지만 설계 구성 시 이를 배제하였기에 main 의 while 구문을 이용하여 최대한 신경 쓸 수 있도록 하였다.

iv) Dot Matrix 의 한 Column 씩 출력하도록 하였고, 각 줄의 출력마다 딜레이를 주어서 어느 정도 전류가 공급될 시간을 갖도록 하였다. 이때 Dot Matrix 왼쪽 2개는 Green 이고 오른쪽 2개는 RED 인데, Green 의 밝기가 좀더 어두워서 Green 의 딜레이를 3배 더 주었다.

```
// 이하 Main 함수의 while 구문. LED 출력만 담당한다.
while (1) {
int i;
int j;
int a;
int b;
int k=0xFFFFFFFF;
int r=600;
int g=2200;
// 초록색을 빨강색의 3배 시간 켜기 ( r와 g값)
// 이하 초록색부터 켜기
{
//------------PC8~ PC13 불켜기------
for(j=8;j<14;j++)                                    {
            a=0x00000000;
            k=0xFFFFFFFF;
            k &= 0xFFFEFFFF>>(16-j);
            GPIOA->ODR = a;

            GPIOB->ODR &=0xFE1F;
            GPIOC->ODR = k;
//----------  PA1~PA12 , PB5~8  ------ 세트시작

            for(i=0;i<12;i++)
                        {if(TData[j+2][i+1]==1)
            {a += 1<<(i+1);}  }
            GPIOA->ODR = a;
            a=0x00000000;

            for(i=12;i<16;i++)
            {    if(TData[j+2 ][i+1]==1)
            {a += 1<<(i-7);} }
            GPIOB->ODR |= a;
//---------PA1~PA12 , PB5~8  ------ 세트끝

            for(b=0;b<g;b++);   }

            GPIOC->ODR =0xFFFF;
            GPIOB->ODR &=0xFE1F;
            a=0x00000000;

//-----PB2 라인 불켜기------------

GPIOB->ODR &= 0XFFFB;
//----------  PA1~PA12 , PB5~8  ------ 세트시작
for(i=0;i<12;i++)
            {
                        if(TData[j+2 ][i+1]==1)
            {a += 1<<(i+1);}}
            GPIOA->ODR = a;

            a=0x00000000;
             for(i=12;i<16;i++)
            {            if(TData[j +2][i+1]==1)
                              {a += 1<<(i-7);}}
            GPIOB->ODR |= a;
            //---  PA1~PA12 , PB5~8  ------ 세트끝
            j++;
            for(b=0;b<g;b++);
            GPIOB->ODR |= 0X0004;
            GPIOB->ODR &=0xFE1F;
            a=0x00000000;
//-----PB9 라인 불켜기--------------
            GPIOB->ODR &= 0xFDFF;
//------  PA1~PA12 , PB5~8 - 세트시작
            for(i=0;i<12;i++)
            {            if(TData[j +2][i+1]==1)
            {a += 1<<(i+1);}}
            GPIOA->ODR = a;
            a=0x00000000;
            for(i=12;i<16;i++)
            {if(TData[j+2 ][i+1]==1)  {a += 1<<(i-7);} }
            GPIOB->ODR |= a;
//----- PA1~PA12 , PB5~8  --- 세트끝
            j++;
            for(b=0;b<g;b++);
            GPIOB->ODR |= 0x0200;
            GPIOB->ODR &=0xFE1F;
            a=0x00000000;           }
// 이하 빨간색 켜기
{  //------------PC0~ PC7 불켜기----
for(j=0;j<8;j++)
{           a=0x00000000;
            k=0xFFFFFFFF;
            k &= 0xFFFEFFFF>>(16-j);
            GPIOA->ODR = a;
            GPIOB->ODR &=0xFE1F;
            GPIOC->ODR = k;
//---  PA1~PA12 , PB5~8  ------ 세트시작
            for(i=0;i<12;i++)
{           if(TData[j+1 ][i+1]==1)
                        {a += 1<<(i+1);}  }
            GPIOA->ODR = a;
            a=0x00000000;
            for(i=12;i<16;i++)
            {    if(TData[j +1][i+1]==1)
            {a += 1<<(i-7);}          }
            GPIOB->ODR |= a;
//----------  PA1~PA12 , PB5~8  ------ 세트끝

            for(b=0;b<r;b++);
            GPIOC->ODR =0xFFFF;
            a=0x00000000;
                                    }
//while 의 끝
}

}
```

4) 입출력 System – 입력 Part

i) 입력은 Key-Matrix 의 Key Scan 에 의해서 이루어진다. Key Scan을 구현하는 방법에는 여러 가지 방법이 있는데, 이중 Timer를 이용한 Key Scan 방식을 이용하였다. 이유는 즉 Tetris 에서 동일한 키를 지속적으로 누르고 있을 경우 이에 대해 일정한 간격으로 반응을 해야 하는데, 이를 제어하기에 Timer가 가장 적합하여 Timer 방식을 선택하였다.

ii) Key Matrix를 Scan 하는 방식은 앞서 이론에서 설명하였다. 이와 같은 방식을 Timer 4 의 IRQ_Handler 로 처리하여 받도록 하였다. 주기는 PSC를 160 , ARR을 4686 으로 설정하여 48Hz 의 속도로 작동하게 하였고, 이 수치는 초기에 예상한 5Hz가 너무 느려서 속도를 10배 높여 나온 수치이다. 대략 20ms마다 Key-scan하여 너무 느리지도 빠르지도 않게 게임을 진행할 수 있도록 설계하였다.

```
void TIM4_IRQHandler (void){
GPIOB->ODR |= 0xFC03;
cdata_temp = 0x0000;
c=0;

// KEY 값 받기
// while 문을 이용하여 C값이 4보다 작으면 유지
키가 4x4 이기 때문에 Keyscan 도 4번 루프가 필요
while(c<4){
GPIOB->ODR &=  scanpat[c];
for(keydelay=0;keydelay<10;keydelay++);
cdata_temp = (~GPIOB->IDR)>>12;
cdata =0x000F & cdata_temp;

if(c==0){
if(cdata == 0x0001){
key=1;

c=100;}

else if (cdata == 0x0002) {
key=2;
c=100;                    }
else if (cdata == 0x0004){
                                    key=3;
                                    c=100;
}
else if (cdata == 0x0008){
                                    key=4;
                                    c=100;
}
}
else if(c==1){
if(cdata == 0x0001){                key=5;
                                    c=100;}
else if (cdata == 0x0002) {         key=6;
                                    c=100;}
else if (cdata == 0x0004){          key=7;
                                    c=100; }
else if (cdata == 0x0008){          key=8;
                                    c=100; }
}
else if(c==2){
if(cdata == 0x0001){                key=9;
                                    c=100; }
else if (cdata == 0x0002) {         key=10;
                                    c=100; }
else if (cdata == 0x0004){          key=11;
                                    c=100; }
else if (cdata == 0x0008){          key=12;
                                    c=100; }
}
else if(c==3){
if(cdata == 0x0001){                key=13;
                                    c=100; }
else if (cdata == 0x0002) {         key=14;
                                    c=100; }
else if (cdata == 0x0004){          key=15;
                                    c=100; }
else if (cdata == 0x0008){          key=16;
                                    c=100; }
}
c++;
}
TIM4->SR &= ~0x1;
}
```

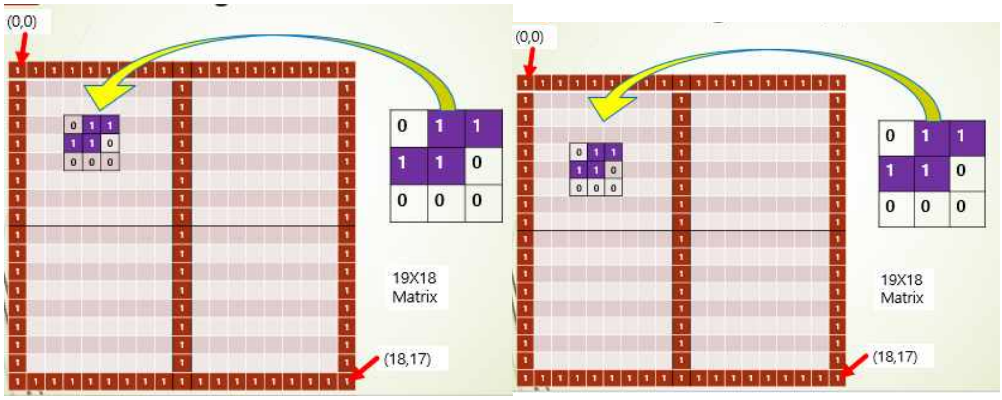
5) Tetris

테트리스 기본 알고리즘은 다음과 같다.

- 테트리스 블록은 3x3 Array에 생성한다.
- Dot-Matrix Array에 테트리스 블록 중심점에 대한 좌표(Tx, Ty) 가 존재하고, 이 좌표에 테트리스 블록 3*3 Array 의 중심부를 맞추어 출력시킨다. 쉽게 생각하면 해당 좌표에 3*3 Array 도장을 찍는다고 보면 된다. 한 바퀴 수행하고나면 이를 지우고 다음 좌표에서 다시 도장을 찍는 식이다.

(Tx-1, Ty-1)	(Tx, Ty-1)	(Tx+1, Ty-1)
(Tx-1, Ty)	(Tx, Ty)	(Tx+1, Ty)
(Tx-1, Ty+1)	(Tx, Ty+1)	(Tx+1, Ty+1)

- 이 경우 도형을 이동할때 방향키에 맞게 Tx, Ty 좌표가 바뀌면 된다.



테트리스 Software 는 크게 생성부, 구동부, 연산부, Game END 4블럭으로 구성되어있다.

i) 생성부 – 블록생성

생성부에서는 테트리스 블록 생성을 담당한다. 위에 언급하였듯이 테트리스 블록은 3*3 에 기록되고, 이 3*3 Array 가 Dot Matrix 위를 돌아다니는 형태이다. 즉 한번 생성하게 되면 3*3 Array 에 기록되고, 재생성 신호가 있을 때 까지 이는 안 변하게 된다.

테트리스가 착지해서 재생성 하라는 신호가 있으면 난수를 이용하여 랜덤으로 블록을 생성한다. 이때 난수는 Timer 의 counter 값을 블록의 개수 7 –1 = 6 으로 나눈 나머지를 이용하였다.

테트리스 블록에는 다음과 같이 7개가 존재하며, 구성의 편의상 막대기 블록은 1*3으로 만들었다.

```
// !P 기준 작성한 소스부분. end 가 아니라면 수행.
if(p1end!=1){
if (Tmove1==100 )
{
// Tmove 1 이 100일 경우 생성하라는 명령이다.

            delay=TIM4->CNT;
            blk=delay- (delay/7)*7+1;
            e=1;
            f=1;

            if(Tmove1==100)
            {

for(for_t1=0;for_t1<3;for_t1++)
                        {

for(for_t2=0;for_t2<3;for_t2++)
                                    {

            Tblk1[for_t1][for_t2]=0;
                                    }
                        }
            if (blk==1){
                        Tblk1[e][f]=1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e-1][f-1]=1;
                        Tblk1[e+1][f]=1;

}
            else if (blk==2){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e-1][f]=1;
                        Tblk1[e+1][f-1]=1;


            }
                        else if (blk==3){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e+1][f]=1;
                        Tblk1[e-1][f]=1;

            }
```

```
}
            else if (blk==4){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e+1][f]=1;
                        Tblk1[e+1][f-1]=1;
                        }
            else if (blk==5){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e][f+1]=1;
                                    }
            else if (blk==6){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e][f+1]=1;
                        Tblk1[e-1][f-1]=1;
            }
            else if (blk==7){
                        Tblk1[e][f] =1;
                        Tblk1[e][f-1]=1;
                        Tblk1[e][f+1]=1;
                        Tblk1[e+1][f-1]=1;
            }
            Trot1=0;
            Tmove1=101;
            Ttype1=blk;
            Tx1=5;
            Txt1=5;
            Ty1=1;
            Tyt1=1;


}
}
// 테트리스 Tx Ty 좌표와 비교를 위한 임시좌표 Txt1 Tyt1 도 초기화 및 Tmove1 은 블록 생성이 끝났다는 시그널 101로 바꿔준다. 블록이 무엇인지 알수 있도록 Ttype 에 blk 값을 기록해 준다.
```

ii) 구동부 – 키 값에 맞춰 Tx, Ty 좌표 이동

구동부에서는 Key Scan에서 받은 키 값에 따라 Tx Ty 좌표를 이동해 준다. 이 좌표의 위치에 3*3 Array 가 합해져서 Dot Matrix Array 가 출력 될 것이다.

이때 테트리스 특성에 맞게 Auto Down 이라는 기능이 있다. 아래 버튼을 누르지 않더라도 일정 시간이 경과하면 자동으로 한 칸씩 아래로 내려가도록 설정한 것이다. 이는 TIM1 ISR의 가장 마지막 부분에서 매번 ++ 해준다. 즉 TIM1 ISR이 10회 진행될 때마다 한번 Auto Down 해주는 것이다.

회전의 경우 Trot 값에 1을 준 다음 연산부에서 회전해도 되는지 확인 후 rotate 한다.

```
// 구동부
if(Tmove1!=100)
{
        Tmove1=key;

        //Auto Down 구문
        if (key !=4 && key!=12 && key!=16)
        {
                                if (autodown1 >10 )
        {       Tmove1=8;
                                key=8;
                                autodown1=0;}
        }

        if(Tmove1==4) { // 왼쪽
                Tx1--;
        }
        else if(Tmove1==8) { //아래
                Ty1++; }
        else if(Tmove1==12) { //오른쪽
                Tx1++;          }
        else if(Tmove1==16)     { //회전
                Trot1=1;
                p=1;
                q=1;



        }
        Tmove1=0;

}
```

iii) 연산부

- Tx, Ty 좌표 기준으로 Array 덧셈을 해서 2가 있는지 확인
- 2가 있을 경우 이전 State로 돌림. 착지해야하는 경우 착지하고 생성부 호출
- 최상단까지 블록이 쌓여있을 경우 END

연산부에서는 이동한 좌표(Tx,Ty) 에 3*3 Array를 출력해도 되는지 계산한다.

앞서 언급하였듯이 3*3 Array 가 있고, Dot Matrix Array(18*19) 가 있다. 출력할 때 Tx, Ty 좌표를 기준으로 하여 해당 위치에 3*3 Array를 출력하게 된다. 이때 출력하는 방식은 단순히 Array를 더하는 방식인데, 이때 해당 위치에 이미 블록이 있어 값이 1이 있을 경우 해당 위치는 2가 될 것이다. 연산부에서는 이 2가되는 경우들을 감지하고, 해결하면 된다.

보통의 경우에는 2가 되면 이전 state 로 돌리면 해결된다. 하지만 밑으로 가던 중 원래 쌓여 있던 블록을 만나 결과가 2가 된다면 이를 인지하고 착지를 시도해야한다. 그래서 2가 발견되었을 때 이게 Down 인지 아닌지 인지해서 Down 일 경우 이전 state 로 돌린 다음 Dot Matrix 에 기록(착지)하고 생성부에 Tmove1=100 (생성하라는 명령)을 준다.

그 외는 2가 될 경우 하나 이전 state 로 돌린다. 연산부에서 연산을 다 하고 interrupt를 빠져 나오고 나서야 Dot Matrix 출력이 되기 때문에 사람 눈에는 그 과정까지는 안 보이고 블록이 움직이지 않는 것으로 보이게 된다.

Dot matrix Array (18x19)의 값과 Block Array(3x3)를 더하여 2가 되는 경우 중 블록이 생성되는 구역에서 합이 2가 된다면 블록 생성구역까지 블록이 이미 쌓여있는 것이기 때문에, 더 이상 게임은 진행되지 않고 종료된다.

회전의 경우에는 3*3 Array 상에서 90도 회전하는 것으로 한다. 블록을 회전하고 나서 dot matrix와 block array의 합이 2가 될 수 있다. 따라서 이때도 마찬가지로 연산부에서 확인해보고 2가 생길 경우 회전을 무효화 시켜야 한다.

```
// Tetris 연산부


{
if(key==4 || key==8 || key==12 || key==16 || Tmove1==101 )
{
// 3x3 도장 찍은거 지우기 ( 블록을 이동시켜야하니 이전 출력 지우기 (블럭에 관해서만))
{
if (Tdel1==77 )
{                       for(for_t1=0;for_t1<3;for_t1++)
{
            for(for_t2=0;for_t2<3;for_t2++)
            {
                        if(TData[Txt1-1+for_t1][Tyt1-1+for_t2]>0)
                        {TData[Txt1-1+for_t1][Tyt1-1+for_t2] -= Tblk1[for_t1][for_t2];
                        }
            }
            }
}
}



if ( Trot1==1)  // Rotation 해도 도는지 확인
{
temp1=Tblk1[p+1][q-1];//save3
temp2=Tblk1[p+1][q];//save4

Tblk1[p+1][q-1]=Tblk1[p-1][q-1];//3<-1
Tblk1[p+1][q]=Tblk1[p][q-1];//4<-2

Tblk1[p][q-1]=Tblk1[p-1][q];//2<-8;
Tblk1[p-1][q-1]=Tblk1[p-1][q+1];//1<-7

Tblk1[p-1][q]=Tblk1[p][q+1];//8<-6;
Tblk1[p-1][q+1]=Tblk1[p+1][q+1];//7<-5;

Tblk1[p+1][q+1]=temp1; //5<-3(temp1)
Tblk1[p][q+1]=temp2; //6<-4 (temp2)
//회전 state 입력

}

//            이동해도 되는지 확인
for(for_t1=0;for_t1<3;for_t1++)
{
for(for_t2=0;for_t2<3;for_t2++)
{
TData[Tx1-1+for_t1][Ty1-1+for_t2] += Tblk1[for_t1][for_t2];
if (TData[Tx1-1+for_t1][Ty1-1+for_t2] ==2)
{
            Talert1=999;        // 문제가 있을 경우 alert 신호를 띄운다.
}
}

}
```

```
// 좌표 중첩 (2되는거) alert 가 뜨면
if(Talert1==999)
{
                                        for(for_t1=0;for_t1<3;for_t1++)
{
for(for_t2=1;for_t2<3;for_t2++)
{
            if(TData[4+for_t1][for_t2]==2){
                        p1end=1;              // 좌표 중첩이 시작부분에서 있는 경우 Ending
            }
}
}
            for(for_t1=0;for_t1<3;for_t1++)
{
            for(for_t2=0;for_t2<3;for_t2++)
            {
                        TData[Tx1-1+for_t1][Ty1-1+for_t2] -= Tblk1[for_t1][for_t2];
                        }
            }
// 화면 복구는 했고
            // 이제 이전 state 로 돌아가자

if ( Trot1==1)      // rotation 에 대한 복구가 필요한 경우
{
                        temp1=Tblk1[p+1][q-1];//save3
             temp2=Tblk1[p+1][q];//save4

             Tblk1[p+1][q-1]=Tblk1[p+1][q+1];//3<-5
             Tblk1[p+1][q]=Tblk1[p][q+1];//4<-6

            Tblk1[p+1][q+1]=Tblk1[p-1][q+1];//5<-7
             Tblk1[p][q+1]=Tblk1[p-1][q];//6<-8

             Tblk1[p-1][q+1]=Tblk1[p-1][q-1];//7<-1
             Tblk1[p-1][q]=Tblk1[p][q-1];//8<-2

             Tblk1[p-1][q-1]=temp1;//1<-3(temp1)
             Tblk1[p][q-1]=temp2;//2<-4(temp2)
                        Trot1=0;
}
else // 이동을 복구해야 하는 경우
{
                        Tx1= Txt1;
                        Ty1= Tyt1;

}}
```

```
// 다시 새로운 화면을 뿌려줌
for(for_t1=0;for_t1<3;for_t1++)
                        {
                                for(for_t2=0;for_t2<3;for_t2++)
                                {
                                        TData[Tx1-1+for_t1][Ty1-1+for_t2] += Tblk1[for_t1][for_t2];

                                }
                        }

                        Talert1=0;
                        Tdel1=77;

if(key==8)     // 만약 내려오는 도중 (Key==8) 충돌이 감지된거였으면 생성부에 생성명령을 준다 (Tmove1=100)
{
                        Tmove1=100;
}

}
else
{
Txt1=Tx1;
Tyt1=Ty1;
Tdel1=77;
}

// xy Temporary 좌표를 최신화한다. 다음번 loop에서 비교하려면 temporary 에 값을 넣어놔야 비교가 가능하다.



//
Trot1=0;

}
/// 사실상 연산종료

}
```

iv) Line Clear 감지 및 공격

매 Loop 마다 Line clear 된 것이 있는지 찾아본다. 이 구문은 연산부 뒤에 존재하게 되는데, 블록을 착지 혹은 이동시킨 다음에 확인해야 동작의 정확성이 높아지기 때문이다. 원리는 단순한 편이다. 한 줄에 있는 값들을 다 더해서 8일 경우 그 Line의 모든 자리에 1이 있는 것이기 때문에 Line Clear 가 되고 그 줄을 없앤 다음 stage의 모든 블록을 세로로 한 칸씩 내린다.

Line Clear가 되면 다 되면 상대 플레이어에게 공격이 진행된다. 1줄 공격은 현재 Dot Matrix 에 있는 모든 것을 하나 위로 올리는 방식으로 구현하였다. Tx, Ty 좌표도 올리고, 모든 좌표를 다 하나씩 올린다음 가장 밑에 하나의 빈칸이 뚫린 새 Line을 추가하도록 설계하였는데, 이 한 칸은 위에서 사용한 Counter 난수 방식을 이용하여 임의의 위치에 뚫리도록 하였다.

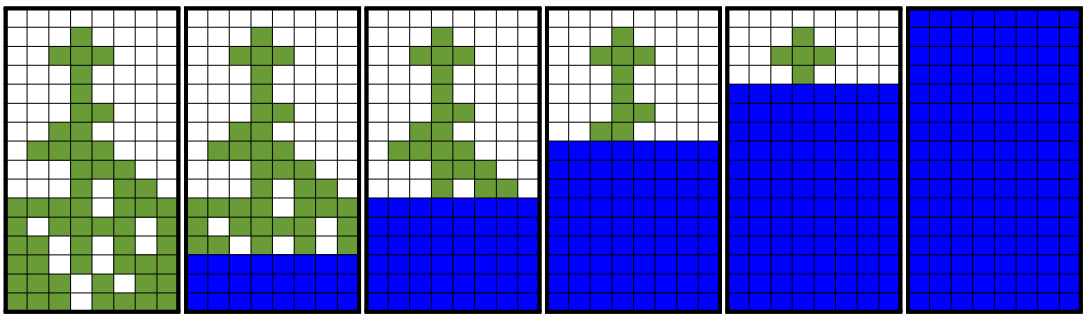
```
// 1P  clear 감지
if(p1end!=1){
if(key==8 && Tmove1==100)
{
for (j=16;j>0;j--)        // 어떤 한 줄에 대하여
{
            {
            for(i=1;i<9;i++)                        // x값을 다 확인해본다.
            {
                        Tline_tmp1 += TData[i][j];
            }

            if( Tline_tmp1 ==8)  // 합한결과 8이면 라인 clear
            {
                                    for(i=1;i<9;i++)                        // 해당 라인 0 으로 초기화해주고
                        {
                                                TData[i][j]=0;
                        }

                        // 해당 라인 윗부분 다 땡겨온다.
                        for(k=j;k>1;k--)
                        {
                                    for(i=1;i<9;i++)
                                    {
                                                            TData[i][k] = TData[i][k -1];
                                    }
                        }

                        Tline1 ++;
                        Tline_attack2 ++;        // 상대방한테 Attack 한다
                        j++;
            }
            Tline_tmp1 = 0;
}                       // 라인 clear if문 닫기
            }
}
}

// 2P 가 1P 에게 공격
if(p1end!=1){
if(Tline_attack1>0)
{
            for(j=1;j<16;j++)
            {
                                    for(i=1;i<9;i++)
                        {TData[i][j]=TData[i][j+1];}
            }
                        for(i=1;i<9;i++)
            {           TData[i][j]=1;            }
                        TData[1+blk2][j]=0;
            Tline_attack1 --;
            Ty1--;
            Tyt1--;
}
}
```

v) End

앞서 3x3 블록의 도장을 찍고 지우는 과정을 반복하는데, 더한 값이 2가 되면 이전 state로 돌아가도록 설정하였다. 이 때, 블록의 생성 위치에 이미 다른 블록들이 쌓여 있어서(=1) 블록의 생성 위치에 3x3블록 도장을 찍었을 때 더한 값이 2가 되면 이전 state로 돌아가게 되면 블록은 생성되지 않으므로 Game Over로 설정하였다. 블록의 생성 위치는 1p에서 (5,1)이고, 2p에서 (14,17)인데 중심부 기준으로 x±1, y±1 의 위치에 1이 있어 블록이 생성될 때 합이 1이 된다면 end를 선언한다.

End 의 과정은 다음과 같다.

a) 가장 하단부터 상단을 채우기

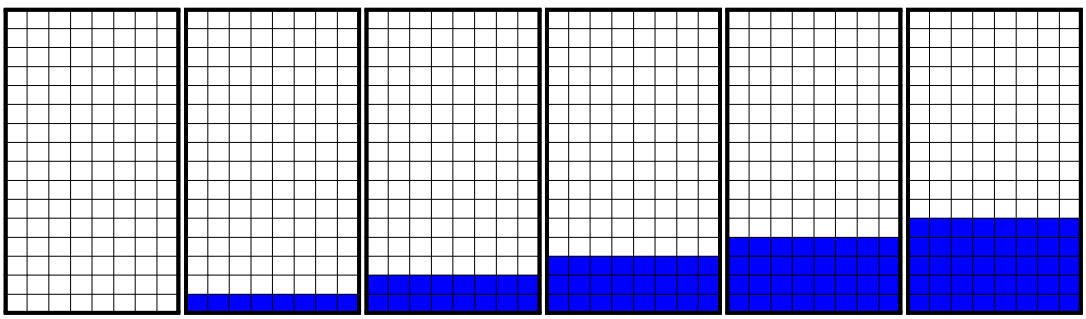


왼쪽 그림을 보면 이제 다음 블록이 생성되어도 블록 생성부에 이미 쌓여있는 블록이 있기 때문에 Game Over가 된다. 이후 가장 하단부터 모든 라인이 채워지고 최종적으로 모든 Line이 채워진다. 그림에서는 3Line씩 채워지게 해 놓았지만, 실제로는 1Line씩 채워진다.

if(p1e1==0){
if(p1c<16){
for(for_t3=1;for_t3<9;for_t3++){
TData[for_t3][16-p1c]=1;}
p1c++;}
else if(p1c==16){p1c=0;
p1e1++;}
	16개의 Line을 채우는 과정이다. p1e1은 1p의 종료 시퀀스 과정의 변수이며, p1c는 16개 Line을 모두 채우기 위한 변수이다. 다음 과정으로 넘어가기 위해 p1e1++를 수행한다.

b) Stage Clear 및 Clear Line 표시

모든 Line이 다 채워지면 게임 종료가 되었음을 알리고, 이제 Stage를 Clear한 뒤 Clear한 Line 개수를 표시한다. 만약 5개의 Line을 Clear하였다면,



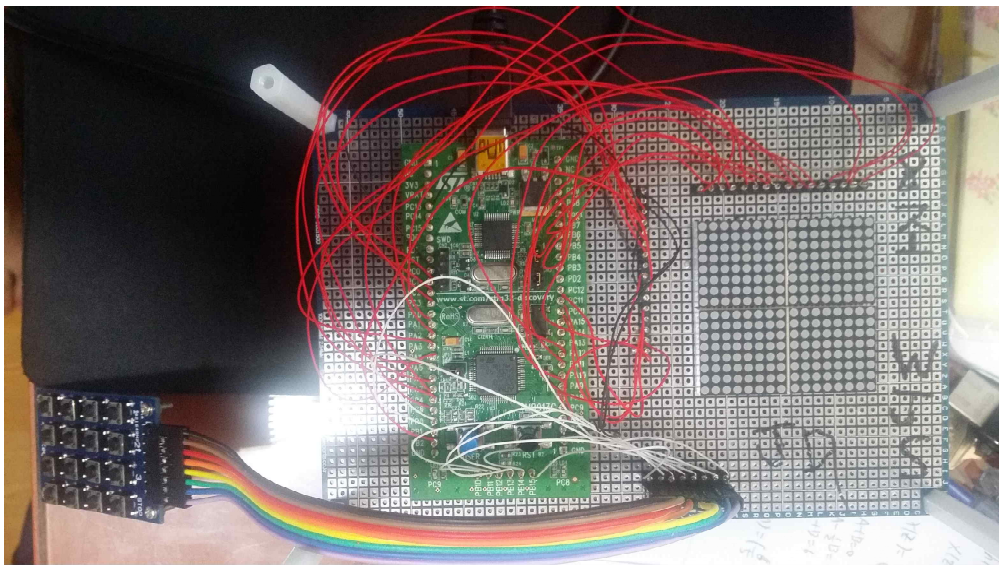
다음과 같은 시퀀스가 진행되고, 결과를 통해 5개의 Line을 Clear하였음을 알 수 있다.

else if (p1e1==1){
for(for_t1=1;for_t1<9;for_t1++){
for(for_t2=1;for_t2<17;for_t2++){
TData[for_t1][for_t2]=0;}
}
p1e1++;
}
else if (p1e1==2){
if(p1c<Tline1){
for(for_t3=1;for_t3<9;for_t3++){
TData[for_t3][16-p1c]=1;}
p1c++;
}
}
	p1e1=1 이면 8x16 stage를 모두 Clear한다. 이 과정이 완료되면 p1e1++하여 다음 과정으로 넘어간다.
	p1e1=2이면 Clear한 Line만큼 1로 채운다. p1e1=0일 때와 코드가 동일한데, p1c가 TLine1보다 작을 때 동작한다. TLine1은 1p가 Clear한 Line의 개수에 대한 변수이다. 즉 Clear한 Line 개수만큼 하단부터 상단으로 1로 채우겠다는 의미이다.

2P도 동일하게 진행되나, 방향만 반대로 해서 진행한다.

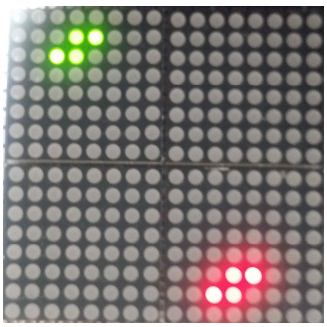
5. 결과 확인

(1) 설계물 전체 사진



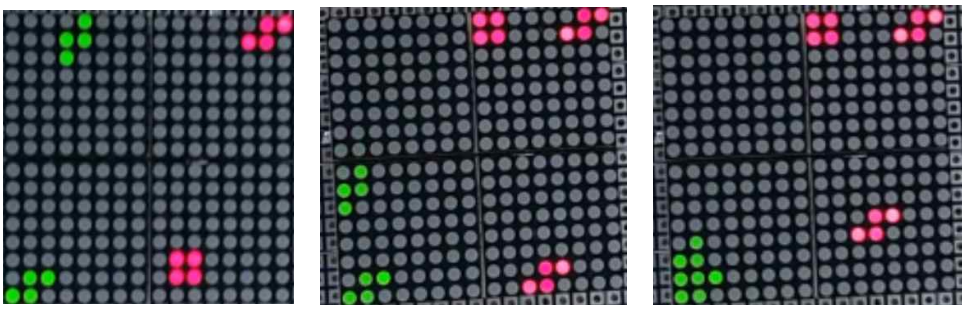
8x8 dot matrix 4개를 정사각형 모양으로 연결하였다. 함께 PCB위에 4x4 keypad와 프로세서가 연결되어 있다.

(2) Tetris 동작



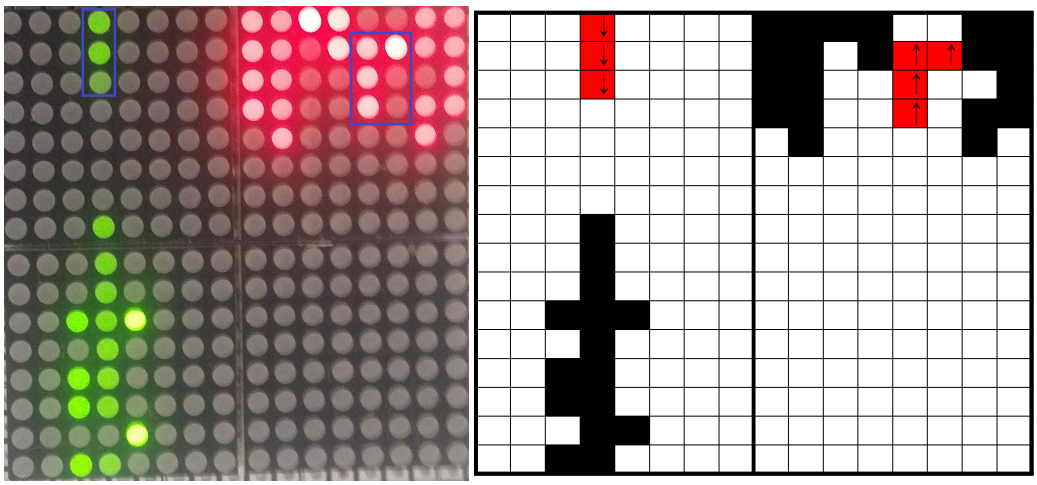
초기 Reset 상태이다. 1p와 2p의 블록 진행 방향을 반대로 했기 때문에, 1p는 위쪽에서, 2p는 아래쪽에서 블록이 생성된다. 4x4 키패드에서 1p와 2p의 버튼을 반대방향으로 설정해 두었으므로 이렇게 동작하면 플레이어들이 게임 화면을 더 수월하게 볼 수 있다.

2-1) 블록 쌓기

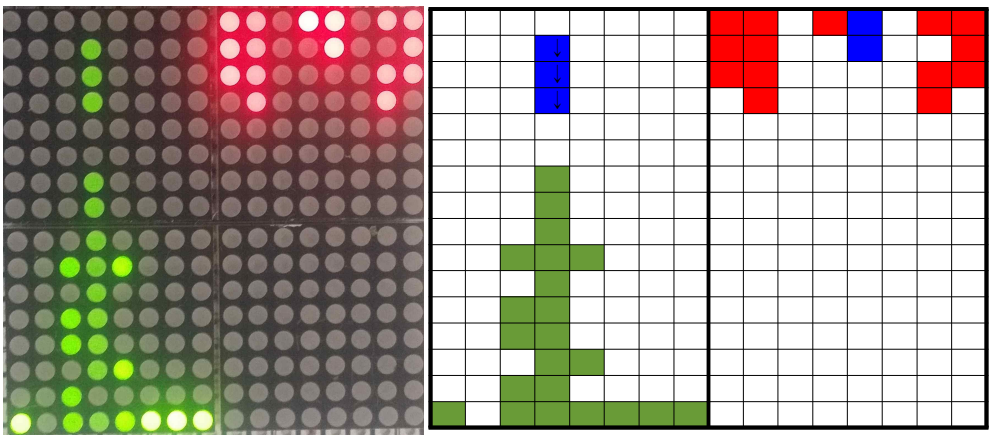


블록을 쌓는 과정이다. 왼쪽 그림을 보면 생성된 블록을 가장자리에 배치하였음을 알 수 있다. dot matrix가 16x16으로 구성되어 있지만, 프로그램 상에서 각 플레이어들의 stage (8x16)은 벽(=1)으로 둘러 쌓여 있도록 구성했기 때문에, 다른 플레이어의 stage로 블록을 이동시킬 수 없고, 가장자리로 블록을 이동시킬 수 없다. 블록이 이동하다가 stage의 가장 아래 밑바닥에서 블록은 정지되고 해당 위치에 블록의 존재를 의미하는 1을 Write한다. 이와 같은 과정을 통해 블록을 쌓을 수 있다. 블록은 Down 버튼을 누르지 않더라도 일정 시간 간격으로 각 player stage의 끝부분으로 향하게 된다.

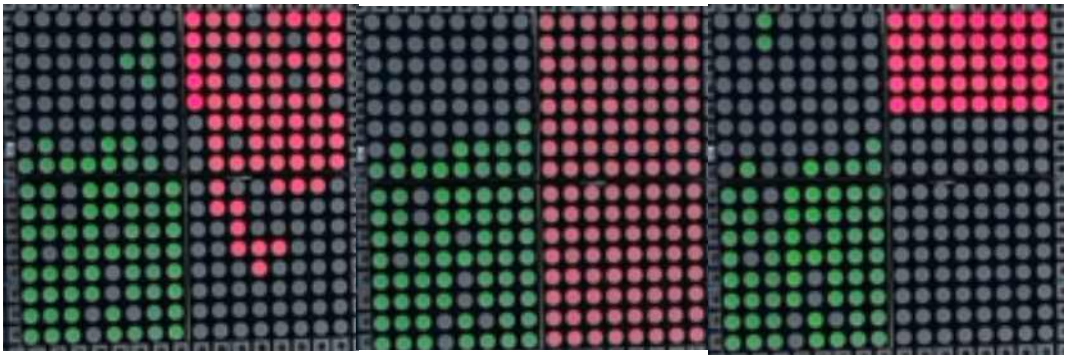
2-2) Line Clear & Attack

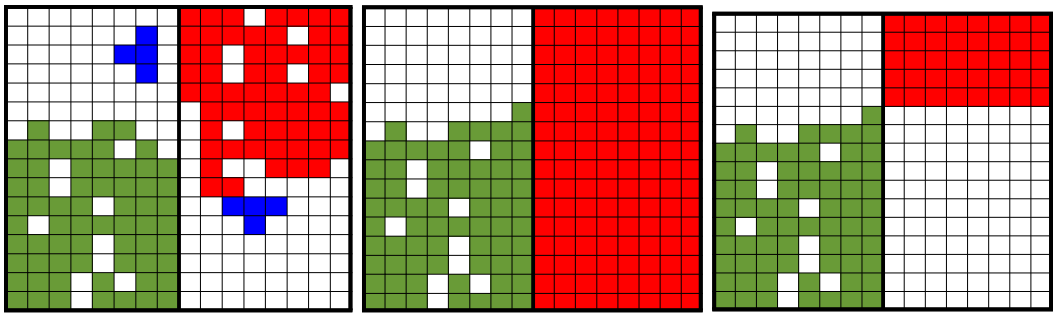


왼쪽 그림의 상황을 오른쪽 표로 나타내었다. 현재 2p (Red)쪽에서 Line Clear를 진행하려고 한다. 화살표 방향으로 블록이 진행되므로 이제 Line이 Clear될 것이다.



Line Clear가 완료되었다. 2p의 Line이 한 칸 내려가고, 1p의 stage에서 빈 공간이 하나 있는 Line이 추가되었다. 플레이어들은 Line Clear를 통해 다른 플레이어를 공격할 수 있다.





왼쪽 그림을 보면 이제 2p(우측) 진영은 위험한 상태이다. 결국 2p의 게임은 끝났고, 가운데 그림과 같이 Stage가 모두 채워진다. 이후 오른쪽 그림을 통해 Clear한 Line의 개수를 출력한다. 2p의 게임 종료 전까지 5Line을 Clear했음을 알 수 있다.

6. 한계 사항

(1) 2 color matrix 4개를 사용하여 테트리스를 구현하였다. 1p와 2p의 구분을 위해 1p를 적색, 2p를 녹색을 이용하여 출력하도록 했는데, 이렇게 설계할 경우 2개의 1 color dot matrix를 이용하고, 2개는 2 color dot matrix를 이용하면 되었는데 그러지 않았고 결국 2개의 2 color dot matrix의 낭비로 이어졌다고 볼 수 있다.

(2) 앞서 블록 생성에 대하여 3x3 좌표계를 생성하여 좌우하 움직임 및 회전을 구현하였다. 그런데, 여기에 1x4의 막대모양 블록을 추가하려고 하면 4x4 좌표계를 새로 만들어야 하고, 여기서는 중심점이 없기 때문에 회전에 대한 예외사항이 많아져서 막대 모양 블록을 1x3으로 대체하였다.

(3) 초기에 테스트 했을 때 Dot Matrix 불을 다 켜면 밝아서 몰랐는데, 이를 60Hz 의 속도로 한 줄씩 번갈아가면서 출력하였더니, 전류가 부족하여 어두워지는 현상이 발견되었다. 엄밀하게 얘기하면 전류가 낮더라도 공급시간이 길면 평균전력이 높아져 괜찮은데, Dot Matrix 의 점을 개별 Control하기 위해선 한 줄씩 번갈아가면서 출력할 필요가 있었고, 이 때문에 한 줄에 전류가 공급되는 시간이 짧아져 밝기가 어두워지는 문제가 있었다.
특히 초록색의 경우 같은 시간 공급 시 더 어둡게 보여서 문제였다.

그래서 이렇게 해결하였다.

1. 우선 테트리스를 서로 마주보고 하도록 설계하였다. 한 줄 한 줄 쌓이는데, 두 명의 플레이어 다 맨 밑줄에 쌓여있을 경우 한 줄이 불이 다 들어와야 하는데, 그러면 동일한 전류를 16개의 Dot 이 나누어 가져야해서 너무 어두워졌다. 그래서 서로 마주보고 하도록 설계하였다.
2. 이렇게 설계하여도 초록이 빨강에 비해 어둡게 나오는 문제가 있었다. 그래서 출력을 Row 단위가 아닌 Column 단위로 출력하게 바꾸었다. 즉 Column 단위이기 때문에 초록색 불끼리 한줄 켜지고 빨간색 끼리 한줄 켜지게 되었다. 이렇게 한 다음 초록색에선 빨강색보다 3배정도 더 전류공급시간을 갖도록 하였다. (단순 For문을 이용.)

이렇게 하면 평소에는 밝게 잘 나오는데, 어느 부분에 블록이 2/3 넘어가게 쌓일 경우 점점 어두워지는 문제가 있었다. 이를 완전히 해결하기 위해선 전류 공급 시간을 늘려주거나 아니면 전류자체를 한번에 많이 공급해 주어야 하는데, 시간을 늘릴 수는 없으니 별도로 외부 Driver를 이용하여 전류를 늘려주어야 한다.

7. 결론

이번 텀 프로젝트의 목표는 10주 동안 배운 내용을 토대로 제한된 조건 내에서 최대한 설계를 해보는 것이 목표였다. 제한된 조건이라 하면 입출력 장치가 Dot Matrix 와 Key Matrix 뿐이라는 것이었고, 10주 동안 배운 내용 중 핵심내용들인 GPIO, Timer, Interrupt를 이용하였다.

처음에 테트리스를 목표로 했을 때 금방 할 줄 알았는데, 테트리스 알고리즘 뿐만 아니라 메인 키트도 구성해야 되어서 생각보다 어렵고 시간도 많이 걸렸다. 특히 Dot Matrix 경우 Data Sheet를 봐도 헷갈려서 어떻게 되는 건지 파악하는데 시간이 좀 걸렸었고, Key Matrix 또한 Debug 모드를 시뮬레이터로 해놓고 안 되는 이유를 파악하는 바람에 시간이 조금 걸렸었다. 하지만 두 경우 다 시행착오를 겪으면서 좀 더 원론적인 내용에 다가갈 수 있는 좋은 시간이 되었었고, Dot Matrix 와 Key Matrix를 구성하며 확실히 마이크로 프로세서에 대한 장벽이 낮아지는, 좀더 친근해 지는 느낌을 받았다.

테트리스 알고리즘이야 생각한 것을 C언어로 변환하는 것이었기에 힘들지 않게 구현할 수 있었다. 물론 처음에 뜻하던 대로 작동하지 않아 아예 알고리즘을 바꾸는 일이 생겼지만 이것 또한 값진 경험이 되었다고 생각한다. 실제로 좌표계의 도입 경우 처음에 생각한게 1학년 C언어 프로젝트 때인데, 그때의 경험이 현재에 영향을 미치는 것을 보면 요번에 마이크로 프로세서 프로젝트를 통해 배운 것들이 미래의 나에게 영향을 미치리라 생각해 볼 수 있다.

프로젝트를 성공적으로 마무리하면서 무엇이던 원하는 것을 만들 수 있다는 자신감이 생겼다. 처음에 배울때만 해도 이게 무슨 소리인지 이해하기가 힘들었던 과목이었는데, 어느 덧 원하는 것도 만들고, 관련 이론들도 확실하게 정립할 수 있는 좋은 시간이었던 것 같다. 전자공학도의 길에 한걸음 더 다가갈 수 있는 과목이었고, 프로젝트였다고 생각한다.

8. 참고문헌

[1] STMicroelectronics, “STM32F10X Reference manual”
[2] 김경환, “마이크로프로세서개론” 강의교재, 서강대학교, 2016
[3] STMicroelectronics, STM32VLDISCOVERY Manual
[4] SZ010788 Dot Matrix DataSheet