|  |
| --- |
| **마프2 작품계획서**   * **가속도 센서와 GLCD를 이용한 레이싱 게임 -**   날짜 : 2019 04 24  이름 : 강승우  학번 : 2014161001 |
|  |

1. **작품명**  
   **가속도 센서와 GLCD를 이용한 레이싱 게임**

**2. 작품 개요 및 동작**

실제 핸들을 다루는 것처럼 손잡이를 돌려 조향할 수 있는 레이싱 게임입니다. 가속도 센서를 이용해서 핸들의 기울어진 정도를 읽어오고, 이를 바탕으로 GLCD(주 디스플레이)를 부착한 서보 모터를 회전시켜 화면의 평형을 유지합니다.

레이싱 게임의 가속과 감속은 아날로그 감압 센서를 이용해서 세밀하게 조정 가능하게끔 합니다.

실제 게임은 X, Y 평면상에서 이루어지며, 정점으로만 구성된 벡터 그래픽을 사용합니다. 거리에 따라 각 정점의 오프셋을 스케일링함으로써 원근감을 구현하게 됩니다.

**3. 사용 포트 및 부품**

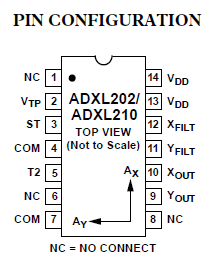
**<사용 부품 리스트>**

|  |  |  |  |
| --- | --- | --- | --- |
| **부품명** | **규격** | **수량** | **기능** |
| KUT-128 | Pin Header | 1 | 메인 MCU 보드 |
| GLCD | LG2401283, 240x128 | 1 | 주 디스플레이 |
| 서보 모터 | HES-288 | 1 | 화면 평형 유지 |
| 가속도 센서 | ADXL202JQC, SMD | 1 | 기울기 감지 |
| SRAM | IS62C256AL-45ULI | 1 | 추가 기억장치, 32KB |
| 스피커 | FQ-031 | 2 | SFX 출력 |
| 압력 센서 | FSR, RA12P | 2 | 엑셀, 브레이크 아날로그 입력 |
| 8비트 D래치 | 74573 | 1 | SRAM 연결용 |
| 정전압 레귤레이터 | LM3940 | 1 | LCD 바이어스용 |

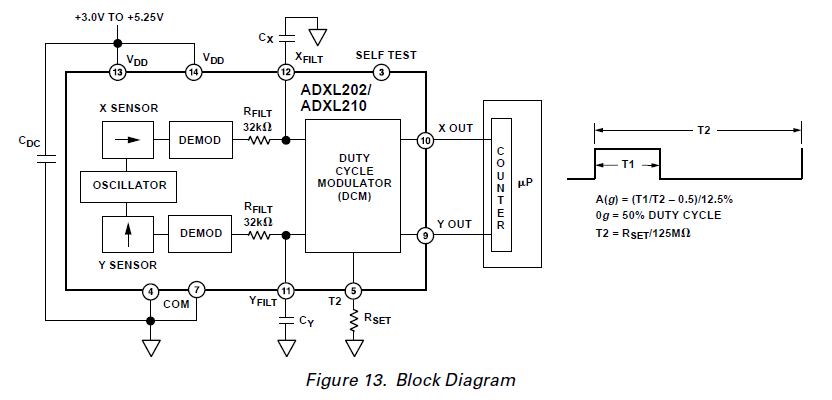
**<사용 포트> : 핀 단위**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **사용 포트** | **특수 기능** | **입/출력** | **연결 부품(핀)** | **기능** |
| PA0 | AD0 | 입출력 | 74573 D0 | SRAM DATA0/ADDR0 |
| PA1 | AD1 | 입출력 | 74573 D1 | SRAM DATA1/ADDR1 |
| PA2 | AD2 | 입출력 | 74573 D2 | SRAM DATA2/ADDR2 |
| PA3 | AD3 | 입출력 | 74573 D3 | SRAM DATA3/ADDR3 |
| PA4 | AD4 | 입출력 | 74573 D4 | SRAM DATA4/ADDR4 |
| PA5 | AD5 | 입출력 | 74573 D5 | SRAM DATA5/ADDR5 |
| PA6 | AD6 | 입출력 | 74573 D6 | SRAM DATA6/ADDR6 |
| PA7 | AD7 | 입출력 | 74573 D7 | SRAM DATA7/ADDR7 |
| PC0 | A8 | 출력 | IS62C256AL AD8 | SRAM ADDR8 |
| PC1 | A9 | 출력 | IS62C256AL AD9 | SRAM ADDR9 |
| PC2 | A10 | 출력 | IS62C256AL AD10 | SRAM ADDR10 |
| PC3 | A11 | 출력 | IS62C256AL AD11 | SRAM ADDR11 |
| PC4 | A12 | 출력 | IS62C256AL AD12 | SRAM ADDR12 |
| PC5 | A13 | 출력 | IS62C256AL AD13 | SRAM ADDR13 |
| PC6 | A14 | 출력 | IS62C256AL AD14 | SRAM ADDR14 |
| PG0 | /WR | 출력 | IS62C256AL /WE | SRAM WRITE EN |
| PG1 | /RD | 출력 | IS62C256AL /OE | SRAM OUTPUT EN |
| PG2 | ALE | 출력 | 74573 LE | 데이터 / 어드레스 전환 |
| PD0 |  | 출력 | LG2401283 D0 | GLCD D0/D4 |
| PD1 |  | 출력 | LG2401283 D1 | GLCD D1/D5 |
| PD2 |  | 출력 | LG2401283 D2 | GLCD D2/D6 |
| PD3 |  | 출력 | LG2401283 D3 | GLCD D3/D7 |
| PD4 |  | 출력 | LG2401283 WR0 | GLCD WRITE CLK |
| PD5 |  | 출력 | LG2401283 CD | GLCD I/D SELECT |
| PF0 | ADC0 | 입력 | FSR 0번 | 아날로그 압력 입력 0 |
| PF1 | ADC1 | 입력 | FSR 1번 | 아날로그 압력 입력 1 |
| PF2 |  | 입력 | ADXL 202 XOUT | 가속도 X축 PWM 입력 |
| PF3 |  | 입력 | ADXL 202 YOUT | 가속도 Y축 PWM 입력 |
| PB4 | OC0 | 출력 | SPK 0 | 0번 스피커 주파수 변조 출력 |
| PB7 | OC2 | 출력 | SPK 1 | 1번 스피커 주파수 변조 출력 |
| PB6 |  | 출력 | HES-288 | 서보모터 PWM 제어신호 |
| PE0 |  | 입출력 | 출력 커넥터 | 예약 |
| PE1 |  | 입출력 | 출력 커넥터 | 예약 |
| PE2 |  | 입출력 | 출력 커넥터 | 예약 |
| PE3 |  | 입출력 | 출력 커넥터 | 예약 |
| PE4 |  | 입출력 | 출력 커넥터 | 예약 |
| PE5 |  | 입출력 | 출력 커넥터 | 예약 |
| PE6 |  | 입출력 | 출력 커넥터 | 예약 |
| PE7 |  | 입출력 | 출력 커넥터 | 예약 |

**<사용 부품 사양서>**

 ADXL202 – 디지털 출력 가속도 감지 센서

|  |  |  |
| --- | --- | --- |
| PIN | NAME | DESCRIPTION |
| 1 | NC | No Connect |
| 2 | VTP | Test Point, do not connect |
| 3 | ST | Self Test |
| 4 | COM | Common |
| 5 | T­­2 | Connect R­SET to Set T2 Period |
| 6 | NC |  |
| 7 | COM |  |
| 8 | NC |  |
| 9 | YOUT | Y Axis duty cycle output |
| 10 | XOUT | X Axis duty cycle output |
| 11 | YFILT | Connect capacitor for Y filter |
| 12 | XFILT | Connect capacitor for X filter |
| 13 | VDD | +3V to +5.25V, Connect to 14 |
| 14 | VDD | +3V to +5.25V, Connect to 13 |

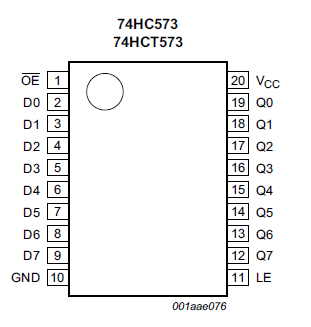
T1과 T2포트의 Duty Ratio를 통해 아날로그 측정값을 디지털로 출력하는 가속도 센서이다. 일반적인 입력 포트에 연결한 뒤, 카운터로 일정 주기 동안의 1의 개수를 세는 방식으로 Duty Ratio를 역산할 수 있다.

주기 T2와 필터 대역폭을 산정하기 위한 저항과 캐패시터 값은 아래와 같다.

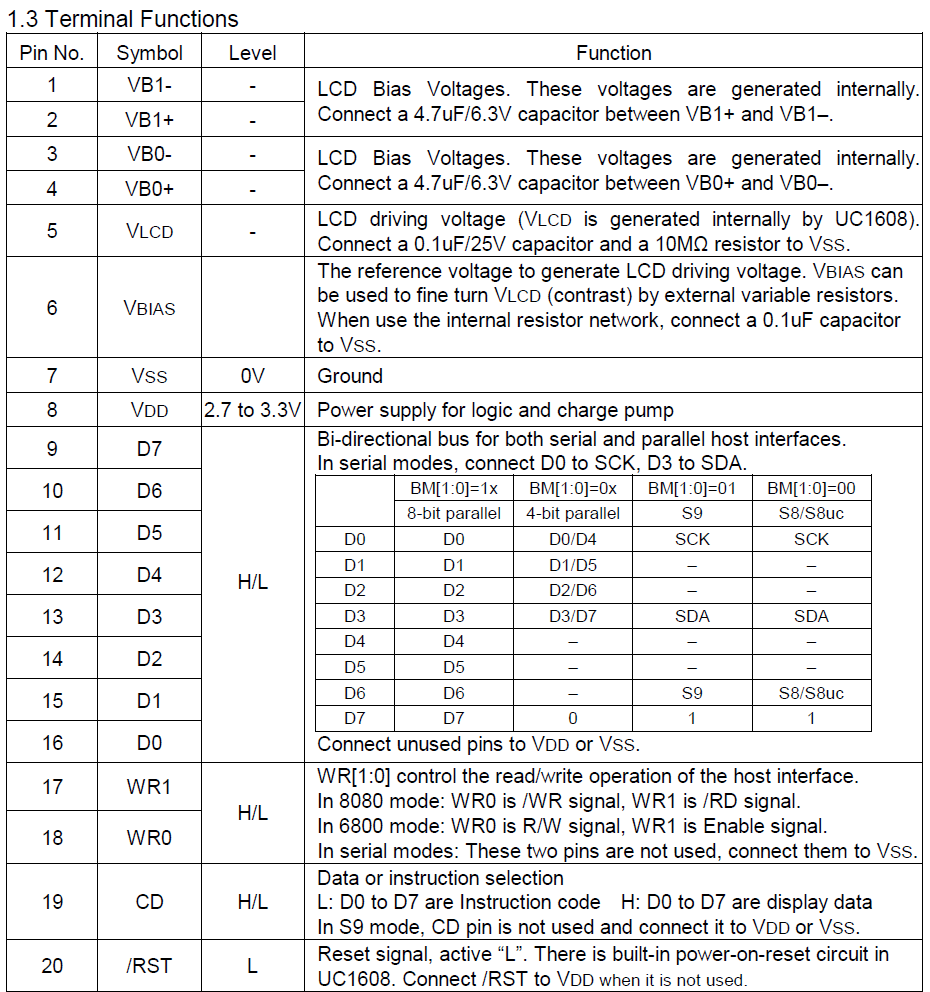
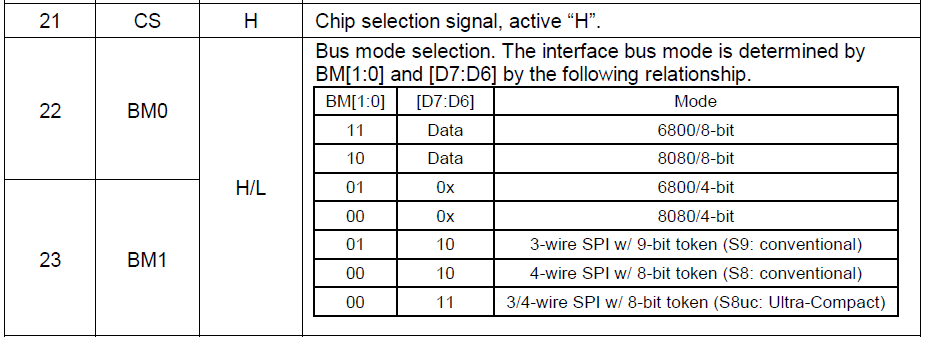
|  |  |
| --- | --- |
| T2 | RSET |
| 1 ms | 125 kOhm |
| 2 ms | 250 kOhm |
| 5 ms | 625 kOhm |
| 10 ms | 1.25 MOhm |

|  |  |
| --- | --- |
| Bandwidth | Capacitor Value |
| 10 Hz | 0.47 uF |
| 50 Hz | 0.10 uF |
| 100 Hz | 0.05 uF |
| 200 Hz | 0.027 uF |
| 500 Hz | 0.01 uF |
| 5 kHz | 0.001 uF |

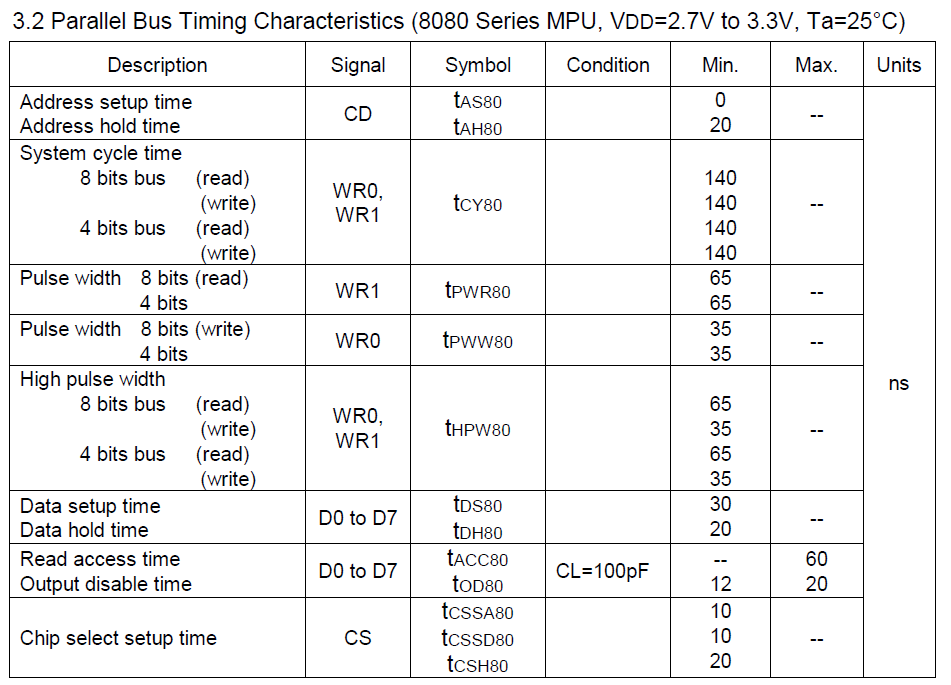
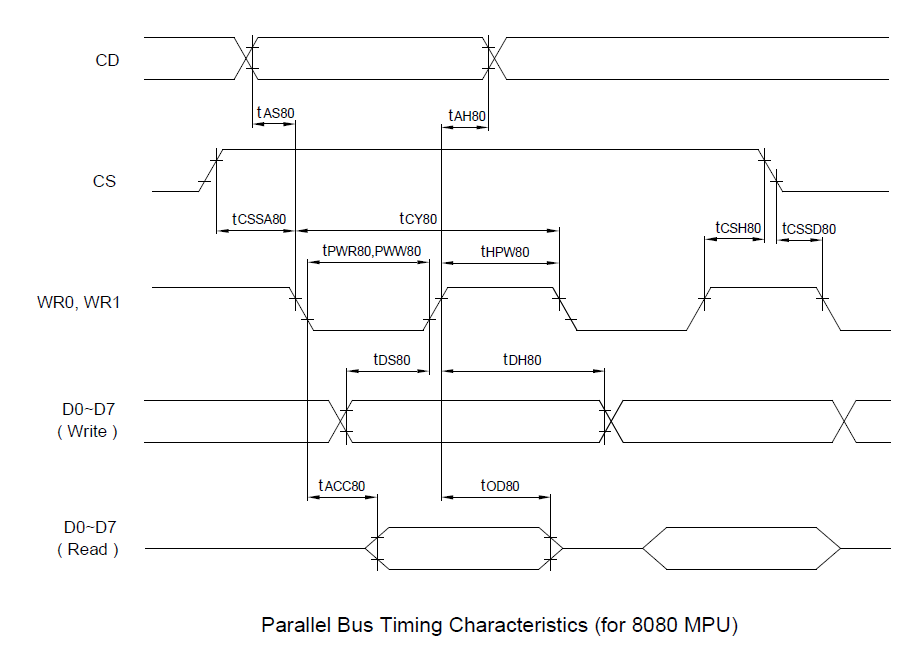
74573 IC – 8BIT D LATCH

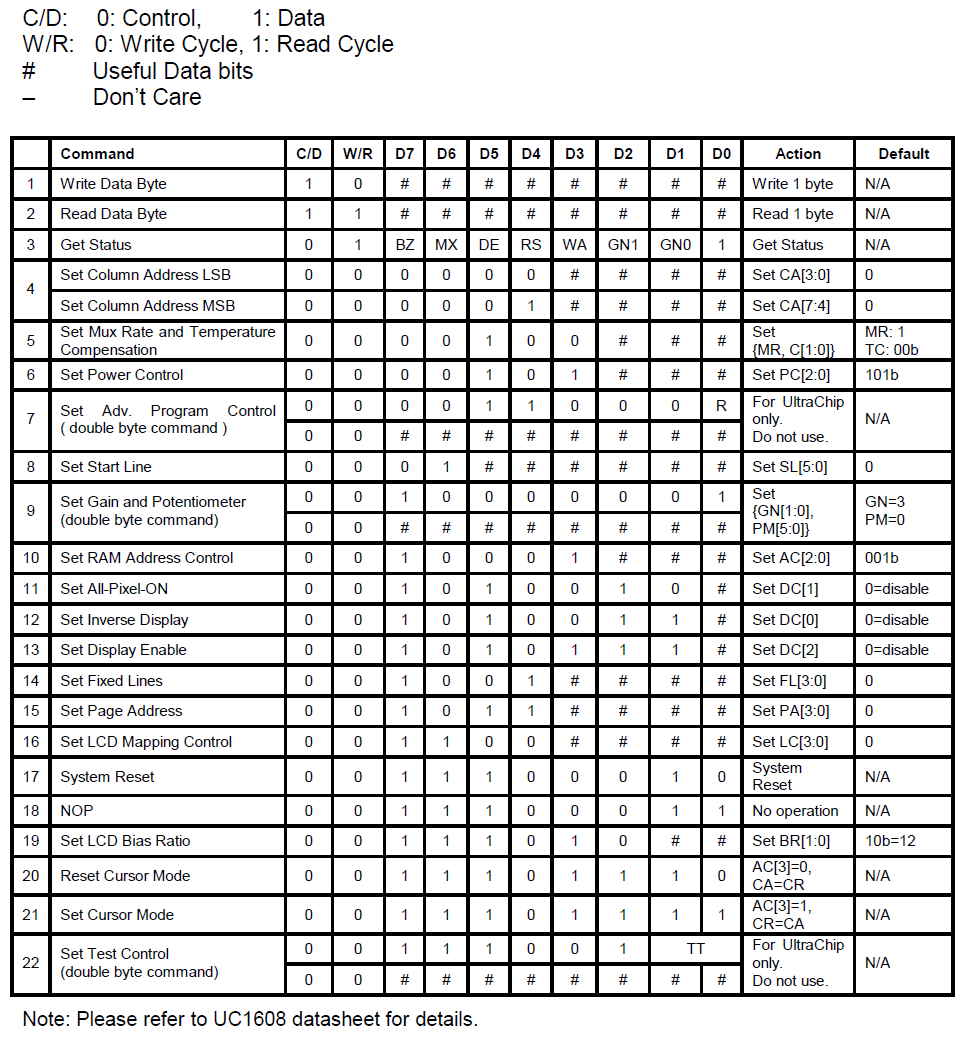
 일반적인 8비트 Latch IC이다. ATMEGA 128의 PORT A를 이용하는 외부 램 인터페이스가 하위 8비트의 데이터와 어드레스를 공유하기 때문에, 먼저 어드레스를 LATCH한 상태에서 데이터를 읽거나 써야 한다.

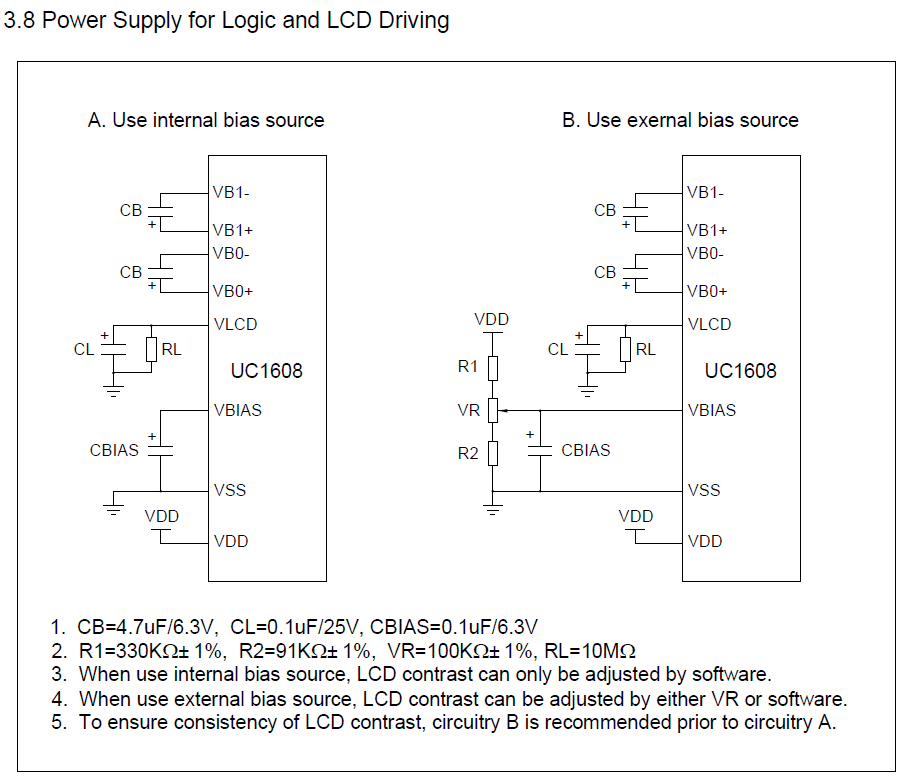
이 작업은 하드웨어 내부에서 자동으로 이루어지므로, 핀 할당과 내부 레지스터 설정만 올바르게 했다면 더 신경 쓸 필요는 없다.

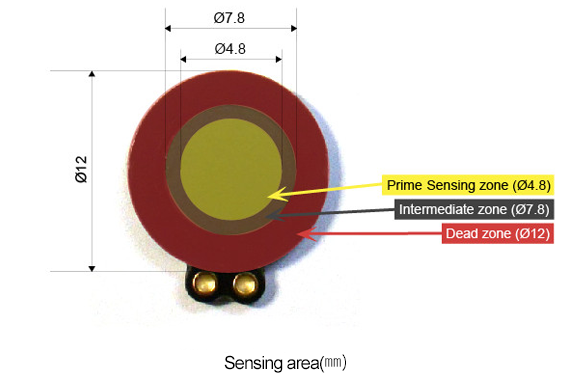
LG2401283 – GLCD

240 \* 128 해상도의 단색 그래픽 LCD이다. 특히, VDD 입력 전압이 2.7~ 3.3V임에 유의해야 한다. 다이오드 등을 이용해 감압 후 바이어스 할 것…



GLCD의 커맨드 리스트.

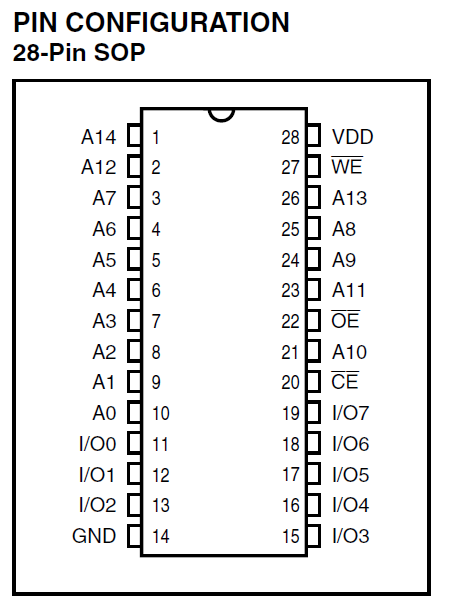


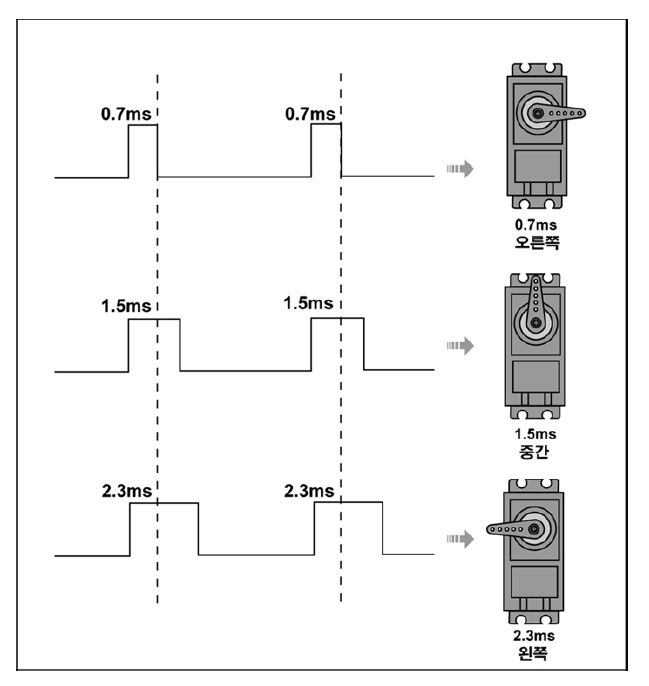
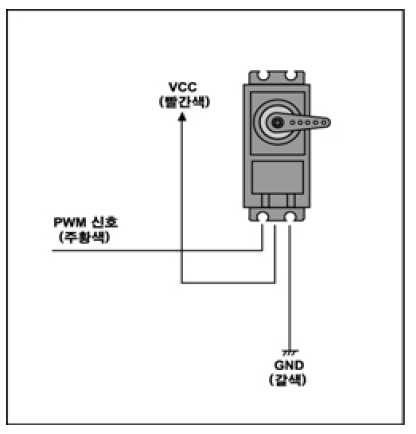
FSR RA12P 압력 센서

VIN 에 대한 특별한 언급이 없는 것으로 보아, 5V Vcc를 인가해도 무리가 없을 듯하다.



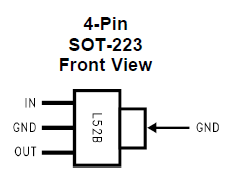
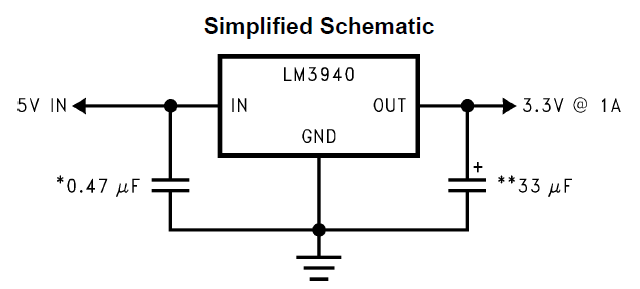
SRAM

전형적인 SRAM이다. 15비트 주소 입력과 8비트 데이터 입출력 포트를 갖는다. 5V 전원이 공급되어야 하며, 최대 응답 속도는 45ns (22.2MHz) 로 ATMEGA128에 충분히 사용 가능.

서보 모터 HES-288

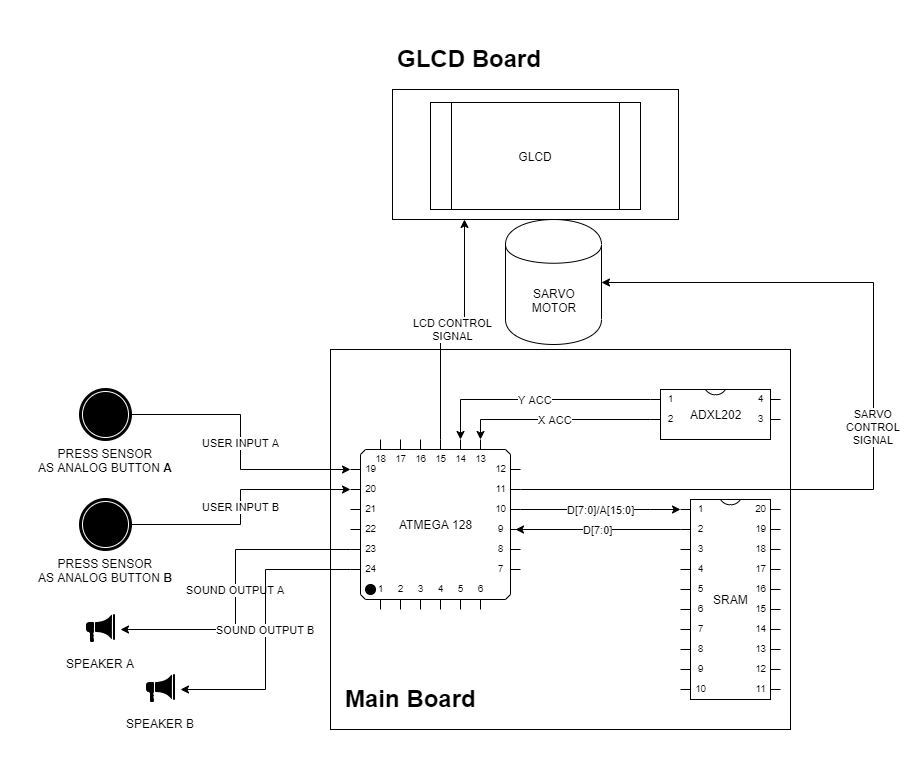
PWM을 이용해 제어되는 서보 모터이다. 게임이 1초에 60번씩 시뮬레이션 되므로, 16.67ms마다 모터에 대해 갱신 신호가 들어간다. OCR1B 비교 매치 인터럽트에서 OC1B 출력을 클리어하는 방법으로, PWM 출력을 소프트웨어 생성

정전압 레귤레이터 LM3940 SOT-223

 GLCD 디바이스의 바이어스 전압인 3.3V를 만들어내기 위해 사용되는 정전압 레귤레이터이다. 위와 같은 핀 배치를 가지며, 회로 상에는 다음과 같이 바이어스되어야 한다.

1A의 전류를 출력할 수 있는데, 이 디바이스에 연결되는 LCD의 V­DD 소모 전류량은 최대 1.5mA에 불과하므로 크게 문제되진 않는다.

**4. 작품 구성도 및 회로도**

작품 구성도

회로도 – 메인 보드

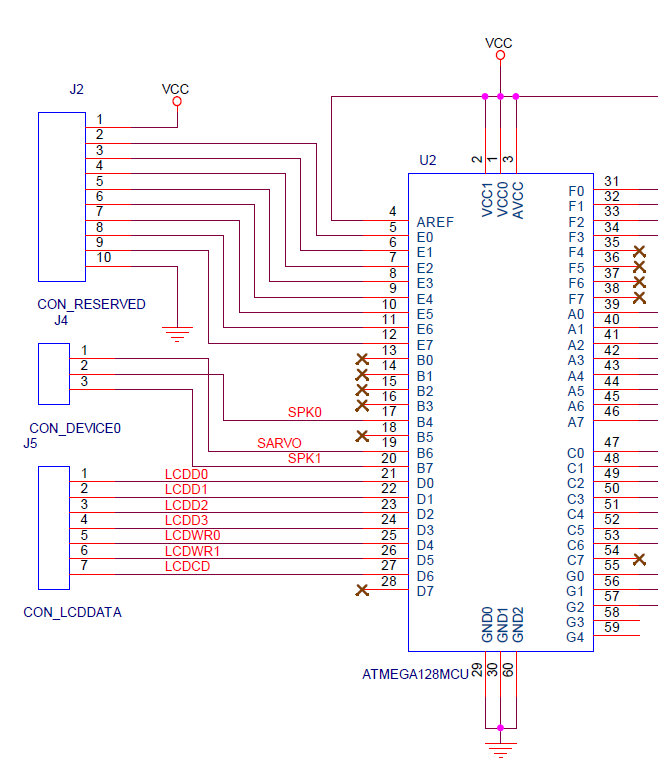


Figure . 외부 커넥터 연결

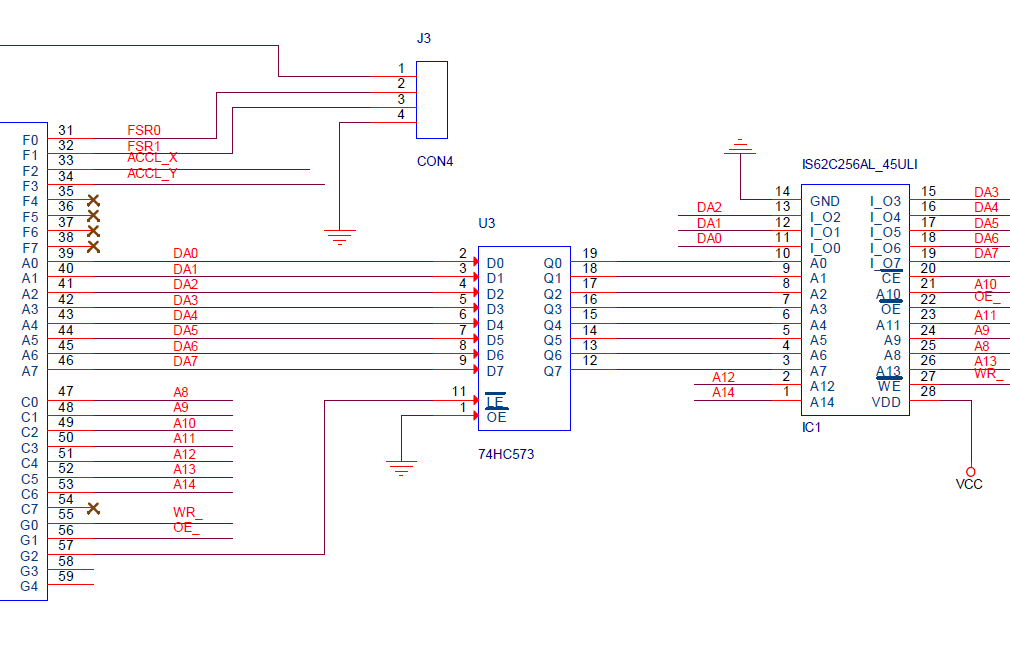


Figure . 메모리 연결

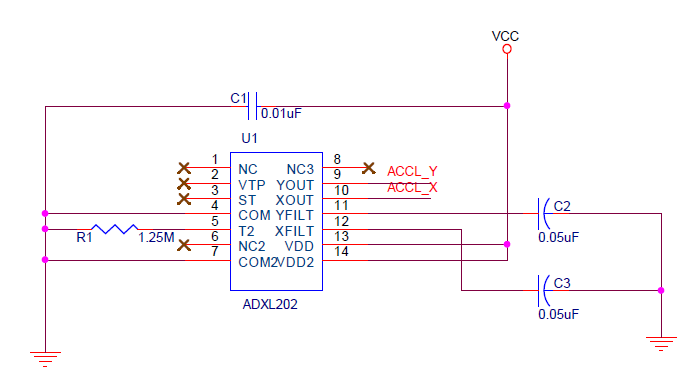


Figure . 가속도 센서

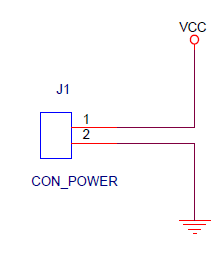
****

Figure . 전원 커넥터

GLCD 보드

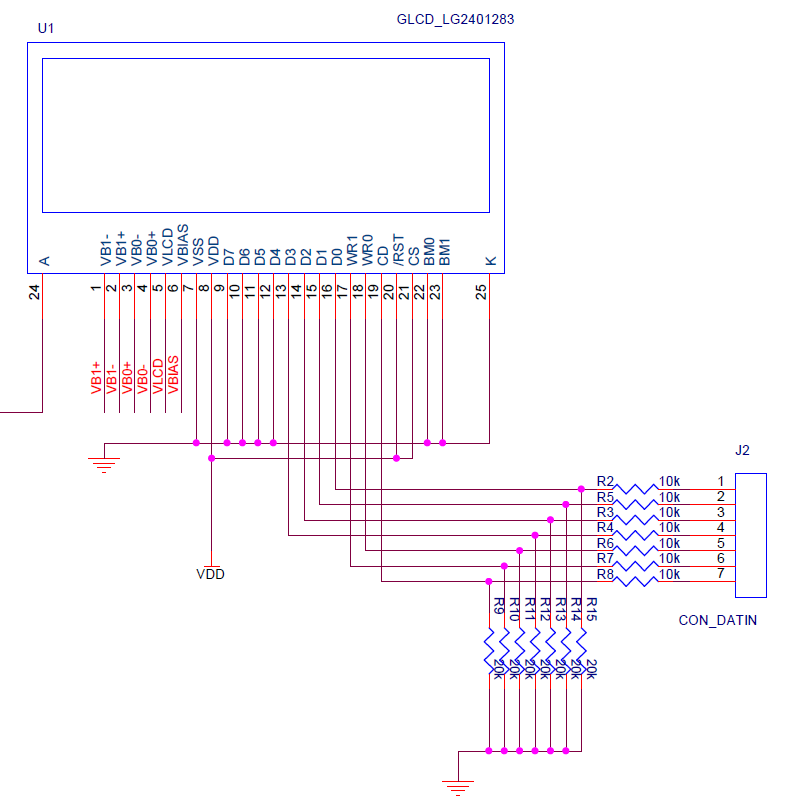


Figure . 커넥터 연결.

GLCD은 신호 입력 전압 또한 3.3V로 맞추어야 하므로, 5V의 출력 전압을 위와 같이 저항을 이용해 분배, 3.3V로 바이어스하였다.

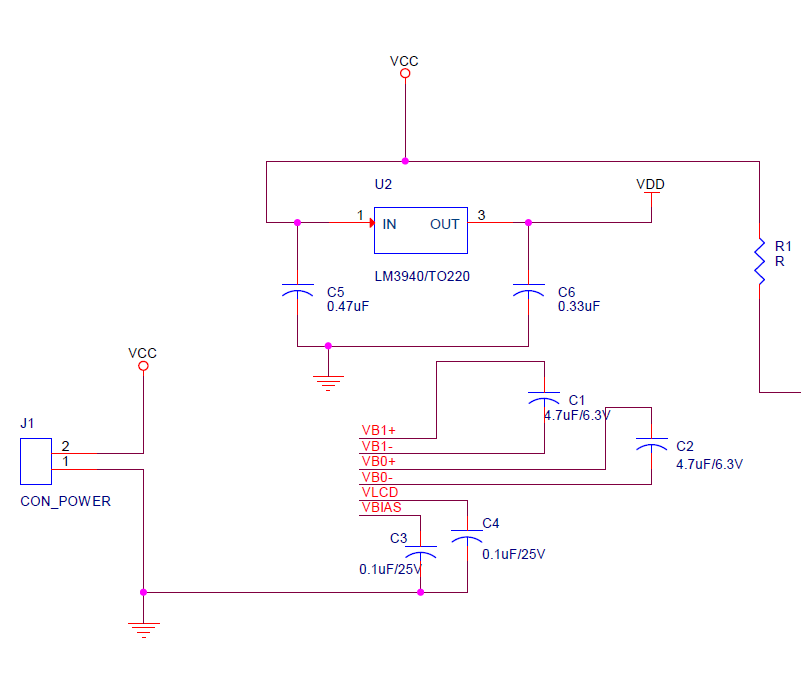


Figure . 바이어스

**5. 작품 진행상황**

첫 주에 PCB 기판의 설계를 잘못 해서 새로운 기판을 뽑아야 하는 관계로, 하드웨어 부분에서는 아무런 진전이 없었다.

소프트웨어 부분에서는 생산성을 높이기 위해 CodeVision IDE가 아닌 Visual Studio IDE에서, 오픈 소스 AVR 컴파일러인 AVR-GCC와 AVR 프로그래머 유틸리티인 AvrDude를 이용해 AVR-C 개발 환경을 구성하였다.

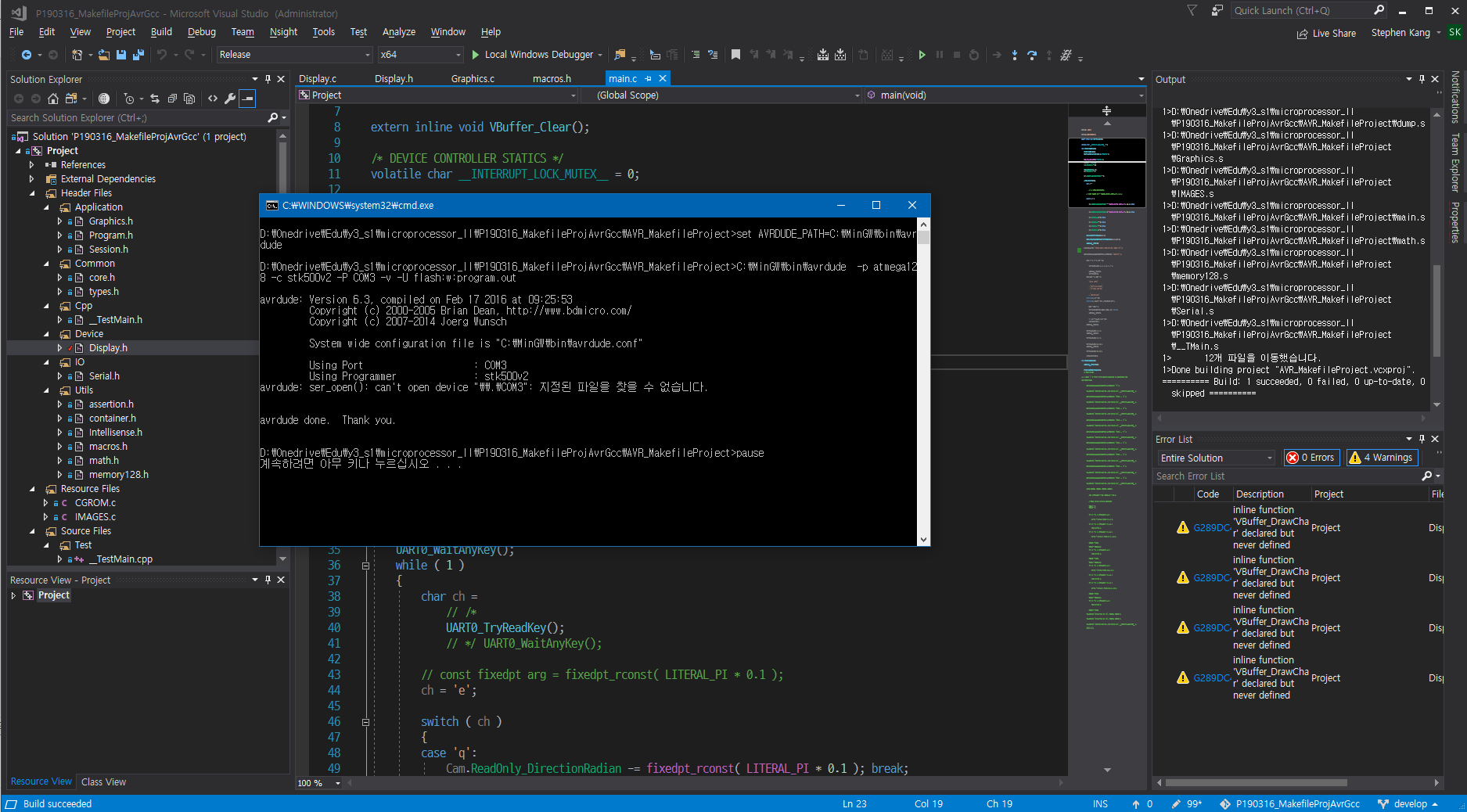


Figure . 개발 환경

지금까지 완성된 기능은 대부분 앞으로의 개발을 돕기 위한 기본적인 컨테이너와 유틸리티 기능들이다. 특히, 본격적으로 하드웨어를 완성하고 개발에 착수하기 전, 여러 가지 복잡한 함수성을 테스트하기 위해 시리얼 포트를 이용한 여러 가지 디버깅 함수에 공을 들였다. 아래는 *assertion.h* 파일에 선언된 매크로 및 함수들이다.

#ifndef \_DEBUG

#define assertf(...)

#define verifyf(expr, ...) if(!is\_true(expr)) { while(1); }

#define log\_verbose(...)

#define log\_display(...) { char \_\_buff\_\_[256]; sprintf(\_\_buff\_\_, \_\_VA\_ARGS\_\_); outputmsg\_uart0(\_\_buff\_\_); outputmsg\_uart0("\r\n"); }

#else // defined \_DEBUG

#if LOG\_VERBOSE

#define log\_verbose(...) log\_display(\_\_VA\_ARGS\_\_)

#else

#define log\_verbose(...)

#endif

#define log\_display(...) { char \_\_buff\_\_[256]; sprintf(\_\_buff\_\_, \_\_VA\_ARGS\_\_); internal\_logslow(\_\_FILE\_\_, \_\_LINE\_\_, \_\_buff\_\_ );}

#define verifyf(expr, ...) assertf(expr, \_\_VA\_ARGS\_\_)

#define assertf(expr, ...) if(!is\_true(expr)) { char \_\_buff\_\_[256]; sprintf(\_\_buff\_\_, \_\_VA\_ARGS\_\_); internal\_assertion\_failed(\_\_FILE\_\_, \_\_LINE\_\_, \_\_buff\_\_ );}

/\*\* Triggered when the assertion has failed.

\* To debug the arguments, dumps memory and output to rs232 serial port.

\*/

void internal\_assertion\_failed( const char\* file, int line, const char\* msg );

void internal\_logslow( const char\* file, int line, const char\* buff );

#endif

// Send message synchronously.

void outputmsg\_uart0( const char\* msg );

특히, 많은 부분이 조건부 매크로 블록 안에 들어가 있는데, 릴리즈 시점에 *macro.h* 파일의 여러 플래그를 조정해 이런 디버깅 기능 활성화 여부를 결정할 수 있게끔 하였다.

아래는 여러 가지 플래그와, 프로그램 전체에서 필수적으로 사용되는 유틸리티인 마스킹 매크로가 정의되어 있는 *macros.h* 파일이다.

#define \_MASK(num) (0x01 <<(num))

#define \_MASK\_SETBIT0(\_a) (\_MASK(\_a))

#define \_MASK\_SETBIT1(\_a, \_b) (\_MASK(\_a)|\_MASK(\_b))

#define \_MASK\_SETBIT2(\_a, \_b, \_c) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c))

#define \_MASK\_SETBIT3(\_a, \_b, \_c, \_d) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c)|\_MASK(\_d))

#define \_MASK\_SETBIT4(\_a, \_b, \_c, \_d, \_e) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c)|\_MASK(\_d)|\_MASK(\_e))

#define \_MASK\_SETBIT5(\_a, \_b, \_c, \_d, \_e, \_f) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c)|\_MASK(\_d)|\_MASK(\_e)|\_MASK(\_f))

#define \_MASK\_SETBIT6(\_a, \_b, \_c, \_d, \_e, \_f, \_g) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c)|\_MASK(\_d)|\_MASK(\_e)|\_MASK(\_f)|\_MASK(\_g))

#define \_MASK\_SETBIT7(\_a, \_b, \_c, \_d, \_e, \_f, \_g, \_h) (\_MASK(\_a)|\_MASK(\_b)|\_MASK(\_c)|\_MASK(\_d)|\_MASK(\_e)|\_MASK(\_f)|\_MASK(\_g)|\_MASK(\_h))

#define \_MASK\_GETMACRO(\_0, \_1, \_2, \_3, \_4, \_5, \_6, \_7, NAME, ...) NAME

#define mask(...) \_MASK\_GETMACRO(\_\_VA\_ARGS\_\_, \_MASK\_SETBIT7, \_MASK\_SETBIT6, \_MASK\_SETBIT5, \_MASK\_SETBIT4, \_MASK\_SETBIT3, \_MASK\_SETBIT2, \_MASK\_SETBIT1, \_MASK\_SETBIT0)(\_\_VA\_ARGS\_\_)

#define ARRAYCOUNT(Array) (sizeof(Array)/sizeof(\*Array))

extern volatile char \_\_INTERRUPT\_LOCK\_MUTEX\_\_;

#define INTERRUPT\_LOCK\_MUTEX (\*((volatile byte\*)&\_\_INTERRUPT\_LOCK\_MUTEX\_\_))

#define DISABLE\_INTERRUPT cli(); ++INTERRUPT\_LOCK\_MUTEX;

#define ENABLE\_INTERRUPT --INTERRUPT\_LOCK\_MUTEX; if(INTERRUPT\_LOCK\_MUTEX <= 0) { sei(); }

#define portc\_dbgout(val) PORTC = val

#define ENABLE\_DEBUG 1

#if ENABLE\_DEBUG

#define LOG\_VERBOSE 0

#define LOG\_MEMORY 0

#endif

#define ENABLE\_HEAP\_CACHE 1

#define USE\_STDLIB\_MALLOC 1

#define USE\_SERIAL\_COMMUNICATION 1

#if ENABLE\_DEBUG

#ifndef \_DEBUG

#define \_DEBUG

#endif

#endif

특히, DISABLE\_INTERRUPT와 ENABLE\_INTERRUPT 매크로는 원자성이 보장되어야 하는 함수들 내부에서 일시적으로 인터럽트를 비활성화 시키기 위해 사용되었다. 내부적으로는 Semaphore를 사용해, Diable된 횟수가 0인 경우에만 인터럽트가 다시 활성화된다.

아래는 시리얼 포트로 메시지 등을 출력하기 위한 함수이다.

void outputmsg\_uart0( const char\* msg )

{

DISABLE\_INTERRUPT;

const char\* head = msg;

while ( \*head != '\0' )

{

while ( !( UCSR0A & 0x20 ) );

UDR0 = \*( head++ );

}

while ( !( UCSR0A & 0x20 ) );

ENABLE\_INTERRUPT;

}

다음은 여러 데이터를 보관하는 데 사용되는 컨테이너이다. 특히 노드로 연결되는 Linked List는 프로그램 전반에 걸쳐 대단히 핵심적인 역할을 수행하므로, 개발 전반에 걸쳐 최적화를 다듬을 예정이다. 아래에는 지면상 헤더 파일만을 포함하였다.

#pragma once

#include "Intellisense.h"

#include "types.h"

#include "assertion.h"

#include <string.h>

#include "memory128.h"

/\*\* Dynamic array. All struct elements are read only. \*/

// N must be 8 \* byte size

typedef struct TArray {

uint16 \_count;

uint8 \_ofst;

uint8\* \_data;

} TArray;

typedef TArray FString;

typedef struct TListNode {

struct TListNode\* Prev;

struct TListNode\* Next;

void\* Element;

} TListNode;

typedef struct TList {

uint8 \_ofst;

// CAUTION: READ ONLY!

TListNode\* Head;

TListNode\* Tail;

} TList;

inline void TArray\_Initialize( TArray\* const pArray, const uint8 ElementSize, const size\_type Capacity );

inline void TArray\_Dispose( TArray\* const pArray );

/\* returns element pointer. \*/

inline void\* TArray\_At( TArray\* const pArray, size\_type Index );

/\* returns new element index. \*/

size\_type TArray\_AddLast( TArray\* const pArray, void const\* const Element );

void TArray\_RemoveElement( TArray\* const pArray, size\_type Index );

void FString\_Initialize( FString\* const pString, const char\* InitData );

inline void TList\_Initialize( TList\* const pList, const uint8 ElementSize );

void TList\_Dispose( TList\* const pList );

void TList\_PushFront( TList\* const pList, const void\* const Element );

void TList\_PushBack( TList\* const pList, const void\* const Element );

void TList\_Insert( TList\* const pList, TListNode\* const pPrecededNode, const void\* const Value );

void TList\_PopFront( TList\* const pList );

void TList\_PopBack( TList\* const pList );

void TListNode\_Remove( TListNode\* const pNode );

inline void\* TArray\_At( TArray\* const pArray, size\_type Index )

{

uint8\* pCursor = pArray->\_data + Index \* pArray->\_ofst;

assertf( (void\*) pCursor < GetMemoryBound( pArray->\_data ), "Invalid memory access!" );

return pCursor;

}

inline void TArray\_Initialize( TArray\* const pArray, const uint8 ElementSize, const size\_type Capacity )

{

pArray->\_ofst = ElementSize;

pArray->\_data = (uint8\*) Malloc( Capacity \* pArray->\_ofst );

pArray->\_count = 0;

}

inline void TList\_Initialize( TList\* const pList, const uint8 ElementSize )

{

pList->\_ofst = ElementSize;

pList->Head = pList->Tail = NULL;

}

inline void TArray\_Dispose( TArray\* const pArray )

{

Free( pArray->\_data );

}

다음은 Rasterization에 사용되는 디스플레이 함수성입니다. 내부