2019

LIDAR Depth Scanner

DOCUMENTATION

씨컨트롤

개요

두 개의 스태핑 모터를 이용하여 X, Y 축으로 센서를 회전시키고, 이를 이용해 주변 환경을 스캔, 이더넷 프로토콜을 통해 측정된 데이터를 연결된 TCP 소켓으로 뿌려주는 장치입니다.

STM32F401VCT6 칩을 기준으로 펌웨어가 작성되었으며, 핀 할당은 다음과 같이 되어 있습니다.

포트	핀	할당	기능
GPIOA	5	[COMMON] SPI1 SCLK W5500 SCLK IN AFBR-S50 SCLK IN	SPI SCLK
	6	[COMMON] SPI1 MISO W5500 MISO OUT AFBR-S50 MISO OUT	SPI MISO
	7	[COMMON] SPI1 MOSI W5500 MOSI OUT AFBR-S50 MOSI OUT	SPI MOSI
GPIOC	0	AFBR-S50 CHIP SELECT	
	1	AFBR-S50 IRQ	
	2	W5500 CHIP SELECT	
	3	NO-USE (W5500 IRQ)	
GPIOC	6	Magnetic sensor horizontal	
	7	Magnetic sensor vertical	
GPIOA	8	Horizontal motor driver [PULSE] input	
	9	Horizontal motor driver [DIR] input	
	10	Horizontal motor driver [ENABLE] input	
GPIOB	8	Vertical motor driver [PULSE] input	
	9	Vertical motor driver [DIR] input	
	10	Vertical motor driver [ENABLE] input	
VCC		Pow LED	
GPIOD	12	LED – status – error	Red LED
	13	LED – status - wait	Blue LED
	14	LED – status - operating	Green LED
GPIOA	13	SWDIO	
	14	SWCLK	

펌웨어 기본 주의사항

처음 실행했을 때, 펌웨어는 모터의 원점을 찾기 위해 모터를 지속적으로 회전시킵니다. 이는 근접 센서 입력이 Low(Active) 되는 즉시 종료되며, 만약 센서를 사용하지 않는 경우 GPIOC의 6, 7번 핀을 반드시 접지시켜야 합니다. 그렇지 않으면 프로그램의 초기화가 진행되지 않습니다. (곧 기본적으로 풀다운 되게끔 수정할 예정입니다).

프로그램 흐름

프로그램의 전체적인 구성은 src/platform/cppmain.cpp 파일에 기술되어 있습니다. 이를 메인 파일이라 부르겠습니다. 대부분 C++ 코드 요소를 활용하므로 C++에 대한 어느 정도의 이해가 필요합니다.

먼저 메인 파일에서 include하는 모든 소스 파일들의 목록과, 그 용도입니다. 따로 주석을 작성하지 않은 파일은 무시하셔도 되는 내용입니다.

```
#include <app/intellisense.h>
#include <stm32f4xx_conf.h>
#include <app/macros.h>
#include <example/discovery_specific.h>
#include <stdio.h>
#include <app/utility.h>
#include <stm32f4xx_gpio.h>
#include <stm32f4xx_exti.h>
#include <platform/s2pi.hpp> - SPI 인터페이스입니다.
#include <platform/hw_timer.hpp> - 하드웨어 타이머 로직입니다.
#include <platform/irqlock.hpp>

extern "C"
{
#include <argus.h> - 센서 구동에 사용되는 API 입니다.
#include <platform/argus_s2pi.h>
#include <platform/argus_timer.h>
#include <ioLibrary/Ethernet/socket.h> - 소켓 통신에 사용되는 인터페이스
#include <ioLibrary/Ethernet/wizchip_conf.h>
#include <ioLibrary/Internet/DHCP/dhcp.h>
```

```
    void delay_us(int us);
    void delay_ms(int us);
}
#include <device/hw_motor.hpp> - 모터 구동 인터페이스입니다.
#include <platform/hw_netrw.hpp> - PC 와의 통신 프로토콜
#include <cstdarg>
#include <device/hw_proximity.hpp> - 근접 센서 구동
```

프로그램이 시작되는 지점입니다.

```
extern "C" void CppMain()
{
    // -- Initialization code
    // Configure SysTick
    // 84MHz / 84000 ---> 1kHz ... 1ms Interval!
    SysTick_Config(g_SystickScalerVal);
 // 먼저, 내부 타이머 구동을 위해 SysTick 을 활성화시킵니다. 내부 타이머의 해상도는 1ms 입니다.

    // Initialize SPI peripherals
    InitializeS2piPeripheral();
 // SPI 장치를 시동합니다. 위의 표에 할당된 것처럼, 칩 선택 출력과 IRQ 입력 핀을 초기화하고, 빠른 SPI 구동을 위해 DMA 장치를 초기화합니다.
 // 이 함수의 구현은 platform/argus_s2pi.cpp 에 작성되어 있습니다.

    // @todo. Initialize motor
    InitMotors();
 // 모터 장치를 초기화합니다. PWM 신호를 생성하는 타이머 장치를 초기화합니다.

    // @todo. Initialize sensor environment
    Proximity_Initialize(MAGNETIC_INDEX_X);
    Proximity_Initialize(MAGNETIC_INDEX_Y);
 // 근접 센서 입력을 받기 위해 GPIOC 핀 6, 7 번을 초기화합니다. 함수의 파라메터로 넘어간 MAGNETIC_INDEX 값은 단순한 정수로, 실제 각각의 GPIO 핀 초기화에 필요한 인자들은 배열 형태로 지정되어 전달된 인덱스에 따라 알맞게 초기화됩니다.

    // Initialize command handler
    auto v = skDeviceCommandHandler();
 // Event handler 클래스로서, 호스트로부터 전달되는 이벤트 알림을 처리하는데 사용됩니다.

    // -- Program initializaer
    APP_INIT();
 // 어플리케이션을 초기화합니다.
```

```
    // -- Program loop
    // includes initialization of net configuration
    g_NetRW.EVENT_PROCEDURE(&v);
 // 프로그램 루프입니다. 내부에 루프가 정의되어 있기 때문에, 디바이스가 종료될 때까지 이 함수를 절대 빠져나오지 않습니다.

    // On program aborted.
    while (1)
        ;
}
```

프로그램은 다수의 타이머와 비동기 이벤트로 구성되어 있기 때문에 인터럽트를 적극적으로 활용하게 되는데, 이 때 인터럽트 내에서 제어를 오래 잡고 있는 경우 다양한 문제의 원인이 되기에 가급적 인터럽트 함수의 점유 시간을 최소화할 필요가 있습니다. 따라서, 이 프로그램은 인터럽트 함수 내에서 작업을 처리하는 대신, Event driven design을 도입, 인터럽트 내에서는 이벤트를 Issue시키기만 하고(Enqueue) 실제 처리는 메인 프로시저에서 처리하게 됩니다.

```
static struct skProgramCmdQueue
```

이것을 담당하는 것이 이 클래스로서, 전역에 단일 인스턴스로 존재하며 프로그램 전반에서 이슈되는 모든 이벤트를 처리하게 됩니다.

```
void QueueCommand(uint32_t Microsec, cmd_t Cmd)
```

이 함수의 호출을 통해 이벤트를 이슈하면, 프로그램 루프가 다시 돌아왔을 때 이슈시킨 순서대로 처리합니다.

```
DECLCMD(App_Tof_Init);
DECLCMD(App_Initialize);
DECLCMD(App_Measure);
DECLCMD(App_OnMeasureDone);
DECLCMD(App_Start);
DECLCMD(App_Continue);
DECLCMD(Device_WaitMotors);
DECLCMD(Device_MotorOn);
DECLCMD(Device_MotorOff);
```

이벤트는 위와 같이 매크로로 선언된 상수들을 위의 QueueCmd 함수로 호출하는 것으로 이루어지며, 필요하다면 새로운 이벤트와 이벤트 핸들러를 DECLCMD 함수를 통해 간단하게 정의할 수 있습니다.

펌웨어의 주요 기능은 모터 제어에 기반한 센서 제어인데, 이것에 대한 전반적인 상태는 아래의 구조체에 정의되어 있습니다.

```
static struct
{
    enum
    {
        BuffSize = DEFAULT_LINE_BUFF_SIZE
    };
    struct Point
    {
        int32_t X = 0;
        int32_t Y = 0;
    } StepsPerCapture, Resolution, StepIdx, MotorOffset;
    int8_t Dir = 0;
    bool bRunning = false;
    q9_22_t *lpBuffHead = nullptr;
    q9_22_t lpLineBuff[DEFAULT_LINE_BUFF_SIZE];
    size_t LinePixelOfst = 0;
    uint64_t CaptureBeginTimeUsec = 0;
    bool bMotorEnabled = false;

    inline bool IsStopped() const
    {
        return bRunning == false && Dir == 0;
    }
} g_App;
```

이 정보는 익명의 구조체로 선언되어 있으며, 내부적으로 두 개의 정수로 좌표를 나타내는 Point 타입을 통해 한 번의 캡쳐에 몇 번의 스태핑(StepsPerCapture, 모터 클럭)이 필요한지, 종횡 총 몇 픽셀을 캡처하는지(Resolution), 현재 몇 번째 픽셀을 찍고 있는지(StepIdx), 모터가 초기화되는 위치가 어디인지 정보를 저장합니다(MotorOffset).

캡처는 X축을 지그재그로 이동하며 진행되는데, 이 때 X축의 방향이 어디인지를 나타내는 변수(Dir), 현재 캡쳐가 진행중인지, 혹은 Idling중인지를 나타내는 플래그(bRunning), 캡쳐된 데이터를 PC로 보내기 전, 효율을 위해 저장해두는 버퍼(lpBuffHead, lpLineBuff), 한 개의 라인이 너무 길 경우 몇 번에 걸쳐 보내질 수 있기 때문에, 현재 보내는 라인이 몇 번째 픽셀인지를 나타내는 변수(LinePixelOfst), 캡처에 소요된 시간을 나타내기 위해 캡처가 시작된 시간을 담는 변수(CaptureBeginTimeUsec), 모터 활성화 여부(모터 En 신호)를 나타내는 플래그(bMotorEnabled) 등의 정보가 담기며, 이들 전반이 프로그램을 제어하게 됩니다.

예를 들어, 아래는 프로그램의 초기화 시점에서 호출되는 APP_INIT() 함수입니다.

```
void APP_INIT()
{
    // Initialize parameters
    // These are default parameters

    g_App.StepsPerCapture = {int32_t(1.f / ANGLE_PER_STEP_X),
int32_t(1.f / ANGLE_PER_STEP_Y)};
    g_App.Resolution.X = 324;
    g_App.Resolution.Y = 16;

    g_StepperX.SetSpeed(DEFAULT_MOTOR_CLK_X);
    g_StepperY.SetSpeed(DEFAULT_MOTOR_CLK_Y);

    QueueCommand(0, eProgramCmd::App_Initialize);

    // Test code
    QueueCommand(0, eProgramCmd::Device_MotorOn);
    QueueCommand(0, eProgramCmd::App_Start);
}
```

아래와 같이, 초기화를 마친 이후 세 가지 작업을 Queue에 넣습니다.

프로시저

```
    // -- Program loop
    // includes initialization of net configuration
    g_NetRW.EVENT_PROCEDURE(&v);
```

위에서 잠깐 언급한 것과 같이, 프로그램 루프는 네트워크 통신 클래스의 내부에 정의되어 있습니다. 이 클래스에 관한 내용은 platform/hw_netrw.hpp, device/hw_netrw.cpp 에 자세하게 기술되어 있습니다. 여기에서는 간략히 프로그램 루프의 진행에 대해서만 짚고 넘어가도록 하겠습니다.

기본적으로 `class iNetDeviceRW : public iNetRW` 클래스는 바이트 스트림을 기반으로 동작하게끔 설계되어 있습니다. 즉, 어떤 종류의 시리얼 통신 메커니즘이라도

```
class iNetRW
{
public:
    enum class StatusType
```

```
    {
        Wait,
        Read,
        Idle,
        Abort
    };
public:
    virtual StatusType GetStatus() const { return StatusType::Abort; }
protected:
    virtual void Initialize() = 0;
    virtual void Write( char const* DataPtr, size_t NumBytes ) = 0;
    virtual void Read( char* OutBuffer, size_t ReadCnt ) = 0;
    virtual void Loop() = 0;

    // Exception handlers
    virtual void EXCEPTION_CommuHeaderInvalid() {};
    virtual void EXCEPTION_UnknownOpCode( int ) {};
};
```

이 인터페이스의 Write와 Read 함수를 재정의함으로써 통신 기능을 만들 수 있는 것입니다. 만약 USB 통신이 필요하다면, 모든 함수가 궁극적으로 Write와 Read 함수를 사용하기 때문에, 이 두 개의 함수를 조정하는 것만으로 손쉽게 통신 메커니즘을 대체할 수 있습니다.

iNetDeviceRW의 i라는 prefix는 interface의 i로서, 동작을 위한 뼈대만을 제공하고 실제 구현은 이 인터페이스를 구현하는 동작 클래스에 맡긴다는 의미입니다. 아래 소개하는 EVENT_PROCEDURE() 함수에서 호출하는 Initialize() Loop() GetStatus() 등의 함수들은 모두 인터페이스 함수로서, 선언만 되어 있을 뿐 실제 동작은 이 인터페이스를 상속/구현하는 동작 클래스의 정의에 의존합니다.

프로시저 함수는 내부에 루프를 정의하고 있어, 탈출 조건(연결 취소)이 성립하지 않는 한 루프를 빠져나오지 않습니다.

```
// -- Callback loop
    while ( GetStatus() != StatusType::Abort )
    {
```

실제 구현된 프로그램에서 GetStatus() 함수는 결코 StatusType::Abort를 반환하지 않으므로, 실질적으로 프로그램은 가동 시간 내내 이 프로시저 함수 내에서 루프를 돌게 됩니다.

아래는 루프 전체를 주석과 함께 작성하였습니다. 흰 글씨로 쓰인 한글이 본문입니다.

```
// -- Callback loop
    while ( GetStatus() != StatusType::Abort )
    {
```

```
        Loop();
        // 위에서도 언급한 바 있는, Loop 함수입니다. 이 함수는 DHCP 주소의
업데이트를 공유기에 요청하고, 지속적으로 상태를 모니터링합니다. 아래에서
자세하게 서술하겠습니다.

        if ( GetStatus() != StatusType::Read ) {
            continue;
        }
        // 만약 읽어들일 데이터가 있는 경우에만, GetStatus()가 반환하는
상태가 StatusType::Read 가 됩니다. 아니라면 프로그램은 계속해서 루프를 돌게
됩니다.

        // Set head
        char* phead = Buff;
        // 여기서부턴 데이터를 읽어들이고, 파싱하는 코드입니다.

        // -- Parse header
        constexpr auto HeaderSize = sizeof( sNetHeader );
        Read( phead, HeaderSize );
        // 2 바이트의 헤더 사이즈만큼 메모리를 읽어들입니다.

        const auto& Header = *ptr_cast<sNetHeader>( phead )++;

        // Verify ID
        if ( Header.ID != sNetHeader().ID )
        {
            EXCEPTION_CommuHeaderInvalid();
            continue;
        }
        // 헤더는 반드시 정해진 값을 만족시켜야 합니다. 아닌 경우 잘못된 읽기
요청으로 간주하고, 읽기 동작이 취소되며 다시 루프로 돌아갑니다.

        // Processing on begin read ... ex) irq lock
        OnBeginParse();
        // 이는 가상 함수로서, 인터페이스 함수와 유사하지만 기본 동작을
제공하는 점이 다릅니다.

        // Device should parse opcode as command !
        switch ( Header.Opcode.Command )
        {
        // 읽어들인 헤더가, 어떤 명령을 지정하는지 확인하고 분기합니다.
```

```
        case eCommand::RequestDeviceProfile:
        // 각각의 명령을 나타내는 상수에 대해, 전달된 요청에 알맞은 동작을
수행합니다. 가장 위에서 언급한 적 있는, Event Handler 클래스가 이 이벤트들의
처리를 맡습니다.
            {
                // Make up net header
                sNetHeader r;
                r.Opcode.Response = eResponse::DeviceProfile;

                // Make up device profile
                device::sDeviceProfile arg;
                Callback->GetDeviceProfile( &arg );

                // Transmit
                Write( tobuf( &r ), sizeof r );
                Write( tobuf( &arg ), sizeof arg );
                break;
            }
        case eCommand::SetAngularResolution_DEPRECATED:
            {
                // Read further information from stream
                Read( phead, sizeof( float ) );
                auto Resolution = *ptr_cast<float>( phead )++;
                Callback->SetAngularResolution_DEPRECATED( Resolution );
                break;
            }
        case eCommand::SetScanningRange:
            {
                // Read further information from stream
                Read( phead, sizeof( device::sScanningRangeDesc ) );
                auto const& Range =
*ptr_cast<device::sScanningRangeDesc>( phead )++;
                Callback->SetScanRange( Range );
                break;
            }
        case eCommand::RequestReset:
            Callback->ResetScanner();
            break;

        case eCommand::RequestStart:
            Callback->StartScanner();
            break;

        case eCommand::RequestPause:
            Callback->PauseScanner();
```

```
            break;

        case eCommand::Ping:
            {
                sNetHeader r;
                r.Opcode.Response = eResponse::Ping;
                Write( tobuf( &r ), sizeof r );
            }
            break;

        case eCommand::RequestReport:
            {
                // Makeup header
                sNetHeader Header;
                Header.Opcode.Response = eResponse::Status;

                // Makeup data
                device::sStatusReportDesc Report;
                Callback->ReportStatus( &Report );

                // Transmit
                Write( tobuf( &Header ), sizeof Header );
                Write( tobuf( &Report ), sizeof Report );
                break;
            }
        default:
            EXCEPTION_UnknownOpCode( ( int) Header.Opcode.Command );
            break;
        }

        // data receive should be less than allocated size.
        assert( phead < Buff + BufferSz );

        // Processing on end read ... ex) irq unlock
        OnEndParse();
    }

    free( Buff );
}
```

Loop() 함수를 보겠습니다.

```
switch ( DHCP_run() )
        {
        case DHCP_IP_ASSIGN:
        case DHCP_IP_CHANGED:
        case DHCP_IP_LEASED:
        case DHCP_FAILED:
        default:
        }
```

코드를 일부 생략하였습니다.

위와 같은 꼴로 구성이 되어 있는데, 이는 즉 DHCP_run() 함수의 실행 결과에 따라 프로그램이 분기하는 모양새입니다. DHCP_run()은 매 틱마다 호출되는 일종의 업데이트 함수이며, 프로그램 기동 직후 공유기로부터 IP를 할당받게 되면 케이블을 뽑지 않는 한 DHCP_IP_LEASED 상태에 있게 됩니다.

프로그램은 DHCP 서버로부터 ip를 할당 받은 후부터 제대로 동작할 수 있으므로, DHCP_IP_LEASED에 핵심적인 동작이 정의되어 있습니다.

```
if ( m_status == eNetStatus_t::Wait )
 // m_status 는 위에서도 언급한 GetStatus() 함수에서 사용되는 변수로서,
GetStatus() 함수는 단순히 m_status 변수를 반환하는 식으로 구현되어 있습니다.
 // 여기서, eNetStatus_t 는 위의 StatusType 과 같은 열거형 변수입니다.
Alias 만 부여하였습니다.
 // 이 조건문은 현재 상태가 Wait 상태, 즉 연결이 되지 않은 상태일때만
진입하게 되며, 연결이 된 Idle 상태인 경우 매 루프에서 무시됩니다.
{
    // g_PortIdx = 5333 + ( rand() ^ SysTick->VAL ) % ( 0xffff -
5333 );
    g_PortIdx = scan::net::CONNECTION_PORT;
    // 통신할 포트 번호입니다. 33419 번으로 고정되어 있습니다.

    // Open UDP broadcast socket
    ::socket( EHS_UDP_SOCK_IDX, Sn_MR_UDP, g_PortIdx, 0 );
    // UDP 통신 소켓입니다. 프로그램이 연결되기 전, 지속적으로 로컬 네트워크
내에 자신의 주소를 알리는 데에 사용합니다.

    // Open the tcp server and wait
    ::socket( EHS_TCP_SOCK_IDX, Sn_MR_TCP, g_PortIdx, 0 );
    auto rslt = SOCK_ERROR;
    while ( rslt != SOCK_OK )
    {
        rslt = ::listen( EHS_TCP_SOCK_IDX );
        printf( "Listening result= %d\n", rslt );
```

```
    }

    printf( "Opening socket on port %d\n", int( g_PortIdx ) );

    // Broadcast IP info to local
    uint8_t brd_dat[8];
    ( ( uint32_t*) brd_dat )[0] = scan::net::CONNECTION_REQUEST_HEADER;
    ( ( uint32_t*) brd_dat )[1] = *( uint32_t*) gWIZNETINFO.ip;
    // 로컬 네트워크 상에 뿌려줄 8 바이트의 주소 데이터입니다. 첫 4 바이트는
헤더이며 (0x12349955), 다음 4 바이트는 주소입니다.

    while ( getSn_SR( EHS_TCP_SOCK_IDX ) == SOCK_LISTEN )
    {
        // Try broadcast my IP. It will be preceded by header to
identify protocol.
        uint32_t addr_broad = 0xff000000u | *(uint32_t*)gWIZNETINFO.ip;

        ::sendto( EHS_UDP_SOCK_IDX, brd_dat, sizeof brd_dat, ( uint8_t*)
& addr_broad, g_PortIdx );

        // Delay for 1000 ms.
        // 네트워크 상에 지나치게 많은 패킷이 쌓이는 것을 막기 위해, 1 초에 한
번만 송신됩니다.
        uint64_t GetLifetimeMicroSec() ;
        volatile auto begin = GetLifetimeMicroSec();
        while(true)
        {
            volatile auto b = GetLifetimeMicroSec();
            if(b - begin > 1000000)
            {
                break;
            }
        }
    }

    ::close( EHS_UDP_SOCK_IDX );
    // 역할을 다한 UDP 소켓은 닫아줍니다.

    printf( "Connection successful on port %d\n", int( g_PortIdx ) );
    void ON_CONNECT();
    ON_CONNECT();
    // 메인 루프에 정의된, ON_CONNECT 함수를 호출해줍니다.
}
```

```
// -- User code
// Try recv
auto rslt = ::CheckIncomingBytes( EHS_TCP_SOCK_IDX );
// CheckIncomingBytes는 말 그대로, 읽을 내용이 있는지를 검출하는
함수입니다. 읽을 값이 있으면 1 이상의 값, 없다면 0, 연결이 끊어진 경우 -1을
반환합니다. 그에 따른 동작은 아래에 정의되어 있습니다.

// If there's any data to recv ...
if ( rslt > 0 )
{
    m_status = eNetStatus_t::Read;
    // 이와 같이, 단순히 상태를 바꾸는 것으로 위에서 언급한 읽기 동작이
수행되거나 합니다.
}
// Set state as busy to keep repeat loop unless there's any input data.
else if ( rslt == 0 )
{
    m_status = eNetStatus_t::Idle;
    void PROCEDURE(); // external program loop
    PROCEDURE();
    // Idle 상태를 나타냅니다. 읽을 것이 없으므로, 메인 루프의 동작을
수행합니다.
}
// If socket is in invalid state ...
else // rslt < 0
{
    m_status = eNetStatus_t::Wait;
    // 연결이 끊어진 경우, 자동으로 다음 루프 진입 시점에서 위의 조건문 안에
코드가 묶이게 되어 안전한 동작을 보장합니다.
}
```

정리하자면,

Cppmain() -> EVENT_PROCEDURE() -> Loop() -> PROCEDURE() 의 형태로 루프가 진행됩니다.

마지막으로, 프로그램의 실 동작을 정의하는 PROCEDURE()를 보겠습니다.

```
void PROCEDURE()
{
    // -- Constant Program Loop

    // -- Output queued log messages
    INTERNAL_FlushLogMessage();

    // -- Set fence before execute commands
    g_CmdQueue.Internal_AdvanceFence();

    // -- Process all accumulated commands
    while (true)
    {
        int Cmd = (int)g_CmdQueue.Internal_TryPopEvent();

        assert(Cmd < (int)eProgramCmd::MAX);

        if (Cmd == 0)
        {
            break;
        }

        skExecCmd::clbk[Cmd]();
    }
}
```

위에서 언급한, 이벤트라는 것은 각각이 하나의 1바이트 상수입니다. 이벤트 하나를 issue시킬 때마다 큐 클래스 내부의 배열에 상수가 큐 되며, 바로 위에서 말한 절차에 따라 프로시저가 호출될 때마다 큐 된 이벤트들이 하나씩 호출이 됩니다.

상수 하나하나가 각각 콜백 배열의 인덱스로 사용되며, 하나씩 큐 된 상수를 Pop해서 그 인덱스에 해당하는 함수를 clbk 배열에서 찾아 호출하는 것으로 이벤트가 실행됩니다. 프로시저 진입점에서 Fence를 설정하는 것을 볼 수 있는데, 프로그램이 Procedure 내부에만 묶여 있으면 각종 통신 장치 등의 처리가 보류되므로, 한 번 프로시저에 진입할 때마다 정해진 숫자의 이벤트만 처리하게끔 강제하는 장치입니다.

CppMain.cpp 파일의 아래 부분에는 위에서 언급한 이벤트들이 하나씩 정의되어 있는데, 이들은 함수 이름 그대로의 역할을 수행합니다.

파일

아래는 모든 파일의 간단한 설명입니다. 중요한 파일은 강조 처리 해두었습니다.

```
|    startup_stm32f40xx.s
|    stm32f4xx_conf.cpp
|    stm32f4xx_conf.h
|    stm32f4xx_it.c
|    stm32f4xx_it.h
|    system_stm32f4xx.c
|    tiny_printf.c
|
├──app
|        assert.h
|        intellisense.h
|        macros.h
|        memory.cpp
|        memory.hpp
|        queue_alloc.hpp
|        timer.cpp – 타이머 구동 로직입니다.
|        timer.hpp
|        utility.cpp
|        utility.h
|
├──deprecated
|        cppmain.cpp.2
|        cppmain.cpp.3
|        cppmain.cpp.4
|        cppmain.cpp.5
|        hw_motor.cpp.0
|        lidar.cpp
|        lidar.hpp
|
├──device
|        hw_Interrupts.cpp
|        hw_mac.h          - 이 기기를 나타내는 MAC 주소입니다.
|        hw_motor.cpp
|        hw_motor.hpp      - 모터 구동 인터페이스입니다.
```

```
|       hw_netrw.cpp      - 네트워크 통신 구현 파일입니다.
|       hw_proximity.cpp
|       hw_proximity.hpp  - 근접 센서 구동 모듈입니다
|       hw_s2pi.cpp
|       hw_timer.cpp
|       hw_usbrw.cpp
|       tlcd.cpp
|       tlcd.h
|
├──example
|       discovery_specific.c
|       discovery_specific.h
|
├──ioLibrary                -
|   |   .cproject
|   |   .project
|   |   liblinks.xml
|   |
|   ├──Appmod
|   |   └──Loopback
|   |           loopback.c
|   |           loopback.h
|   |
|   ├──Ethernet
|   |   |   socket.c
|   |   |   socket.h
|   |   |   wizchip_conf.c
|   |   |   wizchip_conf.h
|   |   |
|   |   └──W5500
|   |           w5500.cpp
|   |           w5500.h
|   |
|   └──Internet
|       ├──DHCP
|       |       dhcp.c
|       |       dhcp.h
|       |
|       └──DNS
```

```
|                   dns.c
|                   dns.h
|
├──platform
|       argus_log.cpp
|       argus_timer.cpp
|       cppmain.cpp        - 프로그램 진입점 및 주요 동작 정의. 매우 중요합니다.
|       hw_netrw.hpp       - 네트워크 함수성 및 통신 프로토콜 구현
|       hw_timer.hpp       - 하드웨어 타이머 기능 정의.
|       hw_usbrw.hpp
|       irqlock.cpp
|       irqlock.hpp
|       main.c
|       s2pi.hpp           - SPI 통신 구현
|       wiz_spi_wrap.cpp
|
└──scanlib
        connection.hpp
        protocol.cpp        - 통신 프로토콜
        protocol.hpp        - 통신 프로토콜 2
```