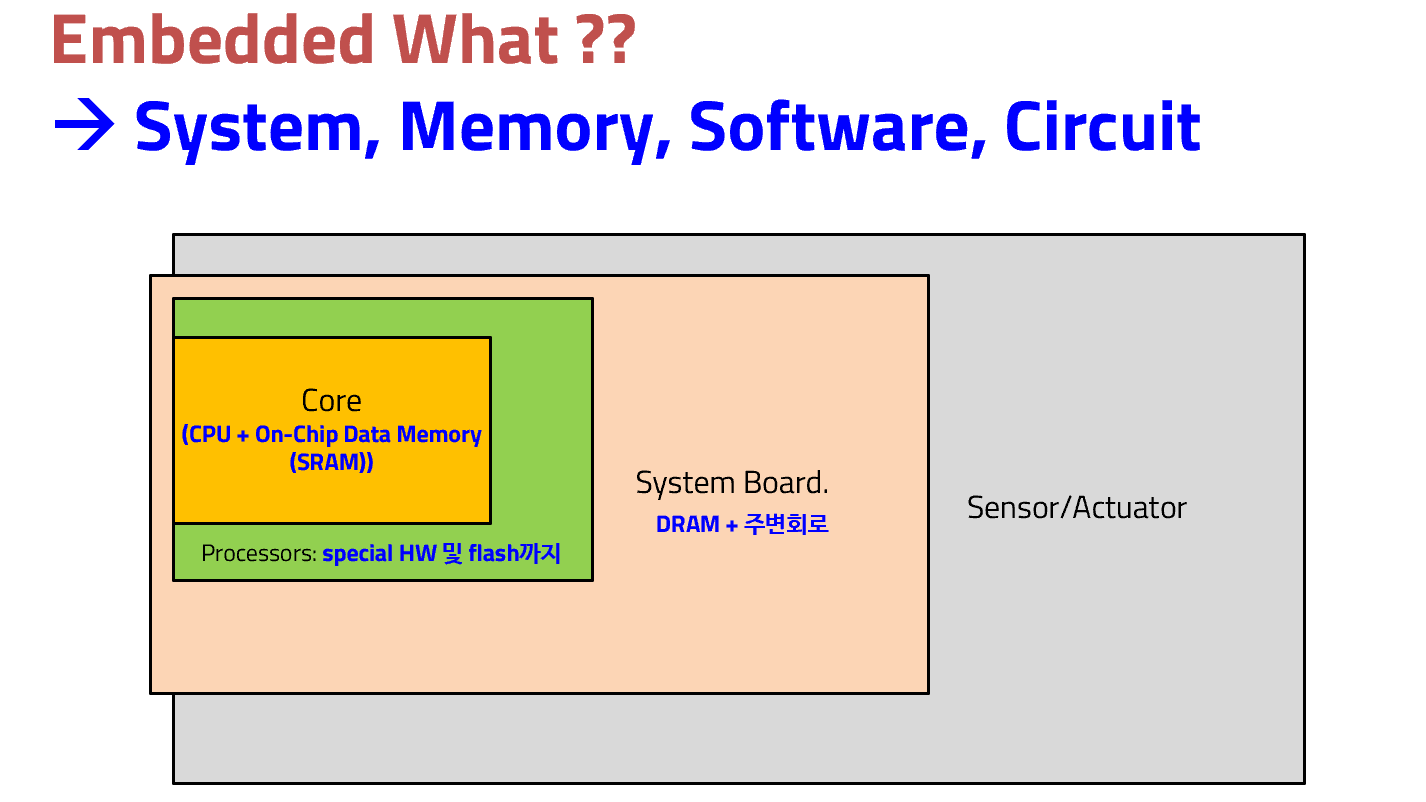
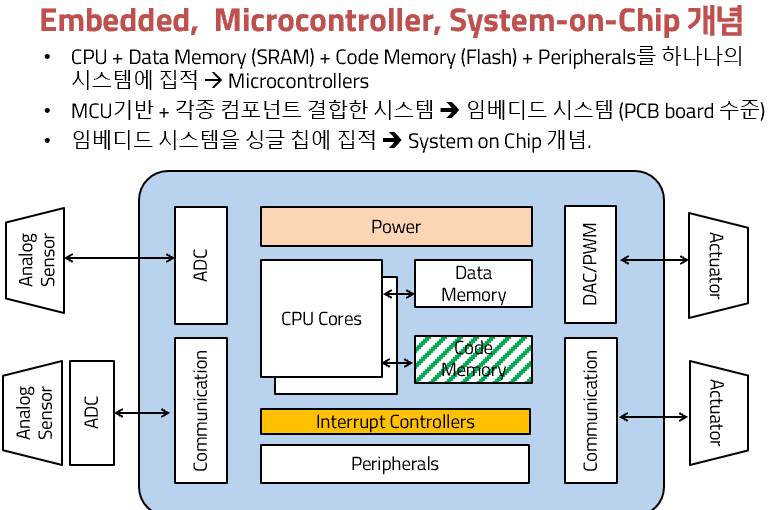
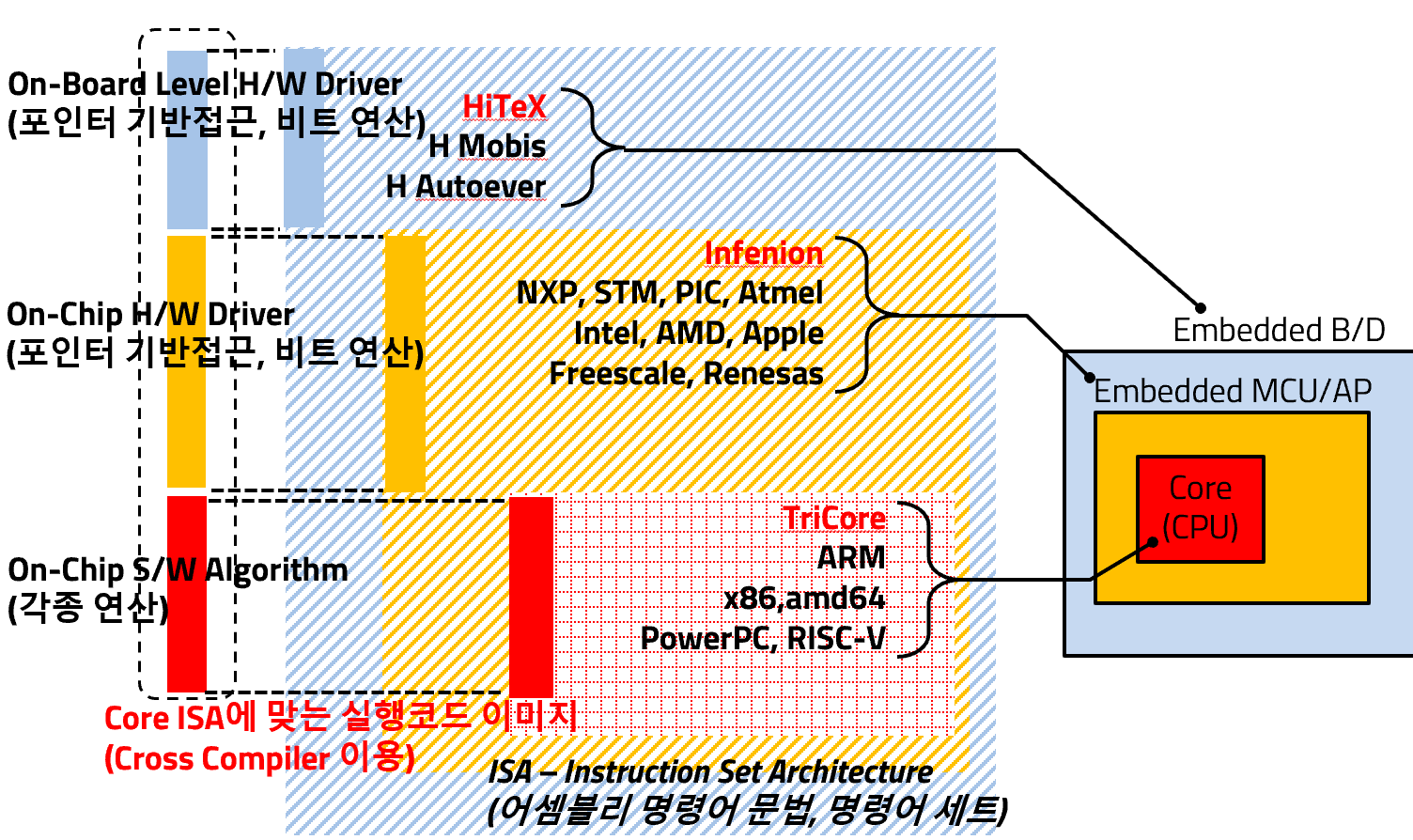
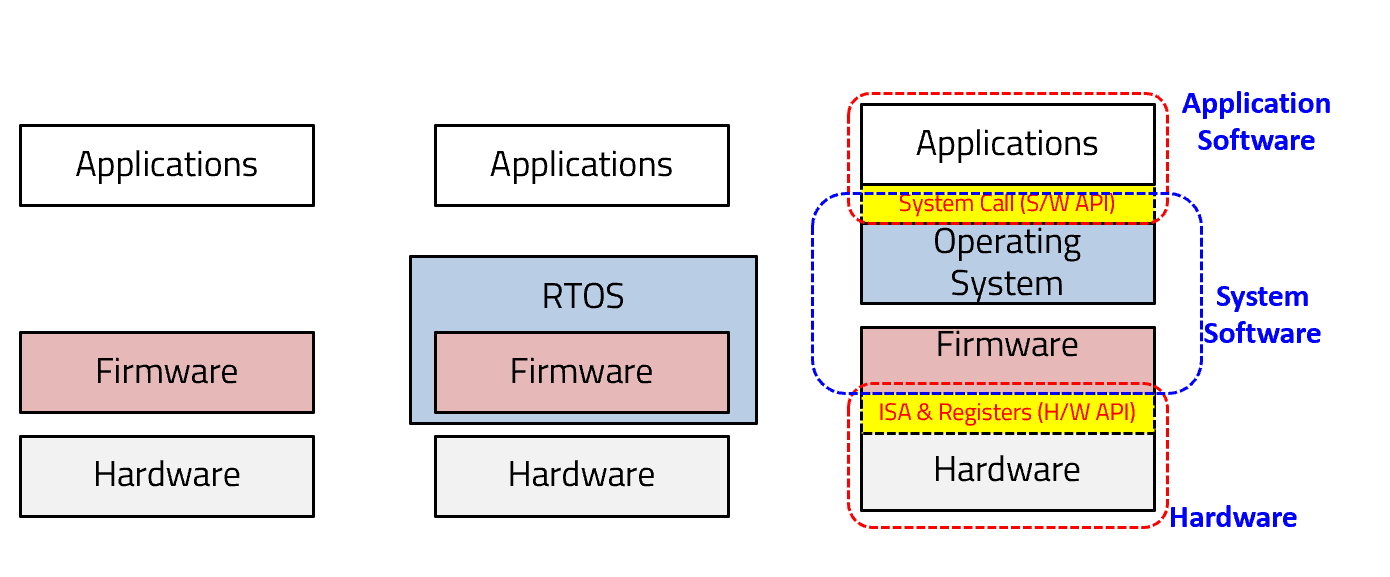
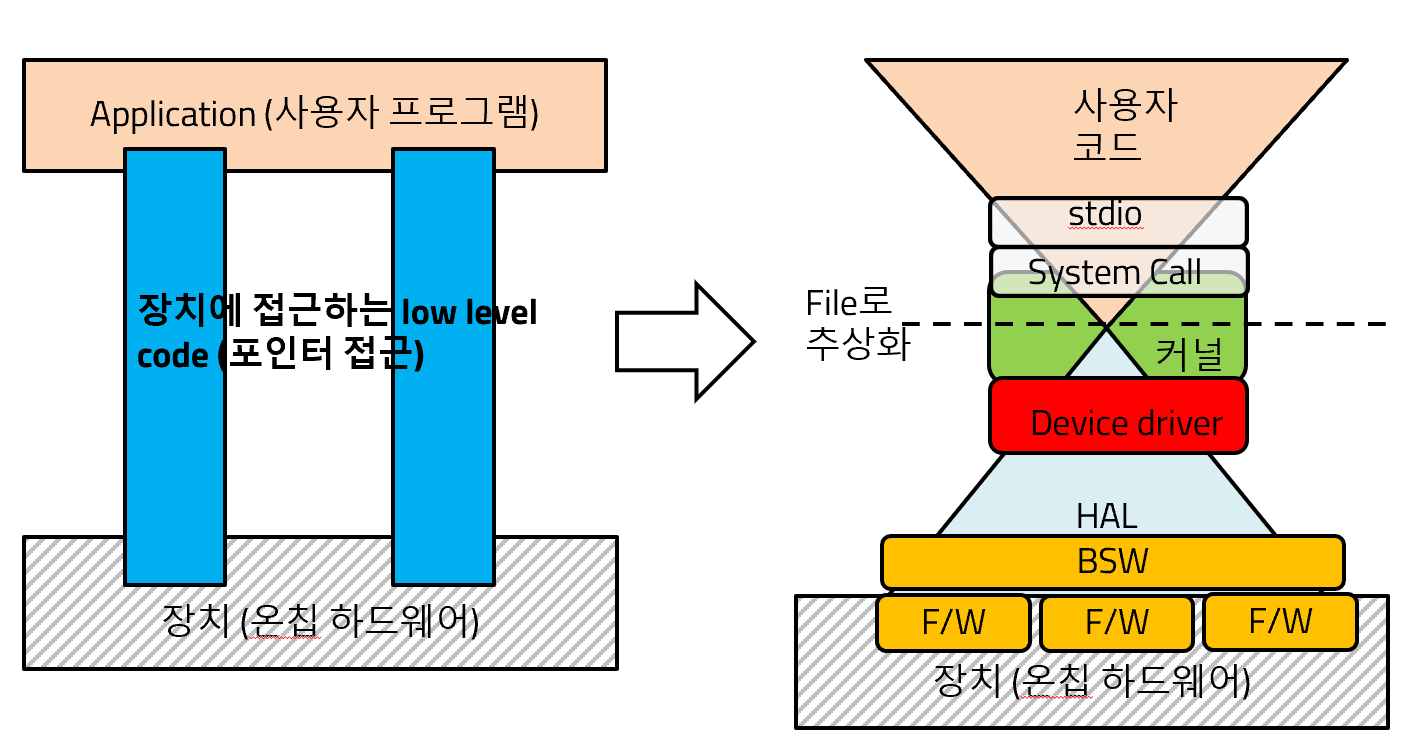
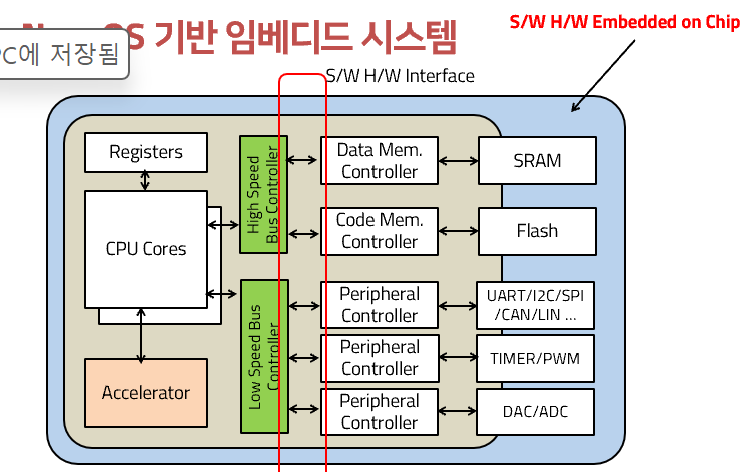
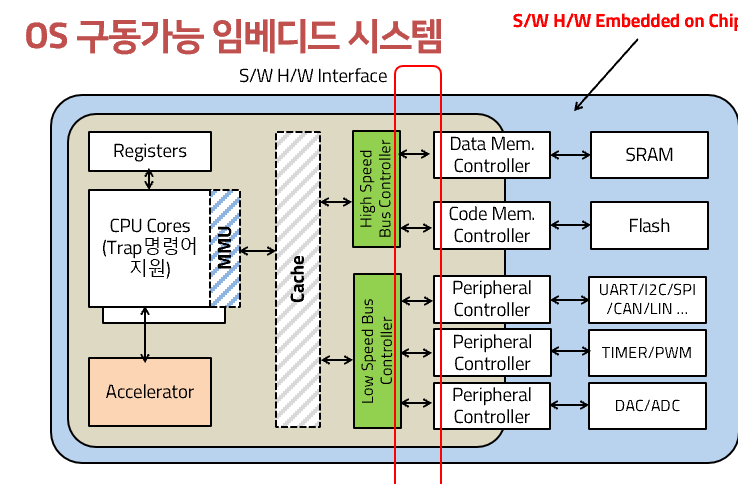
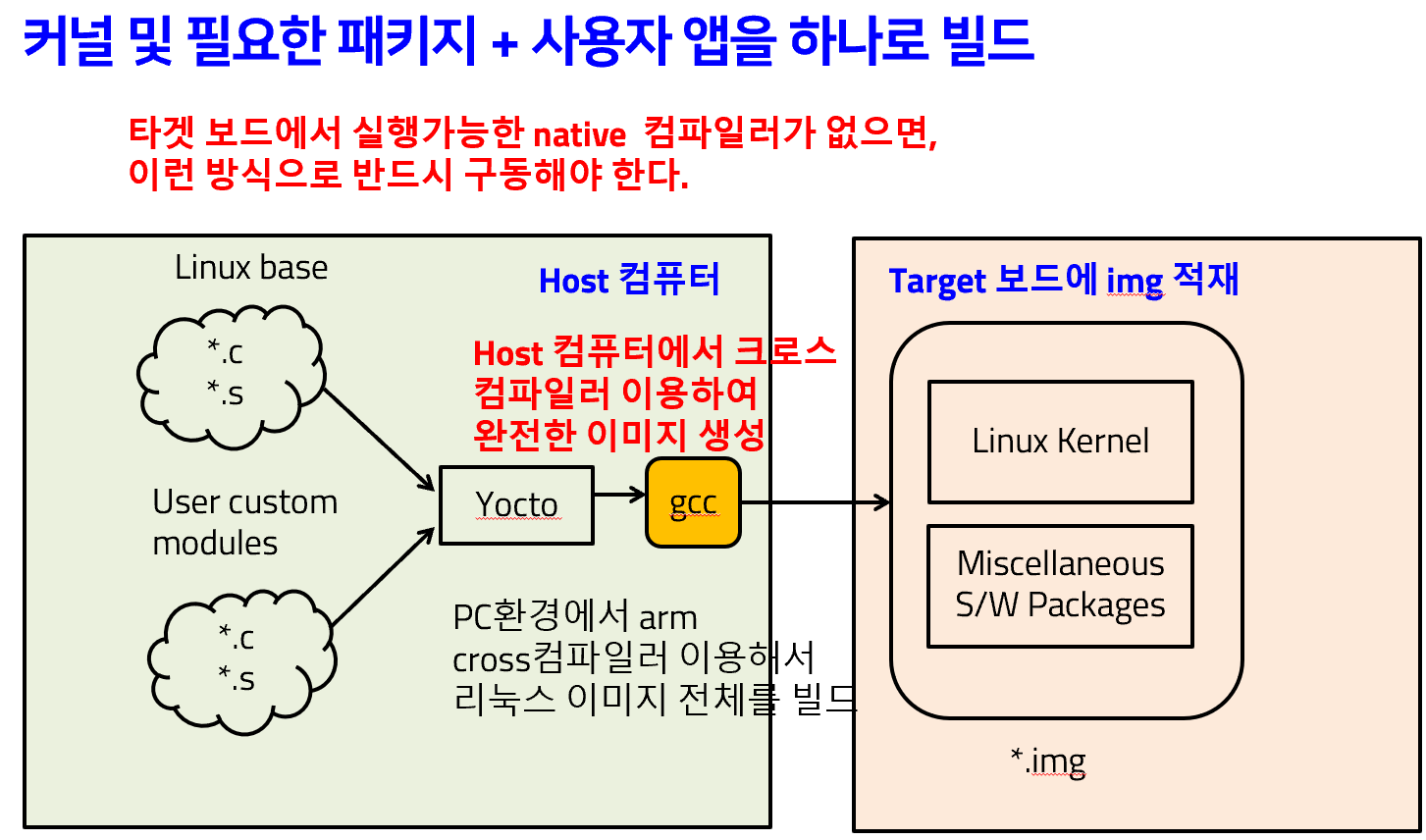
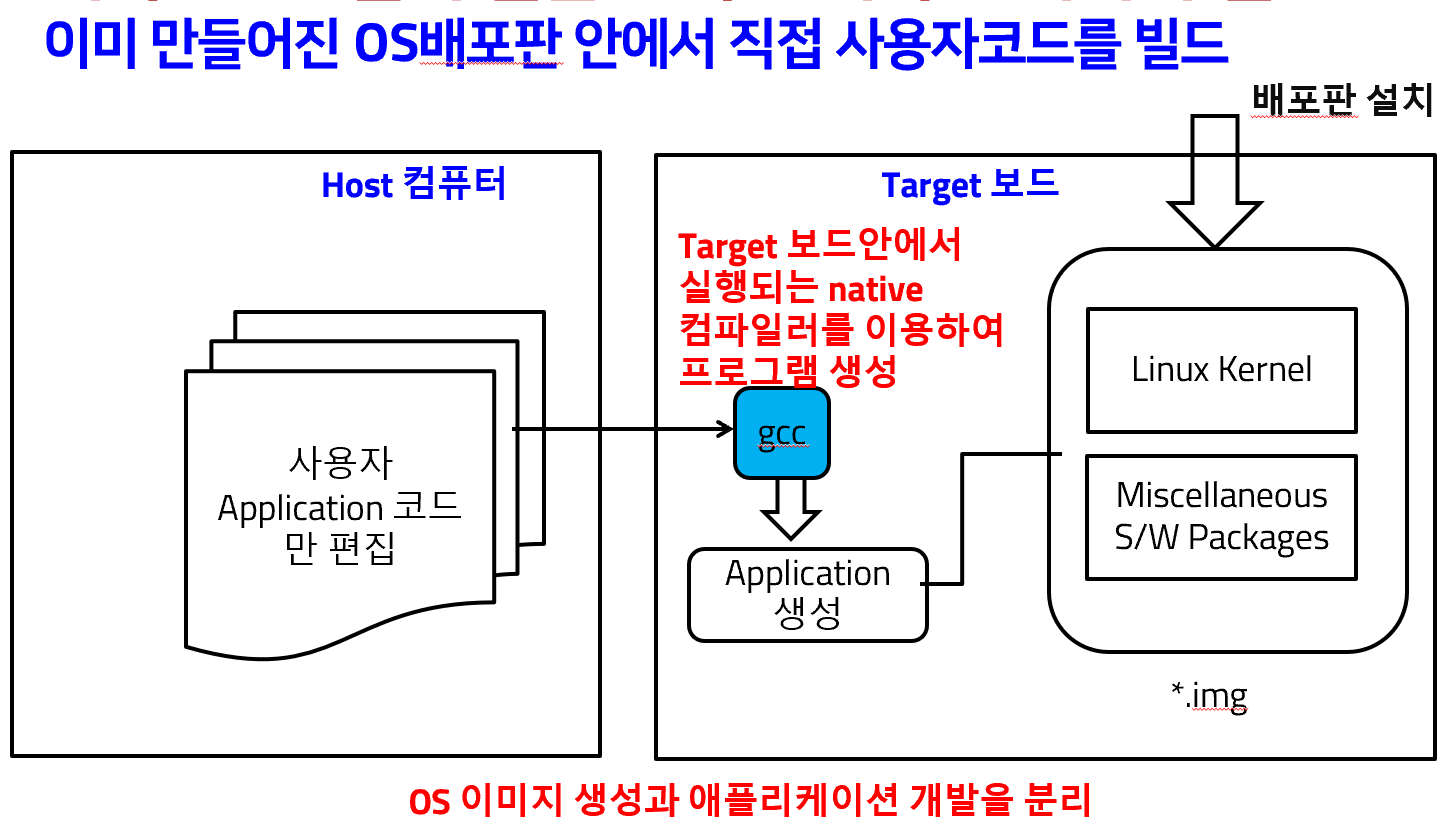
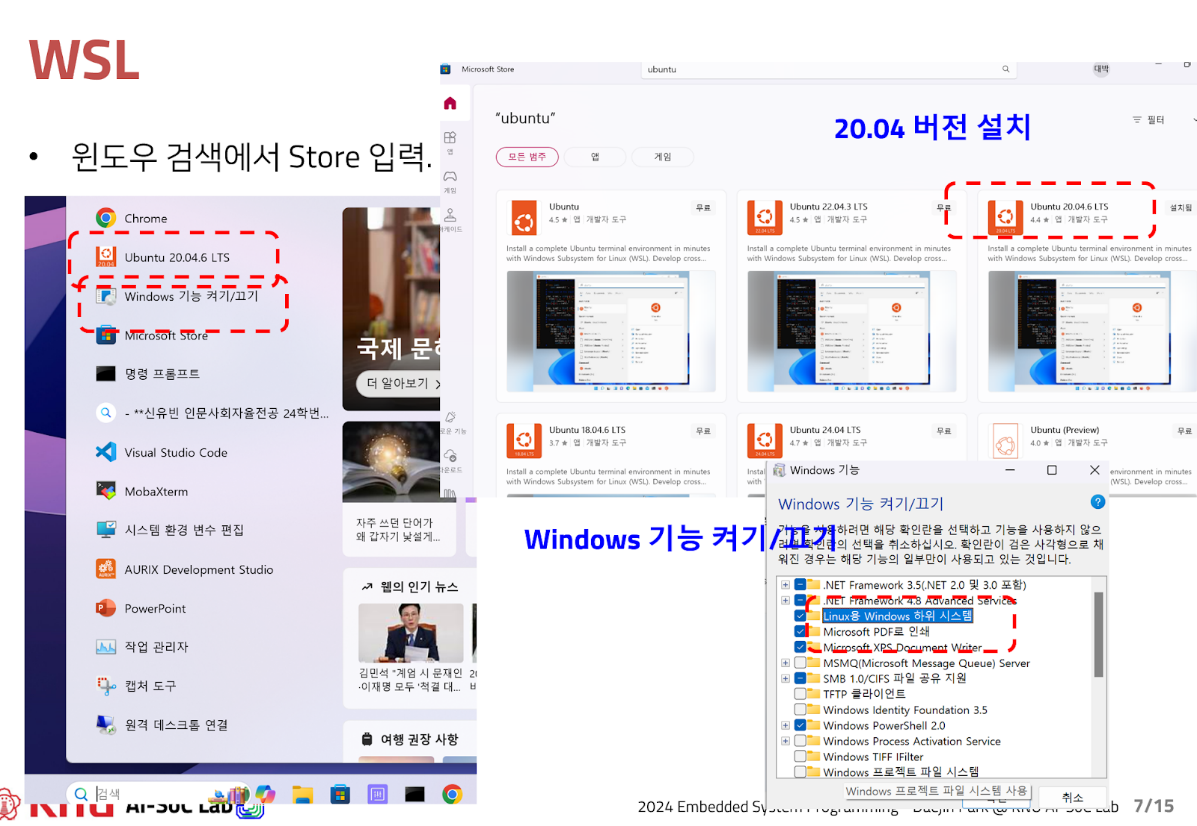
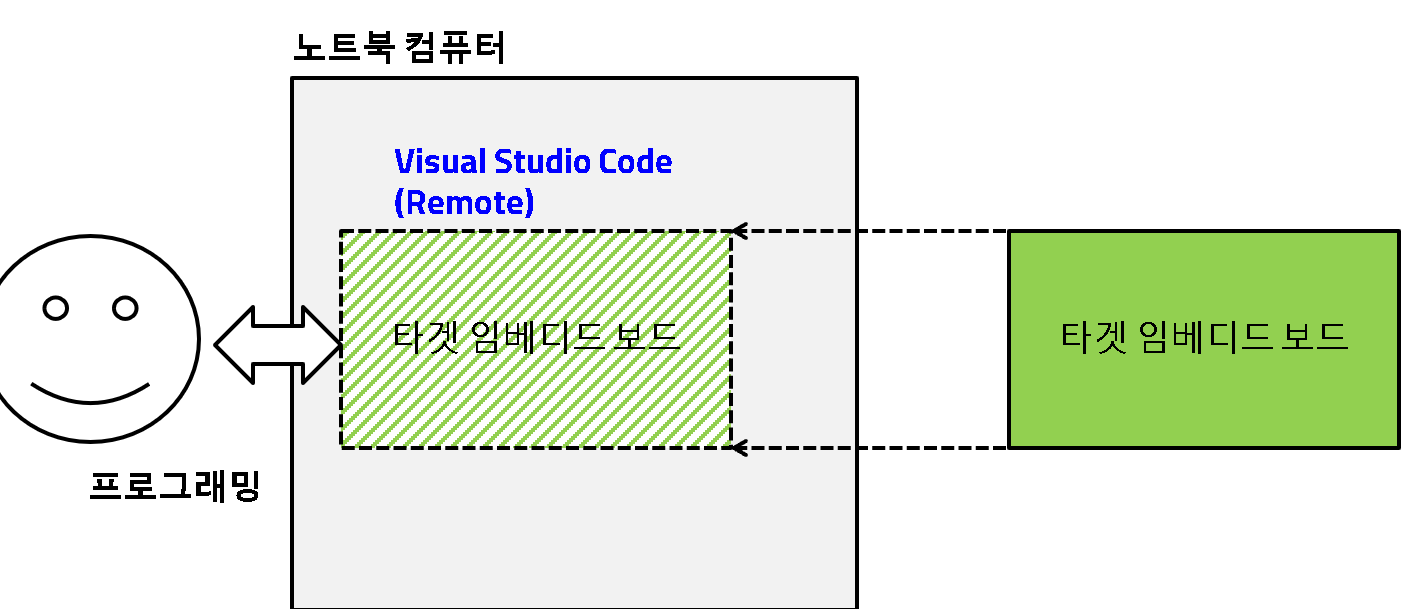
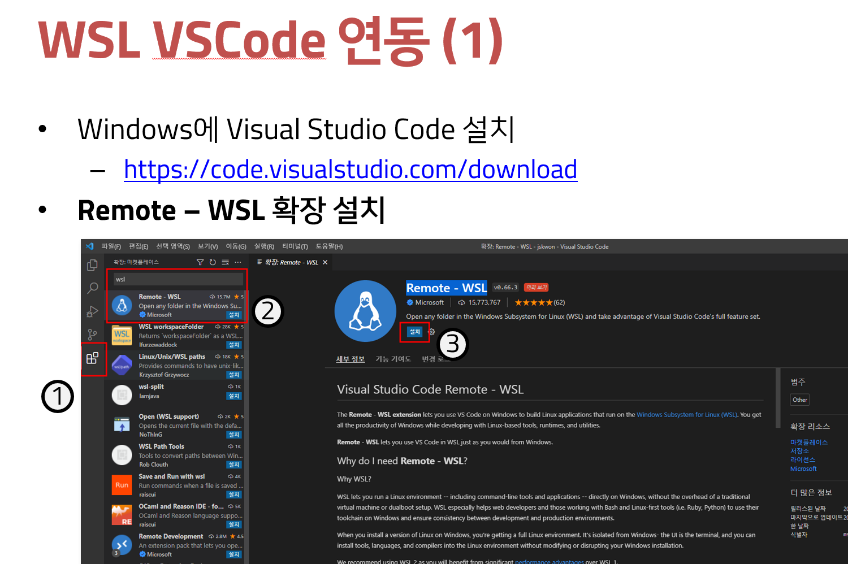
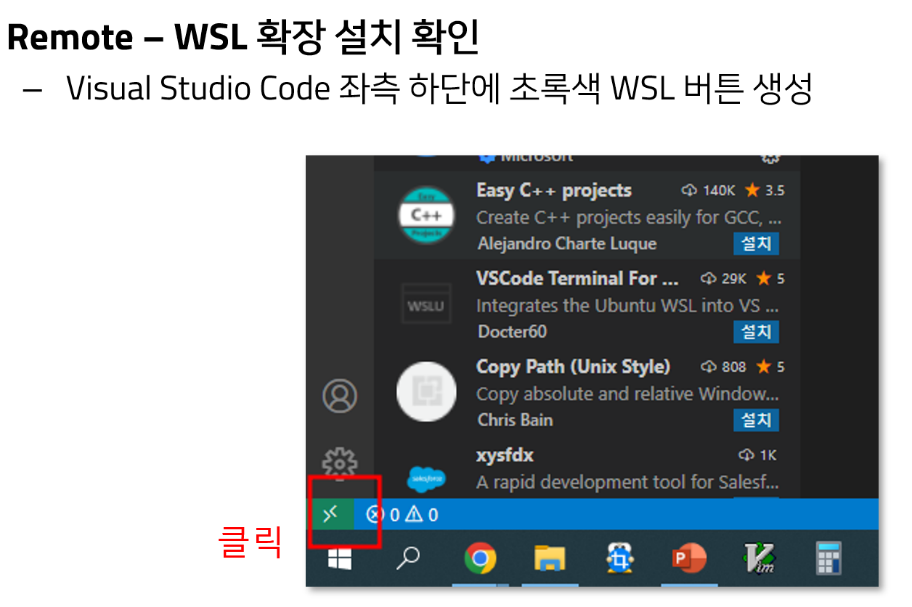
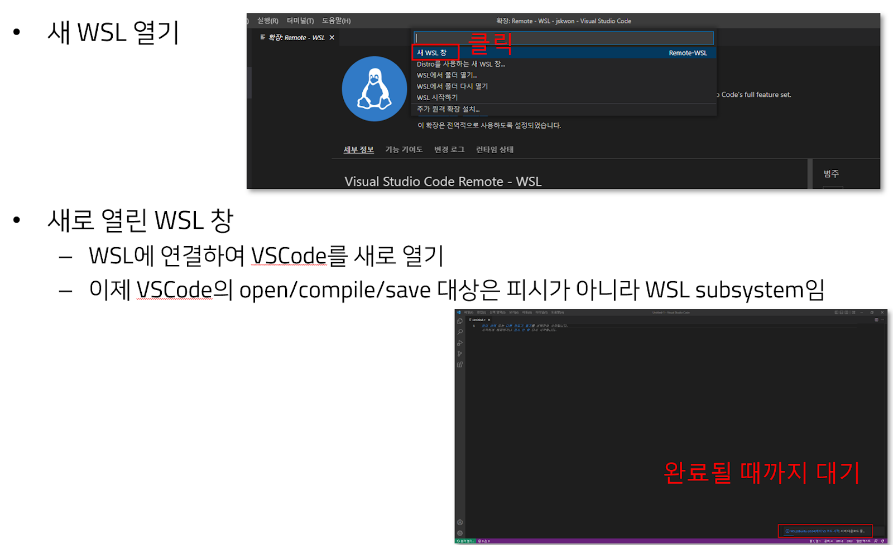
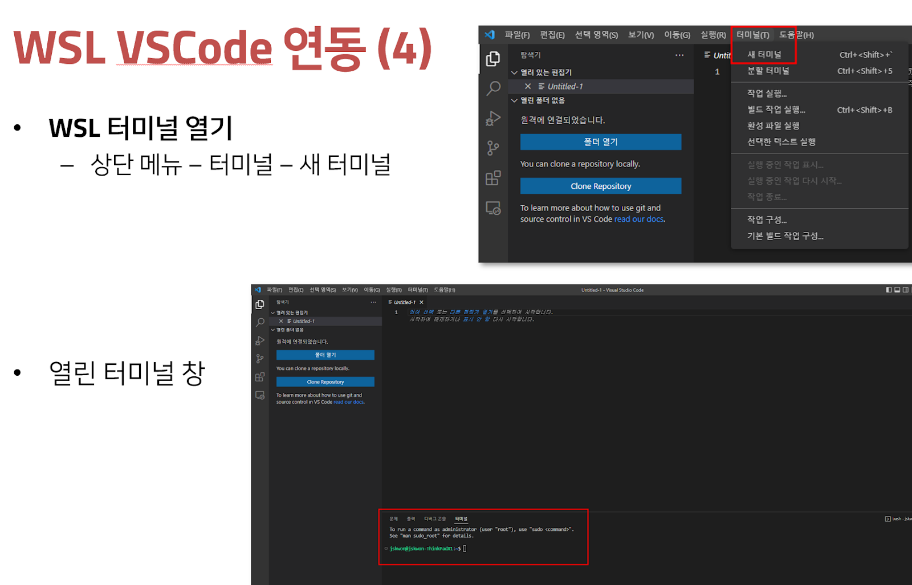
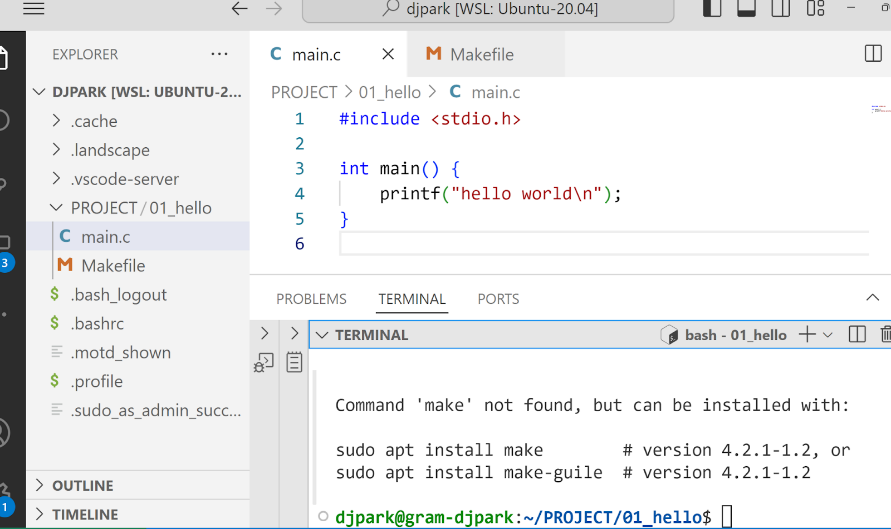
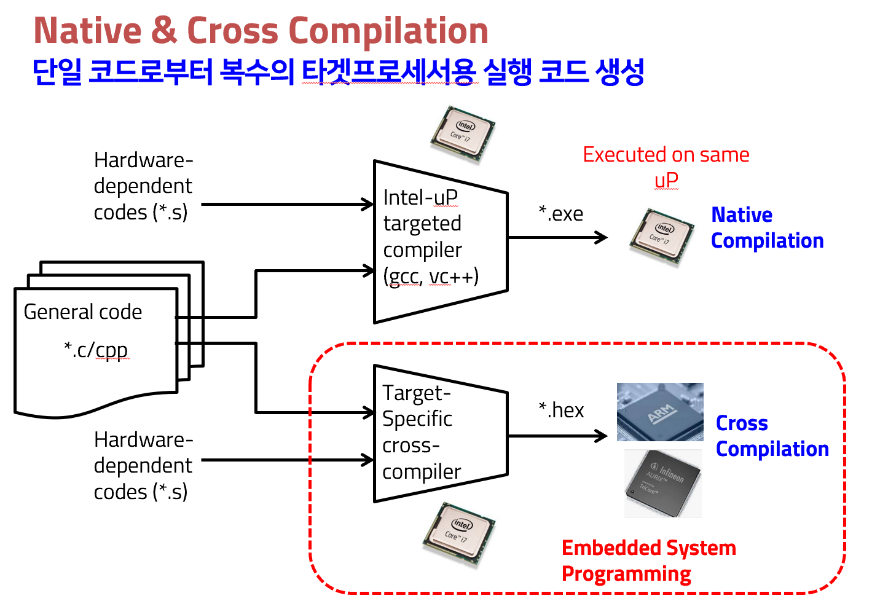
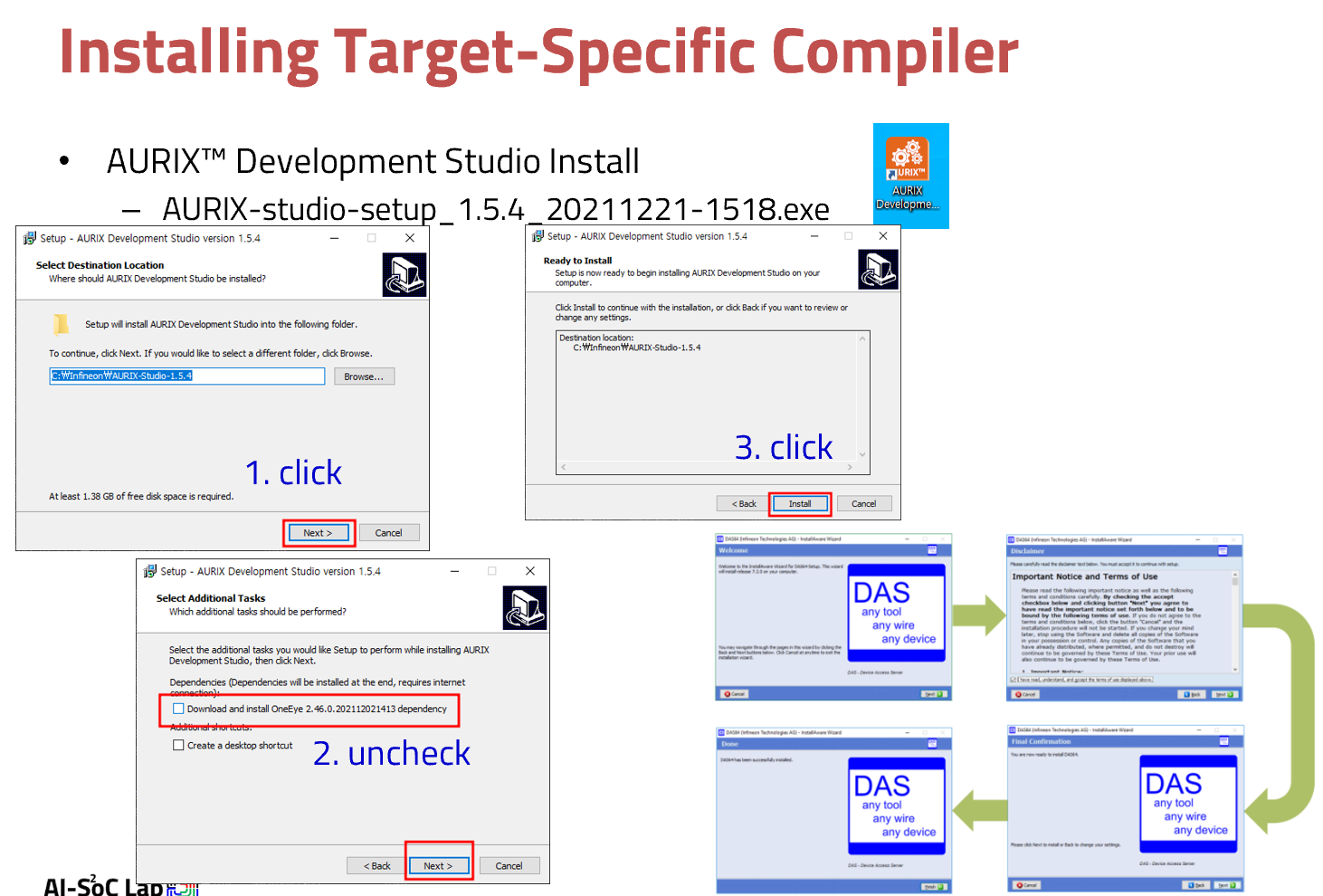
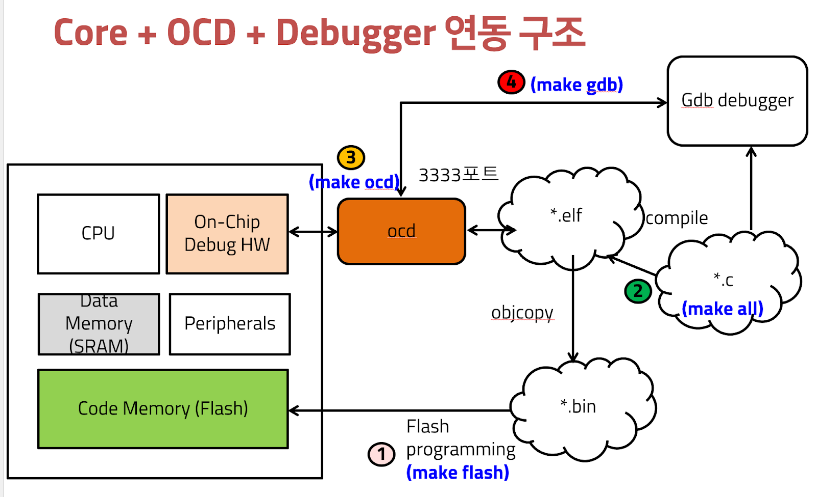
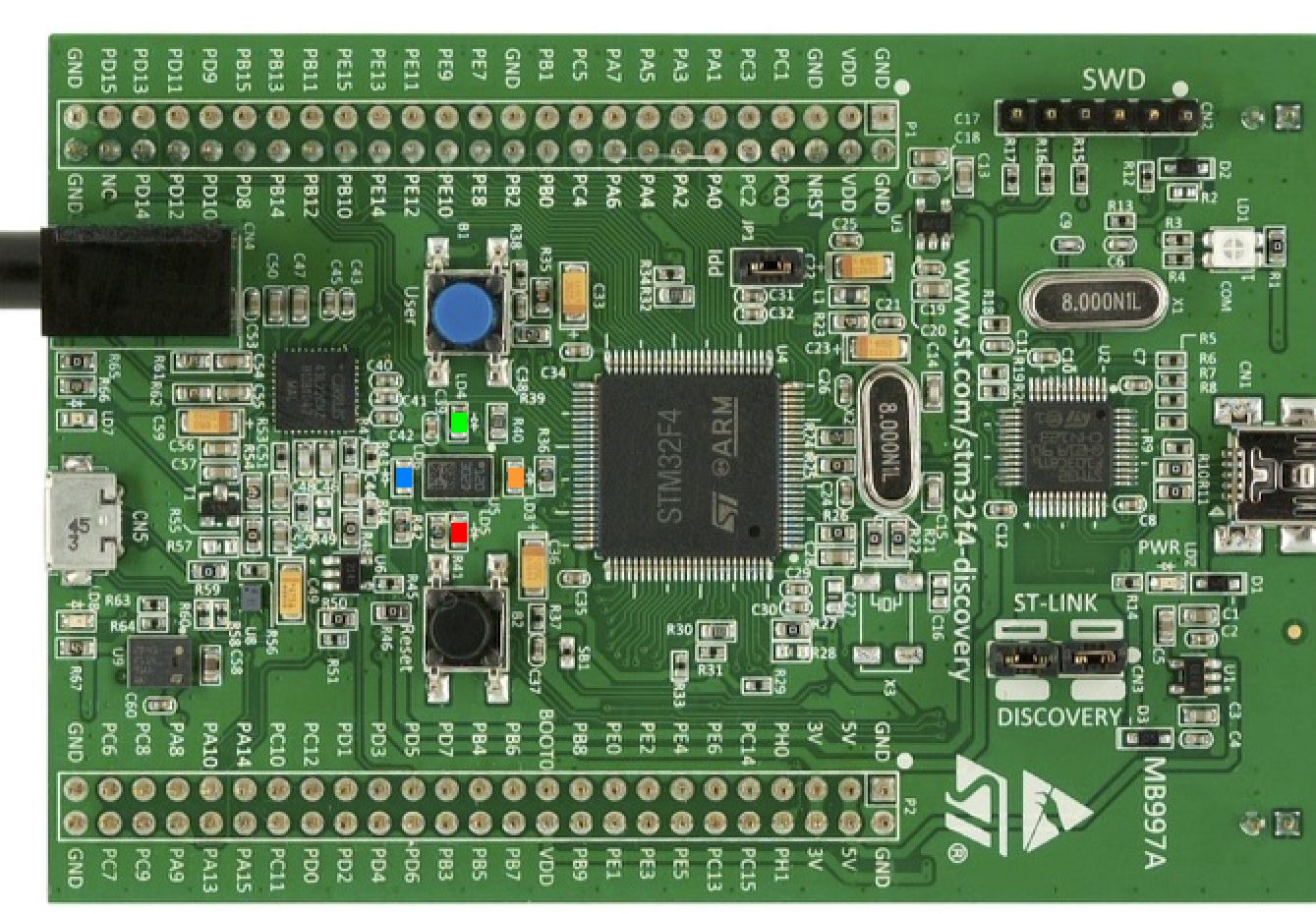
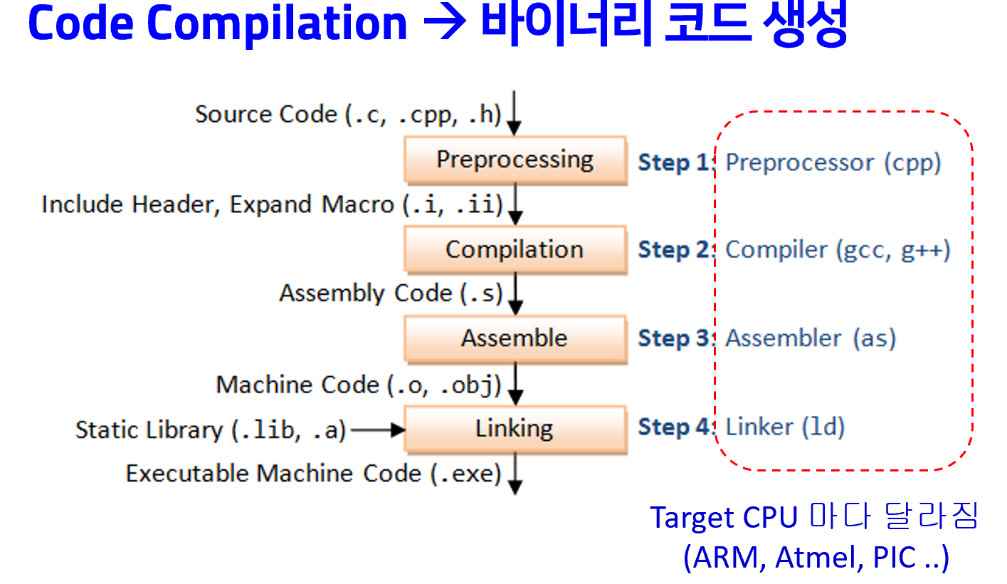
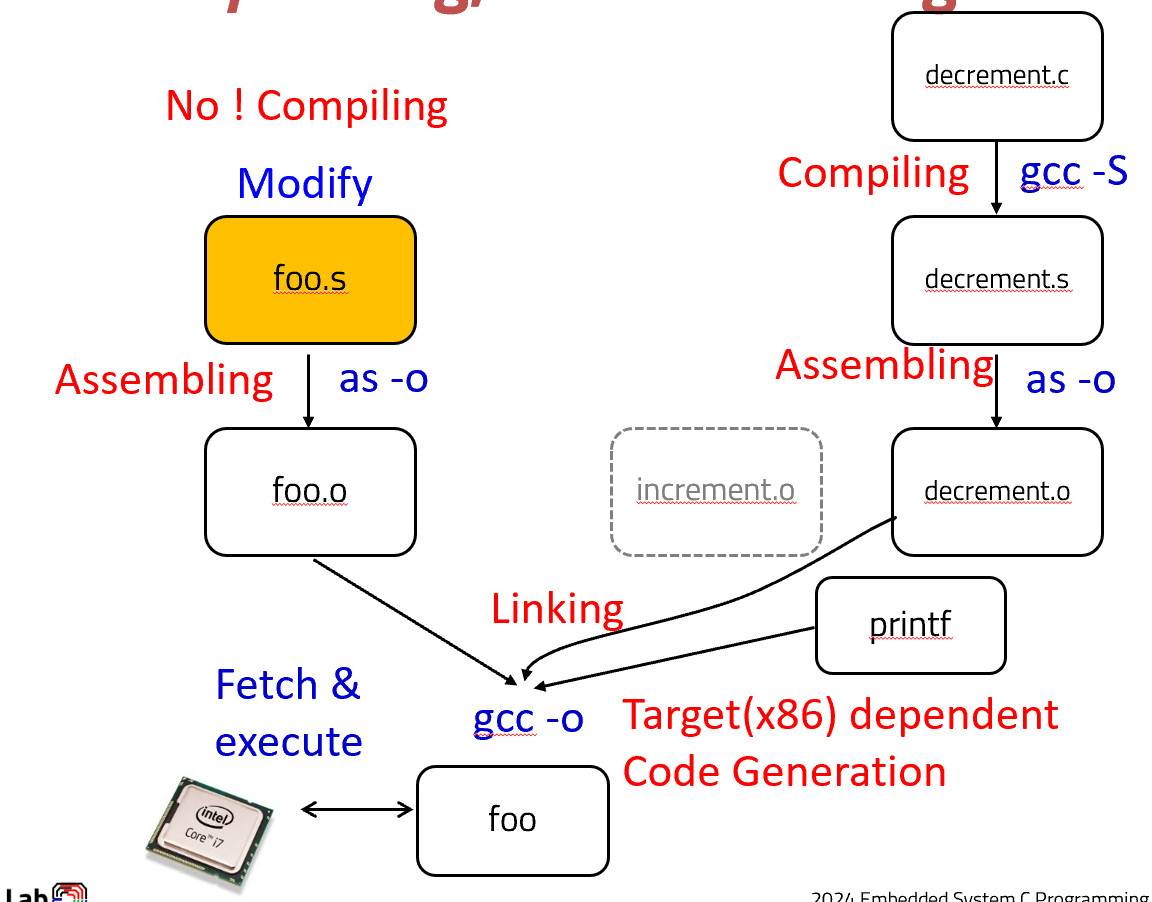
* **1. 임베디드 시스템 및 시스템 프로그래밍이란 ?**
  + **(1) 임베디드 시스템이란.**
    - **A. Firmware의 의미**
      * Software가 반도체 (flash)에 한번 다운로드 (programming)되고 나면
        + **더이상 지워지지 않고 단단하게 남아 있다는 의미.**
      * **임베디드 SW 내장형 시스템의 장점과 단점**
        + **장점 - SW가 시스템에 탑재되고 나면 단독동작가능 (고성능/고신뢰성)**
        + **단점 - HW안에 내장되고 나면 수정이 어려움,**

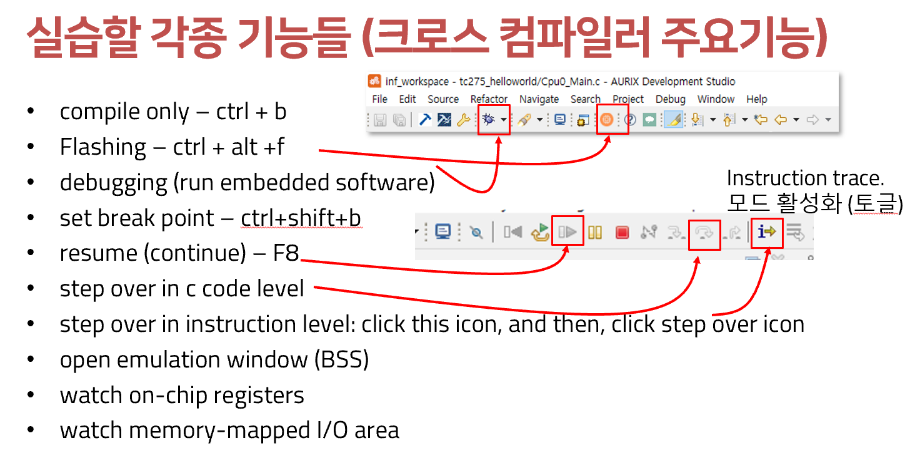
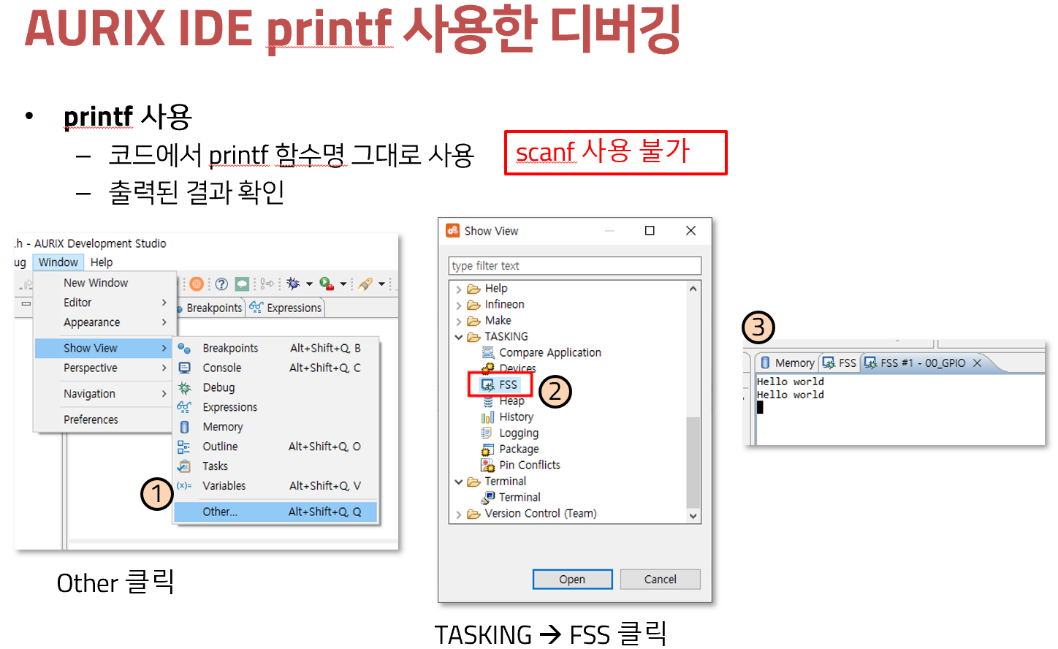
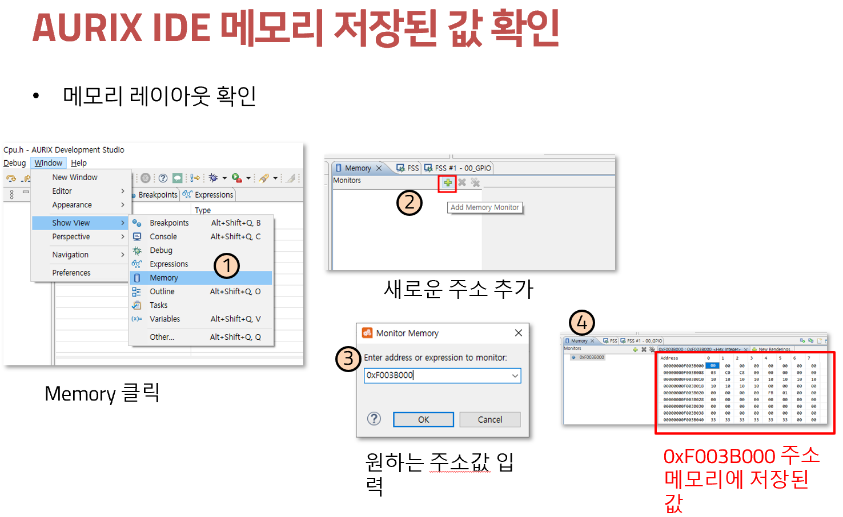
**상황 인지를 통해 adaptive한 SW 동작이 어려움**

**문제(불량,버그)가 발생시, 수정이 용이하지 않음**

* + - * + **=> 이러한한 단점을 극복하기 위한 임베디드 SW개발 방법을 배우게 됨 (본 강의)**
    - **B. 무엇을 Embedded 했는가 ?**
      * **(1) cpu(core, ex. x86, arm, tricore, RISC-V, ) 와 Memory (code, data -> flash, sram으로 실현) 까지를 를 임베딩했을때 임베디드 프로세서라고 한다.**
      * **(2) 다양한 내장 회로 (timer, adc, dac, uart, can, lin 등)까지 탑재해서 임베디드 MCU, 임베디드 AP급이라고 해서 제품으로 판매되고 있음.**
      * **(3) 그리고 PCB회로 보드 레벨에서 DRAM, 및 각종 아날로그 회로 (파워)까지 내장한 시스템을 임베디드 시스템이라고 한다.**
      * 
      * 제어시스템의 최종 동작의 품질/성능/요류들의 근원적 시작은 임베디드 SW로부터 출발하는 경우가 많다.
      * 1. main cpu (프로세서)
      * **2. 메모리 Embedded (이게 가장 중요한 block임)**
        + 썼다 지웠다 할 수 있는 Data Memory (SRAM)
        + 한번 쓰면 안지워지는 Code Memory (Flash)
        + **=> 크기, 스피드, 대역폭, 신뢰성 특성. -> 시스템 특성을 좌우함.**
      * 3. Digital 회로 (HW)
        + timer, gpio
        + spi, uart, i2c, can, lin
      * 4. Analog회로 (CKT)
        + power
        + ADC
        + DAC
      * => 아날로그회로, 디지털회로, 메모리를 하나로 집적 + 소프트웨어
        + => 칩 내부에 flash메모리에 적재되면, firmware가 된다.
        + => System on chip -> Software on chip/systems
        + 
    - **C. 인터럽트 컨트롤러의 필요성**
      * **다양한 HW가 하나의 온칩 시스템 버스에 Embedded되었기 때문에 항상 contention (충돌)이 발생.**
      * 다양한 내장회로는 자신의 데이타 처리를 위해 cpu core쪽으로 요청을 하게 된다.
      * 인터럽트 컨트롤러는 여러 요청을 교통정리해서 (우선순위에 맞게) cpu가 하던 일을 멈춰세우고, 개별 요청에 따른 처리 절차 (인터럽트 서비스 루틴)을 수행하도록 해줌.
      * 마이크로컨트롤러는 필수적으로 인터럽트 컨트롤러를 내장하고 있음 (다양한 회로를 온칩에 내장했기때문에, 이러한 회로들의 요청을 처리해주는 장치가 필요함)
  + **(2) HW-SW Stack Decoupling**
    - **맨먼저 core (cpu+on-chip registers) 부터 확인해야 한다.**
      * **core는 HW가 이해하는 언어의 규격이 바로 instruction set architecture (ISA, 어셈블리 문법으로 생각하면됨)를 이해하는 HW이다..**
      * 그래서 내가  cpu에게 어떤 일을시킬려면 타겟코어에서 이해하는 어셈블리로 코딩하거나 컴파일러 이용해서 c -> asm변환을 해야 한다**. (즉 core가 결정되면, 지원되는 크로스 컴파일러를 같이 생각해야 한다. 지원여부, 성능, 호환성 등)**
      * 
    - **임베디드 프로세서 제작 칩 업체는 이러한 core와 함께 다양한 하드웨어를 내장하여 하나의 제품으로 판매**
      * **내부에 있는 HW장치에 주소기반으로 (포인터 문법 배워야 하는 이유) 접근**
        + 내가 원하는 비트를 셋팅하고 장치를 제어할 코드를 구현해야 하는 영역이다
      * **low level driver의 구현이 필요한 이유.**
        + => 순수 알고리즘하고 low level driver를 분리해서, 향후에 칩 벤더가 바뀌더라도 쉽게 포팅될수 있는 소프트웨어 구조를 정의해야 한다.
    - **사용자 코드안에 내장 하드웨어에 접근하는 코드가 혼재 (tightly coupled)되어 있는 구조가 아니라, 앞으로는 사용자 코드와 하드웨어 접근 코드를 분리해야 한다.**
      * **MCU 기반 (왼쪽 2개)**
      * **AP급 기반 (오른쪽)**
      * 
    - **내장 하드웨어가 바뀌어도 (칩 교체) 구동 드라이버만 교체해주면 상위 사용자 제어 코드는 그대로 호환되도록 구현해야 함.**
      * 
  + **(3) 임베디드 SW 실행 2가지 방식에 따른 하드웨어 구조**
    - **소프트웨어 구동방식에 따른 회로 구조 2가지**
    - **A. OS가 올라가지 않고 직접 HW제어 방식 - Firmware 기반 MCU**
      * 
      * **flash메모리에 내가 구현한 C코드가 적재될 것이고, cpu.는 여기에 direct access 해서 명령어 fetch하고 실행하는 방식을 취함**
        + 코드가 크지 않고, dram을 경유하지 않고 직접 읽고 실행하면 느리긴 하겠지만 일정한 성능 (느리더라도)을 유지하게 된다.
        + 따라서 flash 접근 속도가 SW실행 속도를 결정함.
      * **연산결과를 SRAM (data memory)에 저장**
        + SRAM크기가 결국 프로그램의 실행시간에서 얼마나 여유있게 실행되는지를 결정함 (변수를 크게 잡고 사용가능함)
    - **B. OS가 탑재되어 가상메모리 이용한 Application S/W구동 AP (Application Processors)**
      * 가상메모리를 위한 MMU회로가 반드시 탑재되어야 OS를 올릴 수 있다.
      * 
      * 프로그램을 실행중에도 추가로 설치, 실행순서에 상관없어야 하고, 프로그램 중간에 종료할수도 있고 새로 시작할수도 있고 하나의 프로그램을 두번 이상 실행도 할수 있으려면
        + => 가상 메모리 개념이 필요함
        + => 가상 메모리 기반 소프트웨어 실행을 위해 내장 하드웨어로 MMU가 탑재되어야 하고, MMU를 제어하는 운영체제가 필요하다.
* **2. 임베디드 프로그래밍 개발환경 구축**
  + **(0) 호스트-타겟간 개발환경 방식차이**
    - **A. 커널 + 베이스 시스템을 호스트에서 컴파일하여 결합**
    - 
    - **B. .타겟보드에 실행가능한 컴파일러를 포팅한뒤, 타겟보드에서 실행 프로그램 직접 빌드**
    - 

* **(1) Host 머신에 개발 환경 구축**
  + cross 컴파일러 설치
    - gcc
    - aurix studio
    - tasking compiler
  + 기타 등등
    - 일일이 설치해야 함.
    - make, awk, sed, perl, python
    - 각종 바이너리 분석툴들.
* **(2) Host 머신에 임베디드 SW 구동 레이어 설치.**
  + cygwin
  + virtualbox, vmware
  + **WSL**
  + 듀얼부팅으로 직접 OS설치.
* **(3) WSL (windows subsystem for linux)위에 ubuntu설치**
  + 1. Microsoft Store에 가서 Ubuntu 20.3 설치.
    - 
  + 2. Powershell을 관리자 모드로 실행후 아래 입력
    - **dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart**
    - **dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart**
  + 3. 재시작
    - 재시작후 다시 powershell 관리모드로 실행후
    - **wsl --update 명령어 입력**
  + 4. ubuntu 실행
    - **검색에 ubuntu 입력후 아이콘 클릭**
      * 최초 한번은 id, password 생성해야 함.
    - **만약 root로 바로 로그인된다면 사용자 계정을 수동으로 계정 생성 해야함 (아래명령어 참고)**
      * **adduser gildong**
      * **passwd gildong** 암호 수동설정도 가능.
      * su - gildong
      * 이후에 sudo vi /etc/sudoers 파일을 편집해서, 사용자를 추가해주어야 함. (실습)
  + 5. 필요한 툴 설치
    - **sudo apt-get update**
    - **sudo apt-get install gcc make build-essential git**
  + **6. 예제 다운**
    - **git clone** [**https://github.com/AI-SoC/EMSOFT\_C\_20241126**](https://github.com/AI-SoC/EMSOFT_C_20241126)
* **(4) 원격 편집(코딩)환경 구축**
  + 원력 타겟보드에 있는 파일을 로컬 폴더에 있는 것처럼 매핑
  + 로컬에서 편집하지만 결과는 타겟보드에 적용됨,
  + **1. visual studio code (최신) 설치**
    - 
  + **2. WSL을 타겟으로 remote 접속.**
    - VNC, RDP하고는 개념이 좀 다르다.
    - 불편한 terminal 접속환경을 극복하면서, 원격지 리소스를 host로 미러링하는 강력한 기능이다.
    - 
    - 
    - 
  + **3. 로컬에 open folder해서 마운트**
    - ****
  + **4. 원격 터미널 열기**
    - 
* **(5) 임베디드 SW 이미지 빌드 (native compilation vs cross-compilation**
  + 
  + 기본적인 제어 알고리즘 구현은 native환경에서 구현, 코드 검증해야 함 (빠르게 컴파일, 실행해볼수 있음)
  + 어느정도 알고리즘 구현이 완성되면 크로스 컴파일러 이용해서, 타겟보드에 맞는 코드를 생성하여 flash에 적재해서 프로그램 실행특성 검증해야 함
* **(6) 크로스 컴파일러 설치**
  + **A. TriCore용 Tasking  컴파일러 설치.**
    - 
  + **B. ARM 프로세서용 컴파일러 설치.**
    - **sudo apt-get install gcc-arm-none-eabi**
    - **sudo apt-get install gdb-multiarch**
* **(7) 코드 컴파일, 및 코드 적재.**
  + **컴파일, flashing, debugging**
    - **A. makefile**
      * **compile전과정의 명령어 실행절차를 표현**
      * 단순히 명령어 나열이 아니라
        + **어떤 출력이 생성될려면 어떤 입력이 필요하고**
        + **그렇게 되기 위한 명령어는 어떤게 필요한지 기술한 계층적 명령어 집합임**
      * **dependency기반으로 conditional compil**e을 수행함.
    - **B. compile (preprocessing, compile, assembling, linking)**
      * **compile, assmembing, linking, 결과로 \*.hex생성됨**
      * 실행 절차는 makefile에 담겨있음.
      * **디버깅 정보를 담은 \*.elf파일도 생성됨**
    - **C. flashing**
      * \*.hex파일을 온칩 flash에 다운로드함 (굽는다는 표현을 사용)
      * **flash에 쓰기 위해서는 일단 다 erase한 다음에 새로 써야 함** (시간이 많이 걸림), 쓰고 나서 제대로 써졌는지 읽어서 verify까지 수행함.
    - **D. debugging**
      * 칩내부 flash에 구워져 있는 \*.hex 내용은 완전한 **타겟 프로세서에 이해할수 있는 기계코드assembly수준의 바이너리코드**임. **(C코드의 존재는 사라짐)**
      * **assembly코드가 flash메모리 어느 주소에 배치되었고 그게 C코드의 몇번째 라인지 정보를 담는 \*.elf파일**을 디버거에 올려두고 실시간 비교하면서 임베디드 SW실행흐름을 제어함.
  + 전체 동작구조
    - 
  + QEMU 보드 에뮬레이터 혹은 타겟보드에 적재.
    - 
* **3. Hello World (임베디드 SW를 온칩 메모리에 올려보기)**
  + **(1) Preprocessing, compile, assembling, linking (세부적인 컴파일 절차를 수행).**
    - **컴파일러: C코드를 입력받아서 어셈블리 코드생성**
    - **어셈블러: 어셈블리 코드를 입력받아서, 기계코드(목적코드)를 생성**
    - **링커: 여러개의 목적코드를 입력받아서, 최종 실행코드를 생성.**
    - 
    - 3 스텝으로 분리하면, 일부는 어셈블링, 일부는 컴파일 등으로 하이브리드하게 코드를 빌드할수 있다.
    - 
  + **(2) Makefile**
    - **conditional compilation개념.**
    - **문법**
      * target: requirement
      * -->탭  실행커맨드
    - **target, requirement는 재귀적인 표현이 가능함**
      * 상호 dependency를 표현할 수 있음.

|  |  |
| --- | --- |
|  | help:      @echo "try enter - make all"  hyundai.o: hyundai.c  # compile only (c -> asm)      gcc -S hyundai.c  # assembling only (asm -> obj)      as     hyundai.s -o hyundai.o  main.o: main.c  # preprocessing      cpp main.c > main.i  # compile only (c -> asm)      gcc -S main.i -o main.s  # assembling only (asm -> obj)      as     main.s -o main.o  main.exe: main.o hyundai.o  # linking (multi objs -> executable )      gcc main.o hyundai.o  -o main.exe  all: main.exe      ./main.exe  dump:      objdump -d -S main.exe > main\_dis.S  rebuild:      make clean      make all    clean:      del \*.o \*.s \*.exe |

* **(2) 디버거 활용한 온칩 메모리 분석**
  + **A. 디버거 메뉴**
    - resume, pause
    - step into, step over
    - 
  + **B. printf 이용하여 출력 확인**
    - 상단메뉴에서 window / show view, /other/Tasking/FSS 클릭.
    - 
  + **C. 메모리 inspection**
    - 0x700195FC 값을 확인.
    - 
* **(3) 변수를 온칩 메모리에 올려두고 SW실행.**
  + **A. 변수를 메모리에 할당, 그리고 할당주소 (요리 재료를 식탁위에 올리기).**
    - 모든 변수 및 하드웨어 장치는 메모리의 특정 영역에 배치되어 있다.
    - 특정 영역이라 함은 주소를 말한다.
    - 변수의 메모리 상에 배치된 주소를 얻고 싶으면 변수 이름 앞에 & 기호를 붙인다.

|  |  |
| --- | --- |
|  | **#include <stdio.h>**  **main() {**  **...**  **unsigned int k1 = 0x5A1234FF;**      // 변수는 메모리의 특정 주소에 배치됨      // 그 주소는 변수이름앞에 &기호를 붙이면 된다.  **printf("k1's address on memory : 0x%p\n", &k1);** |