일반

[CMake 튜토리얼] 2. CMakeLists.txt 주요 명령과 변수 정리

Posted **2017. 02. 26** *Updated* **2018. 05. 31** *Views* **29179** *Replies* **1**

▶ 이 글에서는 CMake 빌드 스크립트인 CMakeLists.txt 파일을 작성하는 방법에 대해 다룹니다. **CMake 2.8.x 버전 기준**이며, C언어 프로젝트를 기준으로 자주 사용되는 명령과 변수들을 선별하여 기능에 따라 구분하여 기술하였습니다. 여기서 다루지 않은 구문들은 다음 CMake 공식 문서를 참조해 주세요.

[CMake 2.8.12 Documentation]
https://cmake.org/cmake/help/v2.8.12/cmake.html

공식 매뉴얼에 들어가 보시면 알겠지만, 전체 매뉴얼이 한 개의 매우 긴 페이지로 되어 있고[...] 세부 옵션 설명을 난잡하게 문장으로 기술해 놓은 등 가독성이 참 떨어지게 만들어 놓았습니다. 3.x 버전 매뉴얼은 그나마 검색해서 항목별로 찾아볼 수 있게 해 놓았지만, 본문은 여전히 2.x 매뉴얼을 거의 그대로 복붙한 것이어서 역시 가독성이 좋

지 않습니다. CMake의 강력한 기능에 비해 매뉴얼이 부실한 점은 약간 아쉬운 대목입니다.

■ SET() - 변수 정의

CMakeLists.txt를 열었을 때 첫 화면에서, 또 가장 자주 쓰이는 명령이 바로 이 변수를 정의하는 SET() 명령이 아닐까 합니다.

CMake 빌드 스크립트를 작성할 때도 Makefile을 비롯한 여타 스크립트와 마찬가지로 상단에는 설정 변수를 정의하는 명령들을 몰아놓고, 하단에서는 이 설정에 따라 빌드 절차를 결정하도록 구성합니다. 이렇게 하면 빌드 환경이나 구성이 바뀌어서 빌드 스크립트를 수정해야 할 필요가 있을 때, 필요한 부분을 금방 찾아서 고칠 수 있습니다.

변수 정의

```
SET ( <변수명> <값> )
```

<값>에 공백이 포함되어 있는 경우, 큰따옴표 **"..."**로 둘러주면 됩니다.

목록(List) 변수 정의

```
SET ( <목록_변수명> <항목> <항목> <항목> ... )
```

<항목>들은 공백문자로 구분합니다. <항목>값에 공백이 포함되어 있는 경우, 역시 큰따옴표로 둘러주면 됩니다. 목록 변수의 항목들은 기본적으로 세미콜론(;)으로 구분되어 저장됩니다.

변수 참조

변수를 참조하고자 할 때는 다음과 같이 변수명 앞에 **$**를 붙이거나, **${...}**로 둘러 주면 됩니다.

```
$변수명
${<변수명>}
```

다음은 빌드 대상 소스 파일 목록을 SRC_FILES 변수로 지정하고, 이들로부터 app.out 실행파일을 생성하기 위한 스크립트입니다. (여기서 ADD_EXECUTABLE(...)은 빌드 바이너리를 정의하는 명령이며, 뒤에서 자세히 설명합니다.)

```
SET ( SRC_FILES main.c foo.c bar.c )
ADD_EXECUTABLE ( app.out ${SRC_FILES} )
```

목록 변수는 자신이 참조되는 위치에 따라 적절하게 자동으로 직렬화 (Serialization) 됩니다. (정확히는 명령을 처리하는 함수 내에서 상황에 맞도록 직렬화 하는 것입니다.)

위 예시에서 ${SRC_FILES}위치에는 본래 공백 문자로 구분된 소스파일명이 나열되어야 하지만, 이렇게 목록 변수를 입력하면 항목들이 공백문자로 구분되어 자동으로 직렬화 됩니다. 즉, 위 예시의 두 번째 줄은 명령을 실행할 때 다음과 같이 해석됩니다.

```
ADD_EXECUTABLE ( app.out main.c foo.c bar.c )
```

예약 변수

CMake에 내장되어 있는 예약 변수들은 전체 빌드 흐름을 좌지우지할 정도로 중요한 역할을 하지만, 사용자 정의 변수처럼 **SET()** 명령으로 값을 변경할 수 있습니다. 후술할 대다수의 명령들의 실체가 사실은 이 내장 변수들을 적절하게 조작하는 함수들입니다.

CMake에서 제공하는 명령들이 이 예약 변수들을 모두 커버하지는 않기 때문에 일부 예약 변수는 **SET()** 명령으로 직접 지정해 줘야 합니다. 혹은, 명령으로 설정한 변수들이 올바르게 설정되었는지 확인하는 등 빌드 스크립트 디버깅 목적으로 이들을 참조해야 하는 경우도 있습니다.

CMake 빌드 스크립트를 쉽게 작성하려면 예약 변수를 잘 알아야 합니다. 수많은 예약 변수들이 있지만, 이 글에서는 빌드를 제어하는 중요한 것들만 추려서 소개하도록 하겠습니다.

* 예약 변수를 한 데 모아서 설명하기 보다는, 주제별로 연관된 명령과 함께 소개하는 편이 좋을 것으로 판단하여 글을 재구성 하였습니다.

지금부터 설명하는 내용에서 녹색은 변수를, 파란색()은 명령을 나타냅니다.

(변수 뒤에 괄호 열고-닫고 해놓고 왜 안되징? 찡찡 하기 없기~^^)

■ 프로젝트 전반 관련

CMAKE_MINIMUM_REQUIRED() - 필요 CMake 최소 버전 명시

CMake 빌드 스크립트를 실행하기 위한 최소 버전을 명시합니다. 보통 CMakeLists.txt의 최상단에 위치하며, 여기에 명시한 버전보다 낮은 CMake가 해당 빌드 스크립트를 해석하려고 하면 오류를 출력하고 종료합니다.

```
CMAKE_MINIMUM_REQUIRED ( VERSION <버전> )
```

- **<버전>** : x.y.z.w 형식의 최소 요구 CMake 버전. Major version인 x는 반드시 명시되어야 하고, 나머지는 생략할 수 있습니다.

예) 다음 명령은 해당 빌드 스크립트를 해석하기 위해 요구되는 CMake의 최소 버전이 2.8임을 나타냅니다.

```
CMAKE_MINIMUM_REQUIRED ( VERSION 2.8 )
```

PROJECT() - 프로젝트 이름 설정

프로젝트 이름을 설정합니다. 프로젝트 이름에 공백이 포함되어 있는 경우 큰따옴표로 둘러 주면 되지만, 가급적이면 공백을 포함하지 않는 편이 좋습니다.

```
PROJECT ( <프로젝트명> )
```

CMAKE_PROJECT_NAME - 프로젝트 이름

PROJECT() 명령으로 설정한 프로젝트 이름이 이 변수에 저장됩니다.

예) 다음은 프로젝트 이름을 콘솔에 출력합니다.

```
MESSAGE ( ${CMAKE_PROJECT_NAME} )
```

CMAKE_BUILD_TYPE - 빌드 형상(Configuration)

CMake 빌드 시스템에서도 여타 IDE에서와 같이 빌드 형상을 지정할 수 있습니다. 빌드 형상이란 빌드 목적(디버깅용인지, 배포용인지, 샵질용인지)에 따라 서로 다른 옵션을 지정해서 빌드하는 것으로, 대표적으로 **Debug**와 **Release**가 있습니다.

CMake에서는 기본적으로 다음과 같이 네 가지 빌드 형상을 지원하며, 필요한 경우 사용자 정의 빌드 형상을 정의할 수도 있습니다.

- **Debug** : 디버깅 목적의 빌드
- **Release** : 배포 목적의 빌드
- **RelWithDebInfo** : 배포 목적의 빌드지만, 디버깅 정보 포함
- **MinSizeRel** : 최소 크기로 최적화한 배포 목적 빌드

이 변수를 지정하면 Makefile을 작성할 때 빌드 형상에 따라 서로 다른 빌드 옵션(플래그)을 삽입해 줍니다. 따라서 빌드 목적별로 빌드 스크립트를 각각 따로 만들어 줘야 할 수고를 하지 않아도 되므로 편리합니다.

MESSAGE() - 콘솔에 메시지 출력

콘솔에 메시지나 변수를 출력합니다. 빌드 스크립트 디버깅시 요긴하게 활용할 수 있습니다.

```
MESSAGE ( [<Type>] <메시지> )
```

<Type>은 다음중 하나이며, 생략할 수 있습니다.

- **STATUS** : 상태 메시지 출력 (메시지 앞에 '--'가 추가되서 출력됨)
- **WARNING** : 경고 메시지를 출력하고, 계속 진행
- **AUTHOR_WARNING** : 프로젝트 개발자용 경고 메시지를 출력하고, 계속 진행
- **SEND_ERROR** : 오류 메시지를 출력하고 계속 진행하지만, Makefile 생성은 하지 않음
- **FATAL_ERROR** : 오류 메시지를 출력하고, 작업을 즉시 중단

<Type>을 생략하면 중요한 정보임을 나타내며, 콘솔에 메시지를 출력합니다.
(당연한 소리지만) <메시지>에는 변수를 입력할 수 있습니다.

예) 다음 명령은 오류 메시지 "Fatal error occurred!" 를 출력하고 Makefile 작성을 즉시 중단합니다.

```
MESSAGE ( FATAL_ERROR "Fatal error occurred!" )
```

한 가지 유의할 사항은, 이 명령은 CMakeLists.txt로부터 Makefile을 생성하는 시점에 메시지를 출력한다는 점입니다. 즉, 이 명령으로 출력하는 메시지는 실제 빌드시에는 출력되지 않습니다. 빌드 중간에 메시지를 출력하고 싶다면, 뒤에서 설명할 **ADD_CUSTOM_COMMAND()** 명령의 COMMAND 옵션에 echo문을 추가하는 방식으로 해야 합니다.

CMAKE_VERBOSE_MAKEFILE - Verbose Makefile 작성 여부

이 변수는 Switch변수이며, 다음과 같이 값을 true(또는 1)로 지정하면 빌드 상세 과정을 모두 출력하는 Makefile을 생성합니다.

```
SET ( CMAKE_VERBOSE_MAKEFILE true )
```

빌드 과정에서 CMake가 실행하는 실제 빌드 명령을 모두 볼 수 있으므로 빌드 스크립트를 작성할 때 이 옵션을 켜 놓는 것이 좋습니다. 이 옵션을 끄고 생성한 Makefile은 빌드 과정에서 어떤 파일을 생성했는지 결과만 한줄씩 출력하기 때문에 실수로 플래그 등을 잘못 지정했더라도 확인할 길이 없습니다.

■ 빌드 대상(Target) 관련

빌드 대상(Target)이란 모든 빌드가 끝나고 최종적으로 출력되는 실행파일과 라이브러리를 의미합니다.

ADD_EXECUTABLE() - 빌드 대상 바이너리 추가

빌드 최종 결과물로 생성할 실행 파일을 추가합니다. 이 명령을 반복하여 생성할 실행 파일을 계속 추가할 수 있습니다.

빌드 스크립트에 이 한 줄만 달랑 써져 있어도 동작하므로, 이거 하나만 알고 있으면 Makefile을 만들기 귀찮을 때 요긴하게 써먹을 수 있습니다.

```
ADD_EXECUTABLE ( <실행_파일명> <소스_파일> <소스_파일> ... )
```

- **<실행_파일명>** : 생성할 바이너리의 파일명
- **<소스_파일>** : 실행 파일을 생성하는 데 필요한 소스 파일

예) 다음 명령은 {main.c, foo.c, bar.c} 소스 파일로부터 app.out 이라는 이름의 바이너리를 생성합니다.

```
ADD_EXECUTABLE ( app.out main.c foo.c bar.c )
```

ADD_LIBRARY() - 빌드 대상 라이브러리 추가

빌드 최종 결과물로 생성할 라이브러리를 추가합니다. 이 명령을 반복하여 생성할 라이브러리를 계속 추가할 수 있습니다.

```
ADD_LIBRARY ( <라이브러리_이름> [STATIC|SHARED|MODULE] <소스_파일> <소스_파일> ... )
```

- **<라이브러리_이름>** : 생성할 라이브러리 이름 (lib~.a / lib~.so 에서 ~에 들어갈 값)
- **[STATIC|SHARED|MODULE]** : 라이브러리 종류 (생략시 STATIC)
- **<소스_파일>** : 라이브러리를 생성하는 데 필요한 소스 파일

예) 다음 명령은 {foo.c, bar.c} 소스 파일로부터 libapp.a라는 이름의 라이브러리를 생성합니다.

```
ADD_LIBRARY ( app STATIC foo.c bar.c )
```

ADD_CUSTOM_TARGET() - 사용자 정의 Target 추가

통상적인 빌드 절차로 생성할 수 없는 Target을 추가합니다.. 라지만, 실상은 make 매크로를 정의할 때 더 많이 사용합니다.(make flash 라

던가.) Taget의 Recipe를 직접 지정해야 하는 경우 유용하게 활용할 수 있습니다.

이 명령으로 정의한 Target은 출력 파일을 생성하지 않아도 무방하며, 항상 'Outdated'로 간주되므로 매 빌드마다 COMMAND 루틴이 실행됩니다. 의존성에 따라 실행 여부를 선택하도록 하려면 ADD_CUSTOM_COMMAND() 명령을 사용해야 합니다.

```
ADD_CUSTOM_TARGET (
                <이름> [ALL]
                [COMMENT <출력_메시지>]
                [DEPENDS <의존_대상_목록>]
                [WORKING_DIRECTORY <작업_디렉토리>]
                COMMAND <명령>
                [COMMAND <명령>]
                ...
)
```

- **<이름>** : Target 이름
- **[ALL]** : make(또는 make all) 명령에서 기본 빌드 대상에 포함할지 여부
- **<출력_메시지>**: 명령 실행 전에 콘솔에 출력할 메시지
- **<의존_대상_목록>**: 이 Target이 의존하는 대상 목록 (공백으로 구분)
- **<작업_디렉토리>**: 명령을 실행할 위치(pwd)
- **<명령>** : Target을 생성하기 위한 명령(Recipe)

예) 다음 구문은 ESP8266 프로젝트에서 생성한 바이너리(app.bin)를 Flashing하기 위한 make flash 매크로를 정의합니다.

```
ADD_CUSTOM_TARGET ( flash
        COMMENT "Flashing binary"
        WORKING_DIRECTORY ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}
        COMMAND python esptool.py write_flash app.bin
        DEPENDS app.out
        )
```

ADD_DEPENDENCIES() - Target간 의존성 정의

Top-level Target간의 의존성을 지정합니다. Top-level Target이란 ADD_EXECUTABLE, ADD_LIBRARY, ADD_CUSTOM_TARGET 명령으로 정의한 Target들을 의미합니다.

Target을 빌드할 때 이 명령으로 정의한 의존 대상들이 'Outdated'인 경우 이들에 대한 빌드를 먼저 수행합니다.

```
ADD_DEPENDENCIES ( <Target_이름> <의존_대상> <의존_대상> ...
)
```

- **<Target_이름>** : Target 이름
- **<의존_대상>** : 이 Target이 의존하는 대상

예) 다음 명령은 flash(매크로)가 app.out에 의존적임을 명시합니다. 이렇게 작성하고 'make flash'를 실행하면 app.out이 최신인지 여부를 먼저 검사해서 필요시 app.out을 먼저 빌드하고 flash 매크로를 실행합니다.

```
ADD_DEPENDENCIES ( flash app.out )
```

※ 여기서 짚고 넘어가야 할 점이 한가지 있는데, 이 명령은 CMake 내부적으로 파악이 불가능한 Target 사이의 의존성을 명시할 때 사용한다는 점입니다. Target과 소스 파일간 의존성은 ADD_EXECUTABLE이나 ADD_LIBRARY 구문에서 묵시적으로 지정되므로 이 명령으로 명시할 필요가 없습니다.

INSTALL() - 설치 매크로(make install) 정의

Makefile에서 관용적으로 설치용 Target으로 사용되는 install target의 동작 방식을 정의합니다. 즉, **'make install'** 명령을 실행했을 때 무슨 동작을 수행할지를 지정합니다.

Makefile 튜토리얼에서 언급했듯이 사실 리눅스 시스템에서 설치(install)란 별 게 아니고, 빌드 완료된 실행 바이너리와 라이브러리 및 기타 부속물(헤더 파일, 리소스 등)들을 시스템의 적절한 위치로 복사하는 동작입니다.

CMake 빌드스크립트에서 INSTALL() 명령을 이용하면 설치 매크로를 쉽게 작성할 수 있습니다.(이게 별로 마음에 안 들면 ADD_CUSTOM_TARGET(...)명령으로 설치 매크로를 직접 작성하면 됩니다.)

* 이 명령은 상당히 많은 옵션을 제외하고 설명하였습니다. 권한(Permission)을 지정하거나, 파일이나 디렉토리를 복사하는 등의 여러 동작을 수행하는 옵션을 제공합니다. 자세한 내용은 공식 매뉴얼 을 참조해 주세요.

```
INSTALL ( TARGETS <Target_목록>
        RUNTIME DESTINATION <바이너리_설치_경로>
        LIBRARY DESTINATION <라이브러리_설치_경로>
        ARCHIVE DESTINATION <아카이브_설치_경로>
        )
```

설치 경로가 상대 경로인 경우 **CMAKE_INSTALL_PREFIX** 변수에 지정한 경로 아래에 설치됩니다.

설치 경로가 모두 같은 경우, 다음과 같이 축약된 형태로도 사용할 수 있습니다.

```
INSTALL ( TARGETS <Target_목록> DESTINATION <설치_경로> )
```

예) 다음 명령은 app.out은 /usr/local/bin에, libapp.a는 /usr/local/lib에 설치하는 install target을 추가합니다.

```
INSTALL ( TARGETS app.out app
        RUNTIME_DESTINATION /usr/local/bin
        ARCHIVE_DESTINATION /usr/local/lib
        )
```

CMAKE_INSTALL_PREFIX - 설치 디렉토리

설치 매크로(make install)에서 실행 바이너리와 라이브러리 등의 최종 생성물을 복사할 설치 디렉토리를 지정합니다. INSTALL() 명령에서 상대 경로를 사용한 경우, 이 변수에 지정한 디렉토리가 Base 디렉토리가 됩니다.

예) 다음은 설치 기본 경로를 /usr/bin 으로 지정합니다.

```
SET ( CMAKE_INSTALL_PREFIX /usr/bin )
```

* 이 변수를 별도로 지정하지 않으면 기본값은 /usr/local 입니다.

■ 전역 빌드 설정 관련

다음은 모든 Target에 공통으로 적용되는 빌드 옵션을 지정하는 변수와 명령들입니다.

한 가지 유의할 사항은, 이 전역 빌드 설정 **명령**들은 선언 **이후**에 정의되는 Target들에게만 적용된다는 점입니다. 따라서 여기서 소개하는 명령들은 대상 Target들이 정의되기 전인 CMakeLists.txt의 상단에 위치시켜야 합니다.

CMAKE_C_COMPILER - C 컴파일러

컴파일 및 링크 과정에서 사용할 컴파일러의 경로를 지정합니다.

ADD_COMPILE_OPTIONS() - 컴파일 옵션 추가

소스 파일을 컴파일하여 Object 파일을 생성할 때 컴파일러에 전달할 옵션(플래그)을 추가합니다.

```
ADD_COMPILE_OPTIONS ( <옵션> <옵션> ... )
```

예) 다음 명령은 컴파일시 디버깅 목적의 심벌 테이블을 포함(-g)하고, 모든 경고 메시지를 표시(-Wall)하도록 합니다.

```
ADD_COMPILE_OPTIONS( -g -Wall )
```

CMAKE_C_FLAGS_<빌드_형상> - 빌드 형상별 컴파일 옵션

특정 빌드 형상에서만 사용할 컴파일 옵션(플래그)를 지정합니다.

예) 다음은 Release 빌드시에만 CONFIG_RELEASE 매크로를 정의하고 3단계 최적화를 수행하도록 컴파일 옵션을 지정합니다.

```
SET ( CMAKE_C_FLAGS_RELEASE "-DCONFIG_RELEASE -O3" )
```

※ 주의: 옵션이 여러개인 경우 목록으로 정의하지 말고, 위의 예시와 같이 쌍따옴표 "..." 로 둘러서 하나의 문자열로 정의해야 합니다. 그렇지 않으면 각 옵션 항목들이 공백 문자가 아닌 세미콜론으로 구분되어 컴파일 명령에 입력되므로 오류가 발생합니다.

ADD_DEFINITIONS() - 전처리기 매크로 추가 (-D)

전처리기에 전달할 매크로를 정의합니다. 컴파일러 옵션중 **-D**에 해당합니다.

```
ADD_DEFINITIONS ( -D<매크로> -D<매크로> -D<매크로>=값 ... )
```

예) 다음 명령은 ICACHE_FLASH 매크로 변수를 정의하고, MY_DEBUG 라는 이름의 전처리 매크로값을 1로 정의합니다.

```
ADD_DEFINITIONS( -DICACHE_FLASH -DMY_DEBUG=1 )
```

INCLUDE_DIRECTORIES() - 헤더 디렉토리 추가 (-I)

각 소스 파일에서 #include 구문으로 포함시킨 헤더 파일을 찾을 디렉토리 목록을 추가합니다. 컴파일러 옵션중 -I에 해당합니다.

```
INCLUDE_DIRECTORIES ( <디렉토리> <디렉토리> ... )
```

예) 다음 명령은 include 디렉토리와 driver/include 디렉토리에서 헤더 파일을 찾도록 합니다.

```
INCLUDE_DIRECTORIES ( include driver/include )
```

LINK_DIRECTORIES() - 라이브러리 디렉토리 지정 (-L)

링크 과정에서 필요한 라이브러리 파일들을 찾을 디렉토리 목록을 지정합니다. 컴파일러 옵션중 -L에 해당합니다.

```
LINK_DIRECTORIES ( <디렉토리> <디렉토리> ... )
```

예) 다음 명령은 링크시 lib 및 /var/lib 디렉토리에서 라이브러리 파일을 찾도록 합니다.

```
LINK_DIRECTORIES ( lib /var/lib )
```

LINK_LIBRARIES() - 링크 옵션 추가

링크시 포함할 라이브러리 목록을 지정합니다. 이 때, 라이브러리 파일명의 Prefix 및 Postfix는 제외하고 라이브러리 이름만 입력합니다. (e.g. libxxx.a에서 xxx에 해당하는 부분만 입력) 컴파일러 옵션중 -l에 해당합니다.

```
LINK_LIBRARIES ( <라이브러리> <라이브러리> ... )
```

* 이 명령으로 링크 옵션도 함께 지정할 수 있습니다. <라이브러리> 값이 하이픈('-')으로 시작하는 경우 링크 명령에 그대로 포함되고, 그렇지 않은 경우 앞에 '-l'이 자동으로 추가됩니다.

예) 다음 명령은 링크 라이브러리로 libuart.a(또는 libuart.so)와 libwifi.a(또는 libwifi.so)를 추가하고, Shared 라이브러리를 제외하는 옵션(-static)을 추가합니다.

```
LINK_LIBRARIES ( uart wifi -static )
```

CMAKE_EXE_LINKER_FLAGS_<빌드_형상> - 빌드 형상별 링크 옵션

특정 빌드 형상에서만 사용할 링크 옵션(플래그)를 지정합니다.

예) 다음은 Debug 빌드시에만 CONFIG_DEBUG 매크로를 정의하고 모든 Symbol을 포함하도록 링크 옵션을 지정합니다.

```
SET ( CMAKE_EXE_LINKER_FLAGS_DEBUG "-DCONFIG_DEBUG -Wl,-whole-archive" )
```

※ 역시 마찬가지로, 옵션이 여러개인 경우 쌍따옴표로 둘러 줘야 합니다.

RUNTIME_OUTPUT_DIRECTORY - 실행 바이너리 출력 디렉토리

빌드 완료한 실행 바이너리를 저장할 디렉토리를 지정합니다.

예) 다음 구문은 빌드한 실행 바이너리를 프로젝트 디렉토리 내의 output/bin 에 저장하도록 합니다.

```
SET ( RUNTIME_OUTPUT_DIRECTORY output/bin )
```

LIBRARY_OUTPUT_DIRECTORY - 라이브러리 출력 디렉토리

빌드 완료한 라이브러리를 저장할 디렉토리를 지정합니다.

예) 다음 구문은 빌드한 라이브러리를 프로젝트 디렉토리 내의 output/lib 에 저장하도록 합니다.

```
SET ( LIBRARY_OUTPUT_DIRECTORY output/lib )
```

ARCHIVE_OUTPUT_DIRECTORY - 아카이브 출력 디렉토리

빌드 완료한 아카이브(Static 라이브러리)를 저장할 디렉토리를 지정합니다.

예) 다음 구문은 빌드한 아카이브를 프로젝트 디렉토리 내의 output/lib/static 에 저장하도록 합니다.

```
SET ( ARCHIVE_OUTPUT_DIRECTORY output/lib/static )
```

■ 특정 Target 한정 빌드 설정 관련

다음은 특정 Target에 한정해서 빌드 옵션을 지정하는 명령들입니다. 최종 생성 Target이 여러개이고, Target마다 빌드 옵션을 서로 다르게 지정해야 할 필요가 있을 경우 사용합니다.

이 단락에서 소개하는 명령들은 모두 **'TARGET_'으로 시작**하고, **첫 번째 인수는 Target 이름**입니다. 이 명령들을 선언하기 전에 대상 Target은 반드시 미리 선언되어 있어야 합니다.

TARGET_COMPILE_OPTIONS() - Target 컴파일 옵션 추가

Target의 소스 파일을 컴파일할 때 전달할 옵션(플래그)을 추가합니다.

```
TARGET_COMPILE_OPTIONS ( <Target_이름> PUBLIC <옵션> <옵션>
... )
```

- **<Target_이름>** : Target 이름

- **PUBLIC** : 전역 컴파일 옵션 변수(COMPILE_OPTIONS)와 인터페이스 컴파일 옵션 변수(INTERFACE_COMPILE_OPTIONS)를 확장합니다. 보통의 경우 PUBLIC으로 두면 됩니다.
  (그 외 가능한 값으로 INTERFACE와 PRIVATE이 있으며, 자세한 설명은 공식 매뉴얼 을 참조하세요.)
- **<옵션>** : 컴파일 옵션

예) 다음 명령은 app.out을 컴파일할 때 디버깅 목적의 심벌 테이블을 포함(-g)하고, 모든 경고 메시지를 표시(-Wall)하도록 합니다.

```
TARGET_COMPILE_OPTIONS ( app.out PUBLIC -g -Wall )
```

TARGET_COMPILE_DEFINITIONS() - Target 전처리기 매크로 정의 (-D)

Target의 소스 파일을 컴파일하여 Object 파일을 생성할 때 전처리기에 전달할 매크로를 정의합니다. 컴파일러 옵션중 **-D**에 해당합니다.

```
TARGET_COMPILE_DEFINITIONS ( <Target_이름> PUBLIC <매크로> <매크로> <매크로>=<값> ... )
```

* ADD_DEFINITIONS()와는 달리, <매크로>를 지정할 때 선행 -D는 생략 가능합니다. 즉, <매크로>가 '-D'으로 시작하는 경우 컴파일 명령에 그대로 포함되고, 그렇지 않은 경우 -D가 자동으로 추가됩니다.

예) 다음 명령은 app.out을 컴파일할 때 UART_BUFFERED, ICACHE 매크로 변수를 정의하고, DEBUG라는 이름의 전처리 매크로값을 1로 정의합니다.

```
TARGET_COMPILE_DEFINITIONS ( app.out PUBLIC UART_BUFFERED -DICACHE DEBUG=1 )
```

TARGET_INCLUDE_DIRECTORIES() - Target 헤더 디렉토리 추가 (-I)

Target에 포함된 소스 파일에서 #include 구문으로 포함시킨 헤더 파일을 찾을 디렉토리 목록을 추가합니다. 컴파일러 옵션중 **-I**에 해당합니다.

```
TARGET_INCLUDE_DIRECTORIES ( <Target_이름> PUBLIC <디렉토리> <디렉토리> ... )
```

예) 다음 명령은 app.out을 빌드할 때 include 디렉토리와 driver/include 디렉토리에서 헤더 파일을 찾도록 합니다.

```
TARGET_INCLUDE_DIRECTORIES ( include driver/include )
```

TARGET_LINK_LIBRARIES() - Target 링크 옵션 및 라이브러리 지정 (-l)

Target 링크시 포함할 라이브러리 목록을 지정합니다. 이 때, 라이브러리 파일명의 Prefix 및 Postfix는 제외하고 라이브러리 이름만 입력합니다. (e.g. libxxx.a에서 xxx에 해당하는 부분만 입력) 컴파일러 옵션중 **-l**에 해당합니다.

```
TARGET_LINK_LIBRARIES ( <Target_이름> <라이브러리> <라이브러리> ... )
```

* 이 명령으로 링크 옵션도 함께 지정할 수 있습니다. <라이브러리> 값이 하이픈('-')으로 시작하는 경우 링크 명령에 그대로 포함되고, 그렇지 않은 경우 앞에 '-l'이 자동으로 추가됩니다.

예) 다음 명령은 app.out을 빌드할 때 libuart.a(또는 libuart.so)와 libwifi.a(또는 libwifi.so)를 포함하고, Shared 라이브러리를 제외하는 옵션(-static)을 지정합니다.

```
TARGET_LINK_LIBRARIES ( app.out uart wifi -static )
```

■ 빌드 절차(Build Step) 관련

CONFIGURE_FILE() - 템플릿 파일로부터 파일 자동 생성

빌드 시작 직전에 템플릿 파일 내용 중 빌드 스크립트에 정의된 변수를 치환해서 출력 파일로 작성합니다. 컴파일러를 실행하기 전에 수행하는 '전전처리' 과정이라 할 수 있습니다.

```
CONFIGURE_FILE ( <템플릿_파일명> <출력_파일명> )
```

템플릿 파일 내용 중 **${<변수명>}** 또는 **@<변수명>@**이 모두 변수 값으로 치환되어 출력 파일로 저장됩니다.

이 명령은 주로 프로그램의 버전을 명시하는 헤더 파일을 빌드 직전에 자동 생성해야 하는 경우 사용합니다. 이렇게 하면 프로그램 내에 버전을 명시할 때 헤더 파일을 수정하는 대신 빌드 스크립트에서 일괄 관리할 수 있으므로 편리합니다.

예) 다음은 version.h.in을 읽어들여서 버전 정보(1.0.9.7)를 입력하고 version.h로 저장하기 위한 구문입니다.

CMakeLists.txt

```
SET ( PROJECT_VERSION_MAJOR 1 )
SET ( PROJECT_VERSION_MINOR 0 )
SET ( PROJECT_VERSION_PATCH 9 )
SET ( PROJECT_VERSION_TWEAK 7 )
CONFIGURE_FILE ( version.h.in version.h )
```

템플릿 파일인 version.h.in의 내용이 다음과 같다면,

version.h.in

```
#define VERSION "${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_VERSION_PATCH}.${PROJECT_VERSION_TWEAK}"
```

출력 파일인 version.h이 빌드 직전에 다음과 같이 자동 생성됩니다.

version.h

```
#define VERSION "1.0.9.7"
```

ADD_CUSTOM_COMMAND(OUTPUT) - 사용자 정의 출력 파일 추가

통상적인 빌드 절차로 생성할 수 없는 출력 파일을 추가합니다. 출력 파일의 Recipe를 직접 지정해야 하는 경우 유용하게 활용할 수 있습니다.

사용자 정의 Target을 추가하는 명령인 **ADD_CUSTOM_TARGET()** 명령과의 차이점은 다음과 같습니다.

- 생성하는 출력 파일들이 최신인지 여부를 검사해서 명령(Recipe)을 실행할지 여부를 결정합니다. 따라서, 사용자 정의 출력 파일을 생성하는 데 많은 시간이 걸리는 경우 사용하면 유용합니다.

- 명령(Recipe)을 실행했을 때 최소 한 개 이상의 출력 파일이 있어야 하며, Outdated 판정을 위해 출력 파일을 OUTPUT 인수로 명시해야 합니다.

```
ADD_CUSTOM_COMMAND (
                OUTPUT <출력_파일_목록>
                [COMMENT <출력_메시지>]
                [DEPENDS <의존_대상_목록>]
                [WORKING_DIRECTORY <작업_디렉토리>]
                COMMAND <명령>
                [COMMAND <명령>]
                ...
)
```

예) 다음은 app.out으로부터 임베디드 프로세서에 Flashing하기 위한 app.bin을 생성하는 명령입니다.

```
ADD_CUSTOM_COMMAND (
                OUTPUT app.bin
                COMMENT "Generating binary from executable"
                DEPENDS app.out
                COMMAND python tools/elf2bin.py app.out ap
p.bin
)
```

ADD_CUSTOM_COMMAND(TARGET) - 빌드 과정 명령 추가

특정 Target의 빌드 전(Pre-build)/중(Pre-link)/후(Post-build)에 수행할 명령을 추가합니다.

ELF파일로부터 임베디드 프로세서에 Flashing하기 위한 BIN파일을 생성하는 것과 같이 통상적인 빌드 과정에서 수행되는 명령으로는 처리할 수 없는 동작을 빌드 대상물에 수행해 줘야 할 필요가 있을 경우 사용합니다.

```
ADD_CUSTOM_COMMAND (
                TARGET <대상_Target_이름>
                <PRE_BUILD|PRE_LINK|POST_BUILD>
                [COMMENT <출력_메시지>]
                [WORKING_DIRECTORY <작업_디렉토리>]
                COMMAND <명령>
                [COMMAND <명령>]
                ...
)
```

- **<대상_Target_이름>** : 여기 지정한 Target의 빌드 과정을 대상으로 합니다.
- **<PRE_BUILD|PRE_LINK|POST_BUILD>** : 명령 실행 시점

예) 다음은 app.out을 빌드 완료한 뒤, 이로부터 임베디드 프로세서에 Flashing하기 위한 app.bin을 생성하는 명령입니다.

```
ADD_CUSTOM_COMMAND (
                TARGET app.out
                POST_BUILD
                COMMENT "Generating binary from executable"
                COMMAND python tools/elf2bin.py app.out ap
p.bin
)
```

* ADD_CUSTOM_COMMAND(OUTPUT)에서 제시한 예시에서는 OUTPUT에 지정한 파일들이 Outdated인 경우에만 실행되지만, 여기에서는 출력물의 최신 여부에 관계 없이 TARGET에 지정한 **대상 Target이 빌드되는 경우**에만 실행됩니다.

이 글에서는 CMake의 빌드 스크립트인 CMakeLists.txt를 작성하기 위한 주요 문법과 내장 변수들에 대해 다뤘습니다. 다음 글에서는 통상적으로 많이 사용되는 CMakeLists.txt 패턴을 제시하면서 튜토리얼을 마치도록 하겠습니다.

연관글

[CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴 (11278) *4
[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리 (30066)
[Make 튜토리얼] Makefile 예제와 작성 방법 및 기본 패턴 (29680) *2

TAG • Make, Makefile, CMake, CMakeLists.txt

‹ **Prev** [CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴

[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리 **Next** ›

Comment **'1'**

정광호 정광호 2018.05.31 05:09:28

안녕하세요. CMake 관련하여 많은 도움이 되었습니다. 감사합니다.

댓글 **0**

번호	분류	제목	글쓴이	최근 수정일	조회 수
191	일반	[WSL] Windows Subsystem for Linux - 디스플레이 서버 설정 및 GUI 사용하기	TUW	2019.01.15	2348
190	일반	[TeraTerm] 명령줄 인수와 공개키 인증으로 간편하게 SSH 접속하기	TUW	2018.11.06	770
189	일반	[WSL] Windows Subsystem for Linux - SSH 서버 자동 시작 설정하기	TUW	2018.11.06	1563
188	일반	[WSL] Windows Subsystem for Linux - SSH 서버 세팅하기	TUW	2018.11.09	2239
187	일반	[WSL] Windows Subsystem for Linux - Bash.exe를 Ubuntu와 유사하게 설정하기	TUW	2018.11.06	1369
186	일반	[WSL] Windows Subsystem for Linux - 초기 설치와 Ubuntu 배포판 설치	TUW	2018.11.06	2345
185	일반	[AutoHotkey] 단축키(Hotkey) 스크립트 작성과 자동 시작 등록	TUW	2018.11.08	1524
184	일반	[AutoHotkey] 소개와 설치 및 기본 설정 - GUI 예시, 기본 에디터 변경	TUW	2018.11.11	1355
183	일반	Windows에서 포트 포워딩(Port Forwarding) 설정하기 - Netsh	TUW	2018.02.03	8105
182	Security	[SSL/HTTPS] Let's Encrypt 무료 SSL 인증서 발급/설치/관리 - certbot 사용법	TUW	2019.02.25	11191
181	Security	[SSL/HTTPS] StartSSL/StartCom 사태와 Let's Encrypt로의 이전 **1**	TUW	2018.05.02	3414
180	Linux	[Ubuntu] Windows와 멀티부팅 환경에서 시간이 맞지 않는 현상 해결하기	TUW	2017.06.08	7016
179	일반	[Windows] 다중 NIC(LAN카드) 환경에서 Routing Table 설정 - route 명령	TUW	2018.06.13	15883
178	일반	[CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴 **4**	TUW	2019.02.11	11278
»	일반	[CMake 튜토리얼] 2. CMakeLists.txt 주요 명령과 변수 정리 **1**	TUW	2018.05.31	29179
176	일반	[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리	TUW	2018.06.13	30066

목록 검색

‹ Prev 1 2 3 4 5 6 7 ... 12 Next ›