PYNQ 3.1 원격 C++ 개발을 위한 최소 SD 카드 이미지 생성 가이드

서론

PYNQ(Python Productivity for Zynq)는 AMD의 적응형 컴퓨팅 플랫폼을 위한 생산성 높은 Python 기반 프레임워크로, Jupyter Notebook을 통해 하드웨어 가속을 손쉽게 활용할 수 있도록 지원합니다.[1] 그러나 프로토타이핑 단계를 넘어 실제 임베디드 시스템 배포나 고성능 애플리케이션 개발 단계에서는 Jupyter 환경의 오버헤드를 제거하고, C++와 같은 고성능 언어를 직접 활용하고자 하는 요구가 발생합니다. 특히, 호스트 PC에서는 Python의 편리함을 유지하면서 원격으로 타겟 보드의 C++ 애플리케이션을 제어하는 아키텍처는 효율성과 개발 생산성을 동시에 만족시키는 이상적인 모델입니다.

본 보고서는 PYNQ 버전 3.1을 기준으로, 이러한 고급 개발 요구사항을 충족시키기 위한 맞춤형 SD 카드 이미지를 생성하는 전 과정을 상세히 기술합니다. 목표는 표준 PYNQ 이미지에서 Jupyter Notebook과 전체 Python 생태계를 제거하고, 오직 C++ 기반 하드웨어 제어를 위한 PYNQ.cpp 라이브러리와 원격 프로시저 호출(RPC)을 위한 gRPC 프레임워크만을 탑재한 최소한의(minimal) 이미지를 구축하는 것입니다. 호스트 PC에서는 PYNQ.remote 라이브러리를 사용하여 이 타겟 보드에 원격으로 연결하고 제어하게 됩니다.

이 문서는 PYNQ의 sdbuild 빌드 시스템을 심층적으로 분석하고, 내부 패키지 구조를 수정하며, 나아가 PetaLinux/Yocto 레시피를 직접 작성하여 외부 라이브러리를 통합하는 고급 커스터마이제이션 기법을 포함합니다. 각 단계는 구체적인 명령어, 설정 파일 수정 내용, 그리고 상세한 설명을 포함하여 사용자가 그대로 따라 할 수 있는 "따라하기(follow-along)" 수준으로 제공됩니다. 본 가이드를 통해 개발자는 PYNQ 프레임워크를 단순한 프로토타이핑 도구를 넘어, 특정 요구사항에 최적화된 고성능 임베디드 시스템 개발 플랫폼으로 확장할 수 있는 전문적인 기술을 습득하게 될 것입니다.

섹션 1: PYNQ 3.1 빌드 환경 구축

성공적인 PYNQ 이미지 빌드의 첫걸음은 정확하고 일관된 빌드 환경을 구축하는 것입니다.

PYNQ의 각 릴리스는 특정 버전의 AMD 개발 도구 및 운영체제와 긴밀하게 연결되어 있으며, 버전 불일치는 빌드 실패의 가장 흔한 원인 중 하나입니다.[2] 이 섹션에서는 PYNQ 3.1 이미지 빌드에 필요한 모든 시스템 요구사항을 명시하고, 재현 가능한 빌드를 보장하기 위해 공식적으로 권장되는 Docker 기반 워크플로우를 단계별로 상세히 안내합니다.

1.1. 시스템 요구사항 및 툴체인 검증

PYNQ 3.1 이미지 빌드를 시작하기 전에, 다음의 소프트웨어 스택이 정확한 버전으로 설치되어 있는지 반드시 확인해야 합니다. 이 사전 검증 단계는 수 시간에 걸친 빌드 과정에서 발생할 수 있는 잠재적 오류를 미연에 방지하는 데 매우 중요합니다.

- 호스트 운영체제**:** Ubuntu 22.04 LTS (Jammy Jellyfish)가 공식적으로 지원되는 운영체제입니다.[5] 다른 버전의 Linux 배포판에서도 빌드가 가능할 수 있으나, 예기치 않은 종속성 문제가 발생할 수 있으므로 권장되지 않습니다.
- **AMD** 툴체인**:** PYNQ 3.1은 AMD의 2024.1 버전 툴체인과 완벽하게 동기화되어 있습니다. 따라서 Vivado, Vitis, PetaLinux는 모두 **2024.1** 버전으로 통일해야 합니다.[6] AMD 공식 다운로드 페이지에서 해당 버전을 설치합니다.
- 시스템 리소스**:** PYNQ 빌드 프로세스는 상당한 시스템 리소스를 요구합니다. 커뮤니티의 경험에 따르면, 원활한 빌드를 위해 최소 200GB 이상의 여유 디스크 공간, 4개 이상의 CPU 코어, 그리고 8GB 이상의 RAM을 확보하는 것이 좋습니다.[8]
- **Docker Engine:** PYNQ 3.1부터는 Docker를 사용한 빌드가 강력히 권장됩니다. 최신 버전의 Docker Engine을 설치해야 합니다.

다음 표는 PYNQ 3.1 빌드 환경의 핵심 요구사항을 요약한 것입니다. 빌드를 시작하기 전, 이 체크리스트를 통해 자신의 환경을 점검하십시오.

표 **1: PYNQ 3.1** 빌드 환경 요구사항

구성 요소	요구 버전	소스/근거	비고
호스트 운영체제	Ubuntu 22.04 LTS	[5]	Jammy Jellyfish 코드네임
AMD Vivado/Vitis	2024.1	[5]	PetaLinux와 버전 일치 필수
AMD PetaLinux	2024.1	[5]	Vivado/Vitis와 버전

			일치 필수
PYNQ Git 저장소	v3.1 태그	[6]	git clone --recursive 옵션 사용
Docker Engine	최신 안정 버전	[5]	권장 빌드 방식

1.2. 권장 **Docker** 기반 워크플로우

과거 PYNQ 버전에서는 Vagrant나 네이티브 Ubuntu 환경 설정의 복잡성으로 인해 많은 사용자들이 빌드 초기 단계에서 어려움을 겪었습니다.[8] 이러한 문제를 해결하기 위해 PYNQ 3.1 릴리스부터는 모든 종속성과 환경 구성을 격리된 컨테이너 내에서 관리하는 Docker 기반 워크플로우가 공식적으로 권장됩니다.[6] 이 방식은 호스트 시스템의 오염을 방지하고 빌드의 일관성과 재현성을 획기적으로 높여줍니다. 이는 PYNQ 프로젝트가 개발 환경의 안정성을 주요 개선 과제로 인식하고 있음을 보여주며, 전문가 수준의 개발에서는 이 방식을 따르는 것이 표준으로 간주됩니다.

1.2.1. 1단계**: Docker** 설치

호스트 시스템에 Docker가 설치되어 있지 않다면, Docker 공식 웹사이트의 설치 가이드에 따라 설치를 진행합니다.

1.2.2. 2단계**:** 호스트 시스템에 **AMD** 도구 설치

여기서 중요한 점은 수십 GB에 달하는 Vivado, Vitis, PetaLinux와 같은 AMD 도구들은 Docker 컨테이너 내부가 아닌, 호스트 시스템에 설치해야 한다는 것입니다.[5] Docker 컨테이너는 빌드 시점에 호스트에 설치된 이 도구들의 디렉토리를 마운트하여 사용하게 됩니다. 예를 들어,

/tools/Xilinx와 같은 경로에 설치합니다.

1.2.3. 3단계: PYNQ 저장소 복제

터미널을 열고 다음 명령어를 실행하여 PYNQ v3.1 저장소를 복제합니다. PYNQ는 여러 하위 모듈을 포함하고 있으므로 --recursive 플래그를 반드시 사용해야 합니다.

Bash

```
git clone --recursive https://github.com/Xilinx/PYNQ.git
```

1.2.4. 4단계: pynqdock Docker 이미지 빌드

복제된 PYNQ 디렉토리 내의 sdbuild 폴더로 이동한 후, 다음 명령어를 실행하여 PYNQ 빌드 전용 Docker 이미지를 생성합니다.

Bash

```
cd PYNQ/sdbuild
docker build \
 --build-arg USERNAME=$(whoami) \
 --build-arg USER_UID=$(id -u) \
 --build-arg USER_GID=$(id -g) \
 -t pynqdock:latest.
```

여기서 --build-arg로 전달되는 USERNAME, USER_UID, USER_GID는 매우 중요합니다. 이 인자들은 컨테이너 내부에서 생성되는 파일들의 소유자를 호스트 시스템의 현재 사용자와 일치시켜, 빌드 완료 후 결과물에 대한 파일 권한 문제를 방지하는 역할을 합니다.[5]

1.2.5. 5단계: 빌드 컨테이너 실행

이제 빌드된 pynqdock 이미지를 사용하여 빌드 작업을 수행할 컨테이너를 실행합니다. 아래 명령어에서 /tools/Xilinx와 /home/user/petalinux 경로는 실제 AMD 도구가 설치된 호스트 시스템의 경로로 수정해야 합니다.

Bash

```
docker run \
 --init \
 --rm \
 -it \
 -v /tools/Xilinx:/tools/Xilinx:ro \
 -v /home/user/petalinux:/home/user/petalinux:ro \
 -v $(pwd):/workspace \
 --name pynq-sdbuild-env \
 --privileged \
 pynqdock:latest \
 /bin/bash
```

위 명령어의 각 옵션은 다음과 같은 의미를 가집니다:

- --init: 컨테이너 내에서 좀비 프로세스가 발생하는 것을 방지합니다.
- --rm: 컨테이너가 종료될 때 자동으로 삭제되도록 합니다.
- -it: 상호작용 가능한 터미널(tty)을 할당합니다.
- -v /tools/Xilinx:/tools/Xilinx:ro: 호스트의 Xilinx 도구 디렉토리를 컨테이너의 동일한 경로에 읽기 전용(:ro)으로 마운트합니다.
- -v $(pwd):/workspace: 현재 sdbuild 디렉토리를 컨테이너의 /workspace에 마운트하여, 컨테이너 내에서 소스 코드를 수정하고 빌드 결과물을 호스트에서 바로 확인할 수 있도록 합니다.
- --name: 컨테이너에 식별 가능한 이름을 부여합니다.
- --privileged: 빌드 과정 중 루프백 디바이스 마운트와 같은 일부 작업에 필요한 특수 권한을 컨테이너에 부여합니다.[5]

이 명령어를 실행하면, 모든 빌드 준비가 완료된 Docker 컨테이너의 셸 프롬프트가 나타납니다.

1.3. 컨테이너 내부 환경 설정

Docker 컨테이너에 진입한 후에는, 현재 세션에서 AMD 도구를 사용할 수 있도록 환경 변수를

설정해야 합니다. 다음 source 명령어를 실행하여 Vitis와 PetaLinux의 설정 스크립트를 로드합니다. 경로는 컨테이너에 마운트된 경로와 일치해야 합니다.

Bash

```
source /tools/Xilinx/Vitis/2024.1/settings64.sh
source /tools/Xilinx/PetaLinux/2024.1/settings.sh
```

이 단계까지 완료하면, PYNQ 3.1 이미지를 빌드하기 위한 모든 사전 준비가 끝난 것입니다. 이제부터 모든 빌드 관련 명령어는 이 Docker 컨테이너 내부에서 실행하게 됩니다.

섹션 **2:** 기준 **pynqremote** 이미지 빌드 및 검증

복잡한 커스터마이제이션을 시작하기에 앞서, 먼저 표준 pynqremote 이미지를 빌드하여 전체 빌드 환경이 올바르게 구성되었는지 확인하는 과정이 필수적입니다. 이 과정은 일종의 "연기 테스트(smoke test)" 역할을 하며, 향후 발생할 수 있는 문제의 원인이 환경 설정인지 아니면 커스터마이징 과정의 실수인지를 명확히 구분할 수 있는 기준점을 제공합니다. 성공적으로 빌드된 pynqremote 이미지는 우리가 만들 최종 커스텀 이미지와 비교할 수 있는 중요한 베이스라인이 됩니다.

2.1. pynqremote Make 타겟의 이해

PYNQ v3.1 릴리스 노트와 문서를 살펴보면, pynqremote라는 새로운 make 타겟이 추가된 것을 확인할 수 있습니다.[6] 이 타겟은 Jupyter와 같은 대화형 환경 없이, 원격 접속 및 제어를 목적으로 하는 헤드리스(headless) 시스템용 이미지를 빌드하기 위해 특별히 설계되었습니다. 이는 PYNQ가 단순한 교육 및 프로토타이핑 도구를 넘어, 안정적인 headless 애플리케이션 배포를 위한 프레임워크로 발전하고 있음을 보여주는 중요한 변화입니다.

pynqremote 타겟은 pynq-remote 서버와 같이 원격 운영에 필수적인 패키지들을 설치하는 반면, GUI나 대화형 툴과 관련된 패키지들은 제외할 수 있도록 구성되어 있어 우리의 커스터마이제이션 목표를 위한 완벽한 출발점입니다.

2.2. 기준 이미지 빌드 실행

Docker 컨테이너 내부의 /workspace 디렉토리(호스트의 PYNQ/sdbuild 디렉토리와 연결됨)에서 다음 명령어를 실행하여 빌드를 시작합니다. <board_name> 부분은 대상 보드의 이름(예: Pynq-Z2, ZCU104 등)으로 대체합니다. 이 예제에서는 Pynq-Z2를 사용하겠습니다.

Bash

```
make pynqremote BOARDS=Pynq-Z2
```

이 명령어는 PetaLinux 프로젝트 생성 및 빌드, 루트 파일 시스템 생성, 패키지 설치, 최종 이미지 패키징 등 여러 단계를 거치게 됩니다. 시스템 성능에 따라 수 시간이 소요될 수 있습니다. 빌드 과정에서 수많은 로그가 출력되지만, 최종적으로 ... Build successful... 과 유사한 메시지가 나타나면 성공적으로 완료된 것입니다.

2.3. 생성된 이미지 확인 및 SD 카드 플래싱

PYNQ 공식 문서에는 놀랍게도 빌드 완료 후 생성된 이미지 파일의 위치나 플래싱 방법에 대한 명확한 안내가 부족합니다.[12] 이 공백을 채우기 위해, 빌드 결과물은

sdbuild/output 디렉토리(컨테이너에서는 /workspace/output)에 생성됨을 명확히 합니다. 파일명은 <board_name>-<version>.img 형식(예: Pynq-Z2-v3.1.0.img)을 가집니다.

이제 이 이미지 파일을 SD 카드에 기록해야 합니다. 최소 8GB 용량의 MicroSD 카드를 준비하고 다음 두 가지 방법 중 하나를 선택하여 진행합니다.

2.3.1. BalenaEtcher 사용 (Windows/macOS/Linux)

BalenaEtcher는 그래픽 인터페이스를 제공하여 초보자도 쉽고 안전하게 이미지를 플래싱할 수 있는 도구입니다.[13]

1. **BalenaEtcher** 다운로드 및 설치**:** balena.io/etcher에서 자신의 운영체제에 맞는 버전을 다운로드하여 설치합니다.

2. 이미지 파일 선택**:** Etcher를 실행하고 'Flash from file' 버튼을 클릭하여 방금 빌드한 Pynq-Z2-v3.1.0.img 파일을 선택합니다.
3. 대상 드라이브 선택**:** SD 카드 리더기를 통해 컴퓨터에 연결된 SD 카드를 선택합니다. 이때, 용량을 다시 한번 확인하여 실수로 다른 드라이브를 선택하지 않도록 각별히 주의해야 합니다.
4. 플래싱 시작**:** 'Flash!' 버튼을 클릭하여 플래싱을 시작합니다. 과정이 완료되면 Etcher가 자동으로 검증까지 수행합니다.

2.3.2. dd 명령어 사용 (Linux 고급 사용자)

Linux 환경에 익숙한 사용자는 dd 명령어를 사용하여 더 빠르게 플래싱할 수 있습니다. 다만, 이 방법은 대상 장치를 잘못 지정할 경우 심각한 데이터 손실을 유발할 수 있으므로 주의가 필요합니다.[16]

1. **SD** 카드 장치명 확인**:** SD 카드를 연결하기 전과 후에 lsblk 명령어를 실행하여 새로 추가된 장치명을 확인합니다 (예: /dev/sdc).
2. 파티션 마운트 해제**:** SD 카드가 자동으로 마운트되었다면, umount 명령어를 사용하여 모든 파티션을 마운트 해제해야 합니다. 예를 들어, /dev/sdc1, /dev/sdc2가 마운트되었다면 다음을 실행합니다.
   Bash
   ```
   sudo umount /dev/sdc1
   sudo umount /dev/sdc2
   ```
3. 이미지 쓰기**:** dd 명령어를 사용하여 이미지를 SD 카드에 기록합니다. if에는 이미지 파일 경로를, of에는 장치명(파티션 번호가 아닌)을 지정합니다.
   Bash
   ```
   sudo dd if=output/Pynq-Z2-v3.1.0.img of=/dev/sdc bs=4M status=progress oflag=sync
   ```
   - bs=4M: 쓰기 작업의 블록 크기를 지정하여 속도를 향상시킵니다.
   - status=progress: 진행 상황을 실시간으로 표시합니다.
   - oflag=sync: 쓰기 캐시를 비우고 물리적 쓰기를 보장합니다.

2.4. 초기 부팅 및 연결 테스트

플래싱이 완료된 SD 카드를 보드에 삽입하고, 부팅 모드 점퍼를 'SD'로 설정합니다.[17] 이더넷 케이블을 공유기나 컴퓨터에 직접 연결한 후 보드의 전원을 켭니다.

부팅이 완료되면, 호스트 PC에서 간단한 Python 스크립트를 작성하여 PYNQ.remote를 통한 연결을 테스트합니다. 먼저, 호스트 PC에 pynq 라이브러리를 설치합니다.

Bash

```
pip install pynq
```

그 다음, 아래와 같은 Python 스크립트(test_connection.py)를 작성합니다. pynq-z2 부분은 보드의 호스트명이며, 네트워크 환경에 따라 IP 주소로 대체해야 할 수 있습니다.

Python

```
from pynq.remote import RemoteDevice

# 'pynq-z2'는 보드의 기본 호스트명입니다.
# DHCP를 통해 할당받은 IP 주소를 직접 사용할 수도 있습니다.
# 예: board_ip = "192.168.2.99"
board_hostname = "pynq-z2"

try:
    # RemoteDevice 객체를 생성하여 보드에 연결합니다.
    # 기본 사용자 이름은 'xilinx'이고 비밀번호도 'xilinx'입니다.
    device = RemoteDevice(board_hostname, "xilinx", "xilinx")

    print(f"Successfully connected to {board_hostname}")

    # 보드의 PYNQ 버전 정보를 가져옵니다.
    pynq_version = device.pynq_version
    print(f"PYNQ Version on board: {pynq_version}")

    # 보드의 OS 이름 정보를 가져옵니다.
    os_name = device.operating_system
    print(f"Operating System: {os_name}")

except Exception as e:
    print(f"Failed to connect to the board: {e}")
```

이 스크립트를 실행했을 때, 성공적으로 연결되었다는 메시지와 함께 보드의 PYNQ 버전 및 OS 정보가 출력된다면, 기준 pynqremote 이미지 빌드와 배포가 모두 성공적으로 완료된 것입니다. 이제 우리는 이 안정적인 상태를 기반으로 본격적인 커스터마이제이션을 진행할 준비가 되었습니다.

섹션 **3:** 고급 커스터마이제이션**:** 최소 **C++/gRPC** 루트 파일 시스템 제작

이 섹션은 본 보고서의 핵심으로, 표준 pynqremote 이미지를 우리의 목표에 맞게 변형시키는 과정을 다룹니다. 목표는 명확합니다: Jupyter와 Python 생태계를 완전히 제거하고, C++ 개발에 필수적인 PYNQ.cpp와 원격 통신을 위한 gRPC 라이브러리만을 포함하는 경량화된 이미지를 만드는 것입니다. 이를 위해 PYNQ sdbuild 프레임워크의 내부 동작 방식을 깊이 이해하고, 패키지 설치 목록을 직접 수정하며, 나아가 표준 저장소에 없는 gRPC를 시스템에 통합하기 위해 PetaLinux/Yocto 레시피를 직접 작성하는 고급 기법을 사용합니다. 이 과정은 단순한 설정 변경을 넘어, 임베디드 리눅스 시스템을 재구성하는 전문적인 작업입니다.

3.1. **sdbuild** 프로세스 해부

PYNQ의 sdbuild는 단순한 셸 스크립트가 아니라, 여러 단계로 구성된 복잡한 Makefile 기반의 빌드 시스템입니다. 효과적인 커스터마이징을 위해서는 이 구조를 이해해야 합니다.

- 빌드 단계**(Stages):** sdbuild는 기본 Ubuntu 루트 파일 시스템 위에 여러 단계에 걸쳐 패키지를 설치합니다. STAGE2, STAGE3, STAGE4가 핵심적인 단계입니다.[9]
  - STAGE2: gcc, udev와 같은 기본적인 개발 도구와 시스템 유틸리티, 그리고 PYNQ의 핵심 의존성(예: python_packages_jammy, jupyter)이 설치되는 단계입니다. 이 단계는 보드에 상관없이 공통적으로 적용되는 '보드 불가지론적(board-agnostic)' 이미지를 생성합니다.
  - STAGE3: xrtlib, pynq와 같이 PYNQ 기능에 더 직접적으로 관련된 패키지들이 설치됩니다.
  - STAGE4: 각 보드별로 특화된 패키지(예: 특정 보드의 이더넷 설정, LED 제어 등)가 설치되는 최종 단계입니다.
- 보드 명세 파일**(.spec):** 각 보드는 자신만의 명세 파일(예: PYNQ/sdbuild/boards/Pynq-Z2/Pynq-Z2.spec)을 가집니다. 이 파일은 해당 보드의 아키텍처(ARCH), PetaLinux BSP 파일 경로(BSP), 그리고 STAGE4에서 설치할 추가 패키지 목록(STAGE4_PACKAGES) 등을 정의하는 변수들을 포함합니다.[9]

이 구조를 통해 알 수 있는 중요한 사실은, Python이나 Jupyter와 같은 핵심 패키지들은 보드별 설정인 STAGE4가 아닌, 공통 단계인 STAGE2와 STAGE3에서 설치된다는 점입니다. 따라서 단순히 .spec 파일만 수정해서는 이들을 제거할 수 없으며, sdbuild의 더 깊은 곳을 수정해야 합니다.[18]

3.2. Python 생태계의 체계적 제거

이제 Python 관련 패키지들을 빌드 과정에서 원천적으로 배제하는 작업을 진행합니다. 이는 sdbuild의 핵심 설정 파일을 직접 수정하는 침습적인(invasive) 작업이므로, 정확한 경로와 변수명을 숙지해야 합니다. 대상 아키텍처에 따라 arm 또는 aarch64 경로를 선택해야 합니다. (예: Pynq-Z2는 arm, ZCU104는 aarch64)

3.2.1. 1단계: STAGE2 패키지 목록 수정

STAGE2에서 Jupyter와 기본 Python 패키지들을 설치하는 부분을 제거합니다. 텍스트 편집기로 다음 파일을 엽니다. (aarch64 기준)

<PYNQ_ROOT>/sdbuild/ubuntu/jammy/aarch64/packages.mk

파일 내에서 STAGE2_PACKAGES_aarch64 변수를 찾아 python_packages_jammy와 jupyter를 제거합니다.

수정 전:

Makefile

```
STAGE2_PACKAGES_aarch64 := gcc-mb udev libsds python_packages_jammy jupyter
STAGE2_PACKAGES_aarch64 += sigrok
STAGE2_PACKAGES_aarch64 += bootpy
STAGE2_PACKAGES_aarch64 += clear_pl_statefile
```

수정 후:

Makefile

```
STAGE2_PACKAGES_aarch64 := gcc-mb udev libsds
STAGE2_PACKAGES_aarch64 += sigrok
STAGE2_PACKAGES_aarch64 += bootpy
STAGE2_PACKAGES_aarch64 += clear_pl_statefile
```

3.2.2. 2단계: STAGE3 패키지 목록 수정

동일한 packages.mk 파일에서, STAGE3에 정의된 pynq 패키지를 제거합니다. pynq 패키지는 Python 라이브러리 자체이므로 반드시 제거해야 합니다. 하지만 xrtlib는 PYNQ.cpp의 핵심 의존성이므로 절대로 제거해서는 안 됩니다.[20]

수정 전:

Makefile

```
STAGE3_PACKAGES_aarch64 := xrtlib pynq x11 resizefs
```

수정 후:

Makefile

```
STAGE3_PACKAGES_aarch64 := xrtlib x11 resizefs
```

3.2.3. 3단계: 보드 .spec 파일 정리

마지막으로, 대상 보드의 .spec 파일을 열어 STAGE4_PACKAGES 변수에 남아있을 수 있는 Python 관련 패키지(예: jupyter, pandas 등)를 모두 제거합니다.

예시 (Pynq-Z2.spec):

Makefile

```
# 수정 전 예시
STAGE4_PACKAGES_Pynq-Z2 := boot_leds ethernet pynq jupyter

# 수정 후 예시
STAGE4_PACKAGES_Pynq-Z2 := boot_leds ethernet
```

다음 표는 Python 생태계 제거를 위한 핵심 수정 사항을 요약한 것입니다.

표 **2: sdbuild** 최소화를 위한 패키지 수정

수정 파일 경로 (aarch64 기준)	변수명	제거할 패키지
sdbuild/ubuntu/jammy/aarch64/packages.mk	STAGE2_PACKAGES_aarch64	python_packages_jammy, jupyter
sdbuild/ubuntu/jammy/aarch64/packages.mk	STAGE3_PACKAGES_aarch64	pynq
sdbuild/boards/<board>/<board>.spec	STAGE4_PACKAGES_<board>	jupyter, pynq, pandas 등

3.3. gRPC 통합을 위한 커스텀 PetaLinux/Yocto 레시피 작성

gRPC는 표준 Ubuntu 저장소나 PYNQ 패키지에 포함되어 있지 않으므로 [21], 단순히 패키지

목록에 추가하는 것만으로는 설치할 수 없습니다. 이처럼 외부의 복잡한 라이브러리를 시스템에 통합하기 위해서는 PYNQ 빌드 시스템의 근간을 이루는 PetaLinux(Yocto) 빌드 시스템을 직접 다루어야 합니다.[22] 이 과정은

gRPC와 그 의존성 라이브러리들(예: protobuf, abseil-cpp 등)을 소스 코드로부터 컴파일하고 패키징하는 방법을 Yocto 빌드 시스템에 알려주는 '레시피(.bb)'를 작성하는 것을 포함합니다.

3.3.1. 1단계: 커스텀 메타-레이어 생성

Yocto 프로젝트의 표준 관행에 따라, 커스텀 레시피들을 담을 '메타-레이어'를 생성합니다. PYNQ 소스 트리 내에 다음과 같은 디렉토리 구조를 만듭니다.

```
<PYNQ_ROOT>/sdbuild/boot/meta-custom/
├── conf
│   └── layer.conf
└── recipes-core
    └── grpc
        ├── grpc_1.51.1.bb
        └── ... (의존성 레시피 파일들)
```

conf/layer.conf 파일에는 이 레이어가 빌드 시스템에 자신을 알리는 데 필요한 기본 정보를 기입합니다.

3.3.2. 2단계: gRPC 및 의존성 레시피 작성

gRPC를 빌드하기 위해서는 protobuf, c-ares, re2, abseil-cpp 등 여러 라이브러리가 먼저 빌드되어야 합니다. 각 라이브러리에 대한 .bb 레시피 파일을 작성해야 합니다. 레시피 파일은 일반적으로 다음과 같은 정보를 포함합니다.

- DESCRIPTION: 패키지에 대한 설명
- LICENSE: 라이선스 정보
- SRC_URI: 소스 코드를 다운로드할 URL (git 저장소 또는 tarball)
- DEPENDS: 빌드 시 의존하는 다른 레시피들
- inherit cmake: 빌드 시스템(예: cmake) 지정

- 컴파일 및 설치 옵션

이는 매우 전문적인 작업이며, Yocto/OpenEmbedded 커뮤니티에서 기존에 작성된 레시피들을 참조하여 작성하는 것이 효율적입니다.

3.3.3. 3단계: PetaLinux 빌드에 메타-레이어 통합

작성된 커스텀 메타-레이어를 PYNQ의 PetaLinux 빌드 과정에 포함시켜야 합니다. PYNQ sdbuild는 각 보드에 대한 PetaLinux 프로젝트를 sdbuild/build/<board>/petalinux_project와 같은 임시 경로에 생성합니다. 이 프로젝트의 build/conf/bblayers.conf 파일에 우리의 meta-custom 레이어 경로를 추가해야 합니다. 이를 자동화하기 위해, sdbuild/Makefile이나 관련 빌드 스크립트를 수정하여 PetaLinux 프로젝트가 생성된 직후 bblayers.conf 파일에 레이어 경로를 추가하는 로직을 삽입할 수 있습니다.

3.3.4. 4단계: gRPC를 루트 파일 시스템에 배포

PetaLinux 빌드를 통해 gRPC 라이브러리가 성공적으로 컴파일되었다면, 이제 이 결과물을 최종 Ubuntu 루트 파일 시스템으로 복사해야 합니다. PetaLinux 빌드의 결과물(컴파일된 라이브러리, 헤더 파일 등)은 PetaLinux 프로젝트 내의 sysroot 디렉토리에 위치하게 됩니다. 우리는 이 두 개의 분리된 빌드 시스템(PetaLinux 부트 파티션 빌드와 PYNQ Ubuntu 루트 파일 시스템 빌드)을 연결하는 다리가 필요합니다.

이를 위해 새로운 PYNQ 스타일 패키지를 sdbuild/packages/grpc와 같은 경로에 생성합니다. 이 패키지에는 qemu.sh 스크립트가 포함됩니다. 이 스크립트는 루트 파일 시스템을 빌드하는 chroot 환경 내부에서 실행되며, PetaLinux 빌드 결과물 경로에서 libgrpc.so와 같은 라이브러리 파일들을 찾아 최종 루트 파일 시스템의 /usr/local/lib와 같은 적절한 위치로 복사하는 역할을 수행합니다. 이렇게 함으로써 PetaLinux에서 컴파일된 gRPC가 최종 이미지에 포함되게 됩니다.

3.4. PYNQ.cpp 및 XRT 의존성 유지 확인

이 모든 과정에서 가장 중요한 것은 우리의 목표인 PYNQ.cpp를 사용할 수 있는 환경을 유지하는 것입니다. PYNQ.cpp는 Xilinx Runtime(XRT) 라이브러리에 의존합니다.[20] 따라서 3.2.2 단계에서

STAGE3 패키지 목록을 수정할 때 xrtlib를 실수로 제거하지 않았는지 다시 한번 확인해야 합니다. xrtlib가 최종 이미지에 포함되어야만, 타겟 보드에서 C++ 코드가 FPGA 하드웨어와 통신할 수 있습니다.

섹션 **4:** 커스텀 빌드 실행 및 이미지 검증

이제 모든 커스터마이징 작업이 완료되었습니다. 이 섹션에서는 수정된 빌드 스크립트를 실행하여 최종 이미지를 생성하고, 생성된 이미지를 보드에 배포하기 전에 호스트 시스템에서 그 내용물을 철저히 검증하는 방법을 안내합니다. 이 사전 검증 단계는 잘못된 이미지를 플래싱하여 디버깅하는 데 소요되는 시간을 크게 줄여주는 효율적인 피드백 루프를 제공합니다.

4.1. 전체 커스텀 빌드 실행

수정된 패키지 목록과 새로 추가된 gRPC 레시피를 반영하여 전체 빌드를 다시 실행합니다. 이때, 이전 빌드에서 캐시된 루트 파일 시스템을 사용하지 않고 우리가 수정한 내용을 기반으로 처음부터 다시 생성하도록 REBUILD_PYNQ_ROOTFS=True 플래그를 추가해야 합니다.[5]

Docker 컨테이너 내부의 /workspace 디렉토리에서 다음 명령어를 실행합니다.

Bash

```
make REBUILD_PYNQ_ROOTFS=True pynqremote BOARDS=Pynq-Z2
```

이 명령은 섹션 2에서 실행했던 것보다 더 많은 작업을 수행합니다. 특히 PetaLinux 빌드 과정에서 meta-custom 레이어를 포함하여 gRPC와 그 의존성을 컴파일하고, 루트 파일 시스템 생성 시에는 수정된 패키지 목록을 적용할 것입니다. 이 과정 역시 수 시간이 소요될 수 있습니다.

4.2. 일반적인 **PetaLinux** 및 **sdbuild** 오류 해결

커스텀 빌드 과정에서는 다양한 오류가 발생할 수 있습니다. 다음은 발생 가능성이 높은 오류 유형과 그에 대한 기본적인 해결 접근법입니다.

- 레시피 소스 다운로드 실패 **(Fetch Failures):** gRPC 레시피에 명시된 소스 코드의 URL이 변경되거나 유효하지 않을 경우 발생합니다. PetaLinux 빌드 로그(sdbuild/build/<board>/petalinux_project/build/log.do_fetch.<...> 파일)를 확인하여 어떤 URL에서 문제가 발생했는지 파악하고, .bb 레시피 파일의 SRC_URI 변수를 올바른 주소로 수정한 후 다시 시도합니다.
- 컴파일 오류 **(Compilation Errors):** 소스 코드를 컴파일하는 과정에서 오류가 발생하는 경우로, 주로 의존성 라이브러리가 누락되었거나 컴파일러 옵션이 잘못되었을 때 발생합니다. 상세한 오류 메시지는 해당 레시피의 빌드 로그(log.do_compile.<...>)에 기록됩니다. Yocto의 bitbake -e <recipe-name> 명령어를 사용하면 해당 레시피의 모든 환경 변수와 의존성을 확인할 수 있어 디버깅에 도움이 됩니다.
- **chroot/QEMU** 오류**:** sdbuild의 패키지 설치 단계(qemu.sh 스크립트 실행 중)에서 발생하는 오류는 QEMU 에뮬레이션 환경의 특성상 원인 파악이 까다로울 수 있습니다.[9] 문제가 되는 패키지의 qemu.sh 스크립트 상단에 set -x를 추가하면 스크립트의 모든 실행 명령어가 터미널에 출력되므로, 어느 부분에서 실패하는지 추적하는 데 유용합니다.

4.3. 빌드 후 이미지 분석

빌드가 성공적으로 완료되면, sdbuild/output 디렉토리에 새로운 커스텀 이미지 파일(예: Pynq-Z2-v3.1.0.img)이 생성됩니다. 이 이미지를 SD 카드에 플래싱하기 전에, 호스트 시스템에서 직접 마운트하여 내용이 우리의 의도대로 구성되었는지 확인합니다.

4.3.1. 이미지 파일 마운트

Linux 호스트 시스템에서 kpartx와 mount 유틸리티를 사용하여 이미지 파일 내의 루트 파일 시스템 파티션에 접근할 수 있습니다.

1. **kpartx** 설치**:**
   Bash
   sudo apt-get update
   sudo apt-get install kpartx

2. 이미지 파티션 매핑**:** kpartx를 사용하여 이미지 파일 내의 파티션들을 루프백 디바이스로

매핑합니다. PYNQ 이미지의 루트 파일 시스템은 보통 두 번째 파티션입니다.
Bash
sudo kpartx -av output/Pynq-Z2-v3.1.0.img

이 명령은 loop0p1, loop0p2와 같은 디바이스 노드를 /dev/mapper/ 아래에 생성합니다.

3. 마운트 포인트 생성 및 마운트:
Bash
mkdir -p /mnt/pynq_rootfs
sudo mount /dev/mapper/loop0p2 /mnt/pynq_rootfs

이제 /mnt/pynq_rootfs 디렉토리를 통해 이미지의 파일 시스템을 탐색할 수 있습니다.

4.3.2. 파일 시스템 내용 검증

마운트된 디렉토리에서 다음 항목들을 확인하여 커스터마이제이션이 성공적으로 적용되었는지 검증합니다.

- **Python** 및 **Jupyter**의 부재 확인:
Bash
ls /mnt/pynq_rootfs/usr/local/bin/jupyter
ls /mnt/pynq_rootfs/usr/lib/python3.10

위 명령어들은 "No such file or directory" 오류를 반환해야 합니다. 이는 Python 런타임과 Jupyter 관련 파일들이 성공적으로 제거되었음을 의미합니다.
- **gRPC** 및 **XRT** 라이브러리의 존재 확인:
Bash
find /mnt/pynq_rootfs -name "*libgrpc*so*"
find /mnt/pynq_rootfs -name "*libxrt*so*"

첫 번째 명령어는 우리가 추가한 gRPC 공유 라이브러리 파일들을 찾아야 합니다. 두 번째 명령어는 PYNQ.cpp의 필수 의존성인 XRT 라이브러리가 여전히 존재함을 확인합니다. 이 두 가지가 모두 존재해야 우리의 최종 목표를 달성할 수 있습니다.

4.3.3. 마운트 해제

검증이 끝나면 반드시 마운트를 해제하고 디바이스 매핑을 제거해야 합니다.

Bash

```
sudo umount /mnt/pynq_rootfs
sudo kpartx -d output/Pynq-Z2-v3.1.0.img
```

이 사전 검증 단계를 통해 이미지가 올바르게 생성되었음을 확신했다면, 이제 섹션 2.3에서 설명한 방법(BalenaEtcher 또는 dd)을 사용하여 이 커스텀 이미지를 SD 카드에 플래싱하고 최종 테스트를 진행할 준비가 된 것입니다.

섹션 **5:** 배포, 호스트 구성 및 종단 간 검증

지금까지의 과정을 통해 우리는 이론적으로 완벽한 커스텀 이미지를 제작했습니다. 마지막 단계는 이 이미지를 실제 하드웨어에 배포하고, 호스트 PC와 연동하여 우리가 설계한 'Python 호스트 -> C++ 타겟' 원격 제어 아키텍처가 실제로 동작하는지 종단 간(end-to-end)으로 검증하는 것입니다. 이 섹션에서는 간단한 "Hello, C++ World" 예제를 통해 PYNQ.remote, PYNQ.cpp, gRPC가 모두 유기적으로 작동함을 증명합니다.

5.1. 타겟 배포 및 첫 부팅

섹션 4에서 검증을 마친 커스텀 .img 파일을 SD 카드에 플래싱합니다. 플래싱된 SD 카드를 보드에 삽입하고, 부팅 모드 점퍼를 'SD'로 설정한 후 전원을 켭니다.

이번에는 부팅 과정을 좀 더 자세히 관찰하기 위해 시리얼 터미널을 연결하는 것이 좋습니다. 보드의 UART 포트와 컴퓨터를 Micro USB 케이블로 연결하고, PuTTY(Windows)나 minicom(Linux)과 같은 터미널 프로그램을 사용하여 시리얼 포트에 접속합니다. 일반적인 시리얼 설정은 다음과 같습니다 [24]:

- Baud rate: 115200
- Data bits: 8
- Stop bits: 1
- Parity: None
- Flow control: No

부팅 로그를 실시간으로 모니터링하면, 커널 패닉이나 특정 서비스 시작 실패와 같은 잠재적인

문제를 조기에 발견할 수 있습니다. 부팅이 완료되면 프롬프트에서 ifconfig 명령어를 실행하여 보드가 네트워크로부터 할당받은 IP 주소를 확인합니다.

5.2. 호스트 측 구성

이제 원격 제어를 수행할 호스트 PC의 환경을 설정합니다.

1. **Python** 가상 환경 생성 및 활성화**:** 프로젝트의 의존성을 격리하기 위해 가상 환경을 사용하는 것이 좋습니다.
   Bash
   ```
   python3 -m venv pynq_remote_env
   source pynq_remote_env/bin/activate
   ```

2. 필수 라이브러리 설치**:** 호스트에서는 PYNQ.remote를 통한 보드 제어와 gRPC 클라이언트 역할을 수행해야 하므로, 다음 라이브러리들을 설치합니다.
   Bash
   ```
   pip install pynq grpcio grpcio-tools
   ```

   - pynq: PYNQ.remote 기능이 포함되어 있습니다.
   - grpcio: Python용 gRPC 런타임입니다.
   - grpcio-tools: .proto 파일로부터 Python 클라이언트 및 서버 코드를 생성하는 데 사용됩니다.

5.3. 검증 워크플로우**: "Hello, C++ World"** 원격 호출

이 종단 간 테스트는 우리가 구축한 모든 구성 요소가 올바르게 작동하는지를 최종적으로 증명합니다.

5.3.1. 1단계**: gRPC** 서비스 정의 **(.proto** 파일 작성**)**

먼저, 호스트와 타겟 간에 주고받을 메시지와 서비스의 형식을 Protocol Buffers를 사용하여 정의합니다. hardware_manager.proto라는 파일을 생성하고 다음과 같이 작성합니다.

Protocol Buffers

```
syntax = "proto3";

// 빈 메시지 정의
message Empty {}

// 서비스 정의
service HardwareManager {
  // FPGA 정보를 요청하는 RPC 메소드
  rpc GetFpgaInfo(Empty) returns (FpgaInfoResponse);
}

// 응답 메시지 정의
message FpgaInfoResponse {
  string info_string = 1;
}
```

이 .proto 파일은 파라미터 없이 호출하면 FPGA 정보가 담긴 문자열을 반환하는 간단한 GetFpgaInfo라는 원격 프로시저를 정의합니다.

5.3.2. 2단계: 타겟 C++ 서버 개발

이제 타겟 보드에서 실행될 gRPC 서버 애플리케이션을 C++로 작성합니다. 이 애플리케이션은 PYNQ.cpp를 사용하여 하드웨어 정보를 얻고, gRPC를 통해 이 정보를 외부에 제공하는 역할을 합니다.

1. 코드 생성: grpc_cpp_plugin을 사용하여 .proto 파일로부터 C++ 서버 인터페이스 코드를 생성합니다.
2. 서버 구현: 생성된 인터페이스를 상속받아 GetFpgaInfo 메소드의 실제 로직을 구현합니다. 이 로직 내부에서 PYNQ.cpp의 API를 호출하여 하드웨어 관련 정보를 조회할 수 있습니다. 여기서는 간단하게 정적 문자열을 반환하는 예시를 보여줍니다.

   C++

   ```
   #include <iostream>
   #include <string>
   #include <grpcpp/grpcpp.h>
   #include "hardware_manager.grpc.pb.h"
   // #include <pynq_cpp.h> // 실제 PYNQ.cpp 헤더

   class HardwareManagerServiceImpl final : public HardwareManager::Service {
   ```

```
    grpc::Status GetFpgaInfo(grpc::ServerContext* context, const Empty* request,
                             FpgaInfoResponse* reply) override {
        std::string info = "Hello from PYNQ.cpp on Zynq Target!";
        // 예: pynq::Device& device = pynq::Device::get_shared_device();
        // info = "Device Name: " + device.name;
        reply->set_info_string(info);
        return grpc::Status::OK;
    }
};

void RunServer() {
    std::string server_address("0.0.0.0:50051");
    HardwareManagerServiceImpl service;

    grpc::ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    builder.RegisterService(&service);

    std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_address << std::endl;
    server->Wait();
}

int main(int argc, char** argv) {
    RunServer();
    return 0;
}
```

3. 크로스 컴파일 및 배포: PetaLinux SDK(또는 적절한 크로스 컴파일 툴체인)를 사용하여 이 C++ 코드를 보드의 아키텍처(ARM/AArch64)에 맞게 컴파일합니다. 컴파일된 실행 파일을 scp 등을 이용해 보드로 복사하고, 보드의 터미널에서 실행하여 서버를 구동시킵니다.

5.3.3. 3단계: 호스트 Python 클라이언트 개발

호스트 PC에서는 PYNQ.remote와 gRPC를 함께 사용하는 Python 클라이언트를 작성합니다.

1. 코드 생성: grpc_tools.protoc를 사용하여 .proto 파일로부터 Python 클라이언트 스텁 코드를 생성합니다.
2. 클라이언트 구현: pynq.remote로 보드에 먼저 연결하여 시스템 수준의 제어 채널을 확보한 후, gRPC 채널을 생성하여 GetFpgaInfo RPC를 호출합니다.

```python
import grpc
from pynq.remote import RemoteDevice
import hardware_manager_pb2
import hardware_manager_pb2_grpc

# 보드 IP 주소 또는 호스트명
BOARD_IP = "192.168.2.99"
BOARD_USER = "xilinx"
BOARD_PASS = "xilinx"

def run():
    # 1. PYNQ.remote를 사용한 연결 (시스템 제어/상태 확인용)
    try:
        device = RemoteDevice(BOARD_IP, BOARD_USER, BOARD_PASS)
        print(f"PYNQ.remote connection successful to {device.name}")
        print(f"Board OS: {device.operating_system}")
    except Exception as e:
        print(f"PYNQ.remote connection failed: {e}")
        return

    # 2. gRPC를 사용한 원격 프로시저 호출
    grpc_address = f"{BOARD_IP}:50051"
    with grpc.insecure_channel(grpc_address) as channel:
        stub = hardware_manager_pb2_grpc.HardwareManagerStub(channel)
        print(f"gRPC client connected to {grpc_address}")

        # GetFpgaInfo RPC 호출
        response = stub.GetFpgaInfo(hardware_manager_pb2.Empty())

        print("\n--- gRPC Response from C++ Server ---")
        print(response.info_string)
        print("-------------------------------------")

if __name__ == '__main__':
    run()
```

5.3.4. 4단계: 실행 및 검증

타겟 보드에서 C++ 서버가 실행 중인 상태에서, 호스트 PC에서 Python 클라이언트 스크립트를 실행합니다. 성공적으로 실행된다면, 터미널에는 다음과 같은 출력이 나타날 것입니다.

```
PYNQ.remote connection successful to zynq
Board OS: Linux
gRPC client connected to 192.168.2.99:50051

--- gRPC Response from C++ Server ---
Hello from PYNQ.cpp on Zynq Target!
-----------------------------------
```

이 결과는 우리가 설계한 모든 요소—커스텀 빌드된 최소 OS, XRT와 PYNQ.cpp 라이브러리, 새로 통합된 gRPC 프레임워크, 그리고 PYNQ.remote를 통한 호스트 제어—가 완벽하게 연동되어 동작함을 명확하게 증명합니다. 이로써 본 보고서의 최종 목표가 성공적으로 달성되었습니다.

섹션 **6:** 결론 및 향후 개발 방향

6.1. 성과 요약

본 보고서는 PYNQ 3.1 프레임워크를 사용하여 원격 C++ 개발에 최적화된 맞춤형 SD 카드 이미지를 제작하는 전 과정을 체계적으로 안내했습니다. 이 과정을 통해 달성한 핵심 성과는 다음과 같습니다.

1. 재현 가능한 빌드 환경 구축**:** PYNQ 3.1의 공식 권장 사항에 따라 Ubuntu 22.04, AMD 2024.1 툴체인, 그리고 Docker를 기반으로 한 안정적이고 격리된 빌드 환경을 성공적으로 구축했습니다. 이는 복잡한 임베디드 리눅스 빌드에서 발생할 수 있는 수많은 환경 변수 문제를 원천적으로 차단하는 현대적인 접근 방식입니다.
2. **PYNQ** 이미지의 경량화**:** PYNQ sdbuild 시스템의 내부 패키지 구조를 분석하고 직접 수정하여, 프로토타이핑 단계에 주로 사용되는 Jupyter Notebook과 전체 Python 생태계를 성공적으로 제거했습니다. 이를 통해 리소스가 제한된 임베디드 환경에 더 적합한 최소한의 실행 환경을 확보했습니다.
3. 외부 라이브러리의 체계적 통합**:** 표준 저장소에 존재하지 않는 gRPC C++ 라이브러리를

시스템에 통합하기 위해, PYNQ 빌드 시스템의 근간인 PetaLinux/Yocto를 직접 다루는 고급 기법을 적용했습니다. 커스텀 메타-레이어와 레시피를 작성하여 gRPC를 소스 코드로부터 컴파일하고, 이를 최종 루트 파일 시스템에 배포하는 일련의 과정을 통해 PYNQ 빌드 시스템의 확장 가능성을 입증했습니다.

4. 종단 간 원격 **C++** 워크플로우 검증**:** 최종적으로, 제작된 커스텀 이미지를 타겟 보드에 배포하고, PYNQ.remote를 사용하는 Python 호스트 클라이언트와 PYNQ.cpp 및 gRPC를 사용하는 C++ 타겟 서버 간의 완벽한 통신을 시연했습니다. 이는 본 보고서에서 제시한 아키텍처가 이론뿐만 아니라 실제 하드웨어에서도 완벽하게 동작함을 증명하는 최종 결과물입니다.

결론적으로, 본 가이드는 PYNQ를 단순한 Python 기반 프로토타이핑 도구에서 벗어나, 특정 요구사항에 맞춰 고도로 최적화할 수 있는 강력한 임베디드 시스템 개발 플랫폼으로 활용하는 방법을 제시했습니다.

6.2. 향후 개발을 위한 제언

본 보고서에서 구축한 기반을 바탕으로, 다음과 같은 방향으로 시스템을 더욱 발전시킬 수 있습니다.

- 빌드 프로세스 자동화**:** 본 보고서에서 수행한 수동적인 파일 수정 및 빌드 명령어 실행 과정을 셸 스크립트나 CI/CD(Continuous Integration/Continuous Deployment) 파이프라인으로 자동화할 수 있습니다. 이를 통해 새로운 gRPC 버전이 출시되거나 PYNQ가 업데이트될 때마다 버튼 하나로 새로운 커스텀 이미지를 생성하는 완전 자동화된 워크플로우를 구축할 수 있습니다.
- **gRPC** 서비스 확장**:** "Hello, World" 수준의 gRPC 서비스를 확장하여, PYNQ.cpp를 통해 제어되는 복잡한 하드웨어 가속기(Overlay)의 레지스터를 읽고 쓰거나, DMA를 통한 데이터 전송을 트리거하는 등 실질적인 하드웨어 제어 기능을 원격으로 호출할 수 있도록 .proto 파일과 서버 로직을 고도화할 수 있습니다. 이는 진정한 의미의 원격 하드웨어 가속 서비스를 구현하는 길이 될 것입니다.
- 커스텀 레이어 유지보수**:** gRPC와 그 의존성 라이브러리들은 지속적으로 업데이트됩니다. 작성된 커스텀 PetaLinux/Yocto 레시피의 SRC_URI와 버전 정보를 주기적으로 업데이트하여, 보안 패치 및 성능 개선이 포함된 최신 라이브러리를 시스템에 반영하는 유지보수 전략을 수립하는 것이 중요합니다.
- 보안 강화**:** 현재 구현된 gRPC 서버는 InsecureServerCredentials를 사용하여 암호화되지 않은 통신을 수행합니다. 실제 제품 환경에서는 TLS/SSL 인증서를 사용하여 통신을 암호화하고, 상호 인증(mTLS)을 통해 허가된 클라이언트만 서버에 접속할 수 있도록 보안을 강화하는 작업이 반드시 필요합니다.

참고 자료

1. PYNQ | Python Productivity for AMD Adaptive Computing platforms, 9월 16,

2025에 액세스, http://www.pynq.io/
2. Building PYNQ image for ZCU111 w/ 2019.2 or 2020.1 - Support, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/building-pynq-image-for-zcu111-w-2019-2-or-2020-1/1858
3. Port PYNQ 3.01 to new board - Support, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/port-pynq-3-01-to-new-board/5110
4. Best Vivado/Vitis Version for PYNQ-Z1 with SD Image v3.0.1 or Most Stable Combo?, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/best-vivado-vitis-version-for-pynq-z1-with-sd-image-v3-0-1-or-most-stable-combo/8329
5. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_sd_card.html
6. Releases · Xilinx/PYNQ - GitHub, 9월 16, 2025에 액세스, https://github.com/Xilinx/PYNQ/releases
7. Change Log — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/latest/changelog.html
8. PYNQ 3.0.1 VirtualBox install and SD build - Learn, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/pynq-3-0-1-virtualbox-install-and-sd-build/4859
9. sdbuild/README.md · 7f4a3b92e09714d02545ffc8432124d5d9f1ad0d · paulrr2 / PYNQ - GitLab at Illinois, 9월 16, 2025에 액세스, https://gitlab-03.engr.illinois.edu/paulrr2/PYNQ/-/blob/7f4a3b92e09714d02545ffc8432124d5d9f1ad0d/sdbuild/README.md
10. PYNQ v3.0.1 Build Failure for ZCU104 - Support, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/pynq-v3-0-1-build-failure-for-zcu104/5398
11. Remote Image Build Guide — Python productivity for Zynq (Pynq), 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/image_build.html
12. PYNQ SD Card image — Python productivity for Zynq (Pynq), 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_sd_card.html
13. Flash the SD Card (only once) - TU/e PYNQ Pages, 9월 16, 2025에 액세스, https://pynq.tue.nl/general/flash-sd-card/
14. Flash with Balena Etcher - Raspberry Pi Forums, 9월 16, 2025에 액세스, https://forums.raspberrypi.com/viewtopic.php?t=279692
15. Pynq Z1_starting help : r/FPGA - Reddit, 9월 16, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/flvbv5/pynq_z1_starting_help/
16. Appendix — NengoPYNQ 0.3.3.dev0 docs, 9월 16, 2025에 액세스, https://www.nengo.ai/nengo-pynq/appendix.html
17. PYNQ-Z1 Reference Manual - Digilent, 9월 16, 2025에 액세스, https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual
18. How to build a minimal PYNQ 3.1 image (no Jupyter), only pynq Python library for Overlay(), fitting in 8 GB eMMC, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/how-to-build-a-minimal-pynq-3-1-image-no-jupyter-only-pynq-python-library-for-overlay-fitting-in-8-gb-emmc/8712
19. Build PYNQ-2.4 SD card Image for Zedboard v2018.3 - Learn, 9월 16, 2025에 액세스,

https://discuss.pynq.io/t/build-pynq-2-4-sd-card-image-for-zedboard-v2018-3/2702
20. How to build projects with PYNQ.cpp — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/pynq_cpp/cpp_install.html
21. DIPTE/awesome-stars - GitHub, 9월 16, 2025에 액세스, https://github.com/DIPTE/awesome-stars
22. Creating a Custom Petalinux Image for Pynq-Z2 : r/FPGA - Reddit, 9월 16, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/1ao89df/creating_a_custom_petalinux_image_for_pynqz2/
23. Petalinux using a version >=2.6 in 2021/2022 tools - Support - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/petalinux-using-a-version-2-6-in-2021-2022-tools/5468
24. PYNQ-Z1 Setup Guide — Python productivity for Zynq (Pynq), 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/getting_started/pynq_z1_setup.html