1&2

2024년 9월 9일 월요일 오전 8:37

임베디드 리눅스의 4요소

1. **툴체인(toolchain):**
   타깃 장치를 위한 코드를 만드는데 필요한 컴파일러와 기타 도구로 구성

2. **부트로더(bootloader):**
   보드를 초기화 하고 리눅스 커널을 로드하는 프로그램

3. **커널:**
   시스템의 심장부로, 시스템 자원을 관리하며 하드웨어와의 접점

4. **루트 파일 시스템:**
   커널이 초기화를 끝낸 뒤 실행되는 라이브러리와 프로그램을 담고 있음

<오픈 소스>

1. **GPL 같은 Copyleft 라이선스**
   본질적으로 라이선스의 조건을 변경하지 않는 한 소스 코드 수정 및 사용 허용

2. **BSD, MIT와 같은 허용적인 라이선스**
   소스 코드 공유

<임베디드 리눅스를 위한 하드웨어 선택하기>

1. 커널이 지원하는 CPU 아키텍처
2. 적절한 크기의 램 필요
3. 비휘발성 메모리가 있음
4. 디버그 포트는 매우 유용하며 UART 기반 시리얼 포트를 선호
5. 아무것도 없이 시작할 때는 소프트웨어를 로드할 방법이 필요

< 툴체인 >

소스 코드를 타깃 장치에서 실행할 수 있는 실행 파일로, 컴파일러, 링커, 런타임 라이브러리를 포함하는 컴파일 도구들의 집합
나머지 세 요소(부트로더, 커널, 루트 파일시스템)을 빌드하기 위해 툴체인이 필요
어셈블리,C,C++ 언어로 작성된 코드를 컴파일 할 수 있어야 함

해당 프로세서를 위한 최적의 명령어 세트를 사용함으로써 하드웨어를 효과적으로 사용할 수 있어야 함
프로젝트 내내 바뀌지 않아야 함

<표준 GNU 툴체인 3요소>

1. **Binutils:**
   어셈블러와 링커를 포함하는 바이너리 유틸리티의 집합

2. **GCC:**
   C언어와 여러 언어를 위한 컴파일러, 공통 백엔드를 사용해 어셈블러 코드를 만들고 GNU 어셈블러로 넘김

3. **C 라이브러리:**
   POSIX 규격에 기반을 둔 표준 API, 애플리케이션에서 운영체제 커널로 연결되는 주요 인터페이스

이와 함께 커널에 직접 접근할 때 필요한 정의와 상수를 담고 있는 리눅스 커널 헤더가 필요
C라이브러리를 컴파일 하기 위해 필요
프로그램을 작성하거나 예를 들어 리눅스 프레임 버퍼 드라이버를 통해 그래픽을 보여주는 특정 리눅스 장치와 상호작용하는 라이브러리를 컴파일 할 때도 필요

<툴체인의 종류>

1. **네이티브:**
   툴체인이 만들어내는 프로그램과 같은 종류의 시스템, 때로는 실제로 같은 시스템에서 실행
2. **크로스:**
   툴체인이 타깃 기계와 다른 종류의 시스템에서 실행됨, 빠른 데스크톱 PC에서 개발한 다음 임베디드 디바이스로 로드해 테스트 가능

<CPU 아키텍처>

툴체인은 타깃 CPU의 특징에 맞게 빌드 돼야 함

1. **CPU 아키텍처:**
   ARM, MIPS 등

2. **빅 엔디언 또는 리틀 엔디언:**
   어떤 CPU는 두가지 모드로 동작할 수 있지만, 기계어 코드가 각각 다름

3. **부동소수점 지원:**
   모든 버전의 임베디드 프로세서가 하드웨어 부동 소수점 장치를 구현하지 않음
   이 경우 툴체인은 소프트웨어 부동소수점 라이브러리를 부르도록 설정할 수 있음

4. **ABI :**
   함수 호출 간에 인자를 넘기는 호출 규칙

<GNU 접두어>

1. **CPU**
   ARM, MIPS,x86_64 같은 아키텍처
   리틀 엔디언(el), 빅엔디언(eb)를 붙여 구별 (ex. mipsel, armeb)

2. **벤더**
   Buildroot, poky,unknown

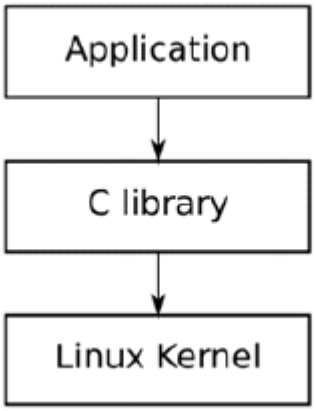
3. **커널**

4. **운영체제**
   사용자 공간 요소의 이름, gnu나 musl이 될 수 있음

$ gcc -dumpmachine 옵션을 이용하면 툴체인을 빌드할 때 쓰인 조합을 알 수 있음

Ex. $ gcc -dumpmachine
  x86_64-linux-gnu

<C라이브러리 고르기>



유닉스 운영체제의 프로그래밍 인터페이스는 C언어로 정의돼 있는데, 지금은 POSIX 표준으로 정의
C라이브러리는 애플리케이션에서 커널로 통하는 관문
자바나 파이썬 같은 다른 언어로 프로그램을 작성하더라도, 각 런타임 지원 라이브러리는 결국 다음 그림과 같이 C라이브러리를 불러야 함
커널의 서비스가 필요할 때마다 커널 시스템 호출 인터페이스를 통해 사용자 공간과 커널 공간을 전환

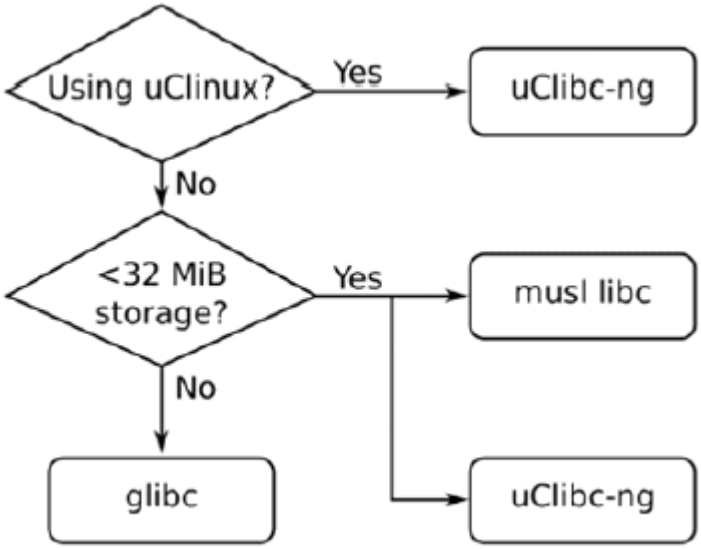
1. **glibc:**
   표준 GNU C 라이브러리
   크기가 크고 최근까지 구성 변경이 용이하지 않지만 POSIX API의 가장 완벽한 구현
   LGPL 라이선스

2. **musl libc:**
   GNU libc의 대안으로 램과 저장 공간의 크기가 제한된 시스템에 좋은선택
   MIT 라이선스

3. **uClibc-ng:**
   마이크로컨트롤러 C라이브러리
   uClibc 프로젝트에서 갈라져 나옴
   LCPL 라이선스

1. **eglibc:**
   glibc를 임베디드용으로 변경한 것
   Eglibc에는 glibc가 다루지 않는 아키텍처를 지원하는 구성 옵션이 추가
   Glibc의 코드 베이스가 다시 glibc 버전 2.20 병합되었으며, 더 이상 유지보수 되지 않음

uClinux를 쓸 때만 uClibc-ng를 쓰는 것을 권장
저장소나 램의 크기가 매우 제한되어 있다면 musl libc 선택



<툴체인 찾기>

크로스개발 툴체인을 구하는 세가지 방법

1. 미리 빌드된 툴체인 중에 필요한 사항을 만족하는 것을 찾는 방법
2. 임베디드 빌드 도구를 통해 생성된 것을 사용하는 방법
3. 직접 만드는 방법

<C라이브러리의 요소를 살펴보자>

1. **Libc:**
   Printf, open, close, read, write 등 잘 아려진 POSIX 함수들을 담고 있는 주 C 라이브러리

2. **Libm:**
   cos, exp, log 같은 수학 함수들

3. **Libpthread**:
   이름이 pthread_로 시작하는 모든 POSIX 스레드 함수들

4. **Librt:**
   공유 메모리와 비동기 I/O를 포함하는 POSIX 실시간 확장

<라이브러리와 링크하기: 정적 링크와 동적 링크>

리눅스용으로 작성하는 모든 프로그램은 C라이브러리인 libc와 링크됨
링크하고 싶은 다른 라이브러리는 -l 옵션을 통해 명시적으로 지정해야 함

라이브러리 코드를 링크하는 두가지 방법

1. **정적 링크:**
   애플리케이션의 모든 라이브러리 함수와 의존 관계가 라이브러리 아카이브로부터 추출돼 실행 파일에 복사
2. **동적 링크:**
   라이브러리 파일과 함수로의 참조가 코드 안에 만들어지지만 실제 링크는 실행 시에 동적으로 이뤄짐

정적 라이브러리

- Busybox와 약간의 스크립트 파일만으로 구성된 작은 시스템을 만든다면 busybox를 정적으로 링크해서 런타임 라이브러리 파일과 링커를 복사할 필요가 없는 편이 더 간단 (애플리케이션이 사용하는 코드만 링크하기 때문에 크기도 더 작아짐)
- 정적 링크는 또한 런타임 라이브러리를 담을 파일시스템이 준비되기 전에 프로그램을 실행 해야할 때 유용
- 명령줄에 -static을 추가하면 모든 라이브러리를 정적으로 링크
- 정적 링크는 보통 lib[name].a 인 라이브러리 아카이브로부터 코드를 복사

공유 라이브러리

- 실행 시에 링크되는 공유 오브젝트로 사용하는 것으로, 코드를 한 본(copy)만 로드하면 되기 때문에 저장 공간과 시스템 메모리를 좀 더 효율적으로 사용
- 라이브러리 파일이 갱신됐을 때 해당 라이브러리를 사용하는 모든 프로그램을 다시 링크할 필요가 없으므로 관리도 쉬움
- 공유 라이브러리용 오브젝트 코드는 위치 독립적이어서 런타임 링커가 메모리의 빈 주소에 자유롭게 위치시킬 수 있어야 함
  (이를 위해서는 gcc에 fPIC 인자를 추가하고 -shared 옵션을 이용해 링크해야 함)
- 공유 라이브러리는 링크하는 실행 파일과 분리되므로 실행 파일을 배포할 때 그 버전을 알아야 함

공유 라이브러리 버전 번호 이해하기

- 공유 라이브러리는 사용하는 프로그램과 독립적으로 갱신할 수 있음

라이브러리 업데이트의 두가지 종류

1. 하위 호환성을 유지하면서 버그를 수정하거나 새 기능을 추가하는 업데이트

2. 기존 애플리케이션과의 호환성을 깨는 업데이트

<크로스 컴파일 기술>

흔히 쓰이는 빌드 시스템

1. 툴체인이 주로 make 변수 CROSS_COMPILE로 제어되는 순수한 makefile 들
2. Autotools로 알려진 GNU 빌드 시스템
3. Cmake

단순 makefile

각각을 컴파일 할 때 make 변수 CROSS_COMEPILE에 툴체인 접두어를 넣기만 하면 됨 (대시 붙는 것 주의)
Autotools와 Cmake는 모두 makefile을 생성할 수 있음

Autotools

서로 다른 버전의 컴파일러와 라이브러리, 서로 다른 헤더파일 위치, 다른 패키지와의 의존 관계를 처리해서 패키지를 컴파일하는 서로 다른 여러 가지 시스템 사이의 차이를 완화
Autotools를 사용하는 패키지는 의존 관계를 확인하고 makefile을 만들어내는 configure라는 스크립트를 담고 있음
Configure 스크립트는 특정 기능을 켜고 끄는 선택 제공
./configure -help를 실행하면 제공하는 옵션들을 찾을 수 있음
네이티브 운영체제용 패키지를 구성, 빌드, 설치 하려면 일반적으로 다음 세 명령을 실행

```
$ ./configure
$ make
$ sudo make install
```

Autotools 패키지 컴파일 시 관여하는 서로 다른 세가지 기계를 이해할 수 있음

1. 빌드: 패키지를 빌드하는 컴퓨터, 기본 설정은 현재 기계
2. 호스트: 프로그램이 실행될 컴퓨터, 네이티브 컴파일의 경우, 이 란은 비어두고 빌드와 같은 컴퓨터로 기본 설정 됨
3. 타깃: 프로그램이 생성하는 코드가 실행될 컴퓨터, 크로스 컴파일러를 빌드할 때 이를 설정

Cmake

소프트웨어를 빌드하기 위해 기본 플랫폼의 기본 도구에서 의존한다는 점에서 메타 빌드 시스템에 가까움
리눅스에서 네이티브 빌드 도구는 GNU make이므로 Cmake는 디폴트로 빌드에 사용할 makefile을 생성

Cmake에 자주 등장하는 핵심 용어

1. 타깃: 라이브러리나 실행 파일 같은 소프트웨어 구성 요소
2. 속성: 타깃을 빌드하는 데 필요한 소스 파일, 컴파일러 옵션, 링크된 라이브러리를 포함
3. 패키지: CMakeLists.txt 자체 내에 정의된 것처럼 빌드를 위한 외부 타깃을 구성하는 Cmake 파일