허교수의 ARM Mbed 프로그래밍 입문

허경용의 ARM Mbed 프로그래밍 입문

초판 1쇄 발행 2019년 8월 8일

지은이 허경용
펴낸이 장성두
펴낸곳 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 회동길 159 3층 3-B호
전화 070-8201-9010 / **팩스** 02-6280-0405
홈페이지 www.jpub.kr / **원고투고** jeipub@gmail.com
독자문의 readers.jpub@gmail.com / **교재문의** jeipubmarketer@gmail.com

편집부 이종무, 황혜나, 최병찬, 이 슬, 이주원 / **소통·기획팀** 민지환, 송찬수 / **회계팀** 김유미
교정·교열 김은미 / **진행** 이 슬 / **본문디자인** 성은경 / **표지디자인** 미디어픽스
용지 신승지류유통 / **인쇄** 해외정판사 / **제본** 광우제책사

ISBN 979-11-88621-65-1 (93000)
값 27,000원

허교수의 ARM Mbed 프로그래밍 입문

허경용 지음

Jpub 제이펍

※ 드리는 말씀

- 이 책에 기재된 내용을 기반으로 한 운용 결과에 대해 저자, 하드웨어/소프트웨어 개발자 및 제공자, 제이펍 출판사는 일체의 책임을 지지 않으므로 양해 바랍니다.
- 이 책에 등장하는 회사명, 제품명은 일반적으로 각 회사의 등록 상표(또는 상표)이며, 본문 중에는 ™, ©, ® 마크 등을 생략하고 있습니다.
- 이 책에서 사용하고 있는 제품은 독자의 학습 시점에 따라 책의 내용과 다를 수 있습니다.
- 책의 내용과 관련된 문의사항은 지은이 혹은 출판사로 연락주시기 바랍니다.
  - 지은이: https://cafe.naver.com/sketchurimagination
  - 출판사: readers.jpub@gmail.com

차 례

머리말

Mbed는 ARM의 Cortex-M 마이크로컨트롤러를 위한 개발 환경으로, 32bit 마이크로컨트롤러의 복잡하고 어려운 개발 과정을 빠르고 간단하게 만들고자 시작되었다. Mbed와 흔히 비교되는 것이 아두이노다. Mbed와 아두이노는 쉽고 빠르게 원하는 것을 구현하자는 목적을 공유할 뿐만 아니라, 쉽고 빠른 개발 환경이 가지는 단점까지도 비슷하다. 하지만 아두이노가 대표적인 마이크로컨트롤러 프로젝트의 하나로 자리매김했다면, Mbed는 아는 사람이 많지 않을 정도로 이름을 알리는 데 실패했다. 이러한 차이는 어디에서 오는 것일까?

여러 이유가 있겠지만 아두이노가 사용자 입장에서 시작된 프로젝트라면, Mbed는 개발자 입장에서 시작된 프로젝트라는 점을 무시할 수 없다. 물론, Mbed가 ARM의 개발자들에 의해 시작되었다는 점을 장점으로 볼 수도 있지만, 사용자의 요구가 아닌 개발자의 요구를 해결하기 위해 만들어졌다는 점 때문에 일부 마니아층만을 위한 전유물로 여겨진 것이 사실이다. 또 한 가지 문제점이라면 Mbed의 대상이 모호하다는 점이다. 아두이노는 초보자와 비전공자를 대상으로 시작되었고 다양한 분야에 적용할 수 있어 시너지 효과를 얻을 수 있었다. 하지만 Mbed는 비전공자를 대상으로 하기에는 조금 어려운 것이 사실이다. C 언어의 main 함수를 그대로 가지고 있다는 점도 비전공자에게는 간단하지 않은 문제가 될 수 있다. 따라서 Mbed는 그저 재미있는 개발 환경의 하나로 인식되거나 일부 사람만 사용하는 플랫폼으로만 명맥을 이어왔다. 단적인 예로 Mbed를 다루는 책은 세계적으로도 10권을 넘지 않는다.

이러한 단점에도 불구하고 Mbed를 소개하는 책을 쓰고자 했던 가장 큰 이유는 'Mbed가 훌륭한 개발 환경'이기 때문이다. 아두이노로 할 수 있는 것이라면 Mbed로도 모두 할 수 있다. 그것도 아두이노와 비교해서 더 쉽고 간단하게 할 수 있다. 더 중요한 것은 아두이노보다 Mbed로 할 수 있는 것이 더 많고 앞으로도 점점 늘 것이라는 점이다. 아마도 ARM의 마이크로프로세서가 시장을 선도하는 한 Mbed의 영역은 계속 확대될 것이다. 스마트폰을 위한 마이

크로프로세서 대부분이 ARM의 설계를 사용하고 있다는 것은 이미 잘 알려진 사실이다. 또한, 사물인터넷의 보급에 힘입어 사물을 구현하기 위한 마이크로컨트롤러의 수요도 늘고 있으며, 그 중심에는 ARM의 Cortex-M 마이크로컨트롤러가 있다. 이러한 변화에 따라 ARM은 Mbed를 사물인터넷을 위한 플랫폼으로 확장하고 있다. 그저 재미있는 플랫폼에서 벗어나 전문가도 사용할 수 있는 실용적인 플랫폼으로 변신을 도모하고 있다. 또한, Mbed의 특징 중 하나인 온라인 개발 환경에서 벗어나 전문가가 사용하기에도 충분한 오프라인 개발 환경까지도 개발되고 있어 ARM 생태계에 중요한 부분을 차지하지 않을까 싶다.

이처럼 Mbed는 빠르게 진화하고 있다. 아두이노를 대체할 수 있는 고성능 마이크로컨트롤러 개발 환경은 물론, 아두이노로는 상상할 수 없었던 영역까지 발을 내디디고 있는 점이 앞으로 Mbed를 주목해야 하는 이유다. 여기에 ARM의 지원이 더해진다면 무엇이 더 가능해질지 상상하기 어렵다는 점에서 기대를 더한다. Mbed가 ARM의 개발자들에 의해 시작되었다는 것이 오롯이 장점이 될 날이 머지않았다.

이 책은 마이크로컨트롤러 개발 환경으로서 Mbed를 소개하고, Mbed에 익숙해지도록 도움으로써 본격적인 개발에 사용할 수 있는 발판을 마련하는 것을 목표로 한다. 이 책을 통해 Cortex-M 마이크로컨트롤러를 사용하는 시스템을 구현하는 환경에 익숙해지고, 마이크로컨트롤러를 넘어 사물인터넷이라는 연결된 세상으로 나가는 길을 발견하기를 바란다. 또한, Mbed의 미래를 즐겁게 기다리는 데 함께한다면 더 바랄 것이 없겠다.

허경용

베타리더 후기

곽상영(NeuRobo)

아두이노를 졸업하고 보다 고성능의 MCU를 접하고자 한다면 Mbed가 답이 될 수도 있을 것입니다. 아두이노로 마이크로컨트롤러의 첫걸음을 뗀 분이라면 이 책이 다양한 스펙트럼의 개발 환경을 접하는 데 좋은 계기가 될 것 같습니다. 기술적으로 깊이 있는 내용을 다룬다는 것이 장점이지만, 좀 더 다양한 주변장치를 다루었으면 하는 아쉬움이 있습니다.

김종욱(네이버)

ARM Cortex 마이크로컨트롤러에서 사용되는 다양한 모듈의 사용 방법과 팁을 총정리한 바이블입니다. 실습할 때 옆에 두고 참고하기에 더없이 좋은 책입니다.

남원우(창원대학교)

이 책 덕분에 임베디드 입문자에게 막연하게 느껴지던 32bit MCU 세상에 한발 다가갈 수 있었습니다. 이 책은 아두이노가 아닌 새로운 환경에서 임베디드 개발을 해보고 싶은 사람에게 Mbed와 Cortex-M 세상을 소개합니다. 게다가 군더더기 없이 깔끔한 구성과 적절한 난이도로 혼자 학습하기에도 무리가 없습니다. 아두이노를 벗어나 새로운 임베디드 환경을 경험하고 싶은 입문자가 있다면 이 책을 추천합니다.

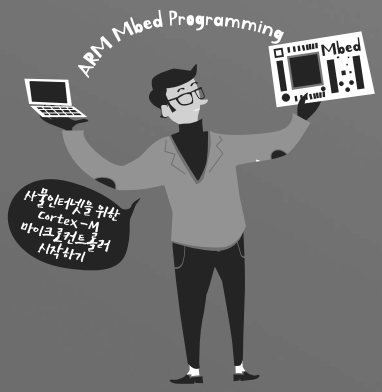
이현수(무스마 기술연구소)

대학 시절 임베디드 시스템 설계 과목을 수강하며 배운 내용이 새록새록 떠올라 반가웠습니다. 문장이 깔끔하고 읽기가 편해서 내용이 더 쉽게 들어오는 것 같습니다. 각 장의 내용이 주제별로 간결하게 나누어져 있어 한 차례 모두 읽은 후에는 참고 자료로 찾아보기에도 손색이 없을 것 같습니다.

PART

I

Mbed 소개



CHAPTER 01

Mbed란 무엇인가?

1.1 ARM 프로세서

Mbed는 ARM에서 설계한 'Cortex-M 기반의 마이크로컨트롤러를 위한 사물인터넷 플랫폼'을 가리킨다. 사물인터넷 플랫폼으로 동작하기 위해 Mbed는 Cortex-M 기반 마이크로컨트롤러를 사용하여 사물인터넷 환경에서 동작하는 장치를 제작하고 이를 통해 서비스를 지원해주는 일련의 도구를 제공하고 있다. Mbed는 2005년 ARM의 연구원인 사이먼 포드(Simon Ford)와 크리스 스타일스(Chris Styles)가 학생들의 아이디어를 쉽게 구현해보기 위한 목적으로 개발하였다. 이후 32bit 마이크로컨트롤러에 대한 수요 증가에 따라 Mbed는 Cortex-M 마이크로컨트롤러의 가능성을 확인하고자 하는 기업의 요구를 만족시키고, 사물인터넷의 확산에 대응할 수 있도록 사물인터넷을 위한 플랫폼으로 확장되었다.

Mbed를 이야기하기 전에 먼저 Mbed 보드의 핵심이라 할 수 있는 ARM의 프로세서에 대해 알아보자. ARM 프로세서는 1985년 영국의 가정용 컴퓨터 제조 회사인 에이콘 컴퓨터스(Acorn Computers)에서 개발한 세계 최초의 상업용 RISC(Reduced Instruction Set Computer) 프로세서다. 에이콘은 자사에서 생산하는 컴퓨터에 탑재할 목적으로 프로세서를 개발하였다. 에이콘에서 개발한 프로세서는 ARM(Acorn RISC Machine)이라고 불렸으며, 이후 에이콘은 애플(Apple), VLSI 테크놀로지(VLSI Technology)와 함께 1990년에 ARM(Advanced RISC Machine)이란 회사를 설립하여 낮은 가격에 저전력, 고효율의 32bit 프로세스를 설계하는 데 집중하였다. 이러한 설계 방향은 1990년대 다른 경쟁 업체들이 고성능 프로세서 개발을 목표로 했다는 점과는 차이가 있다. 하지만 모바일 기기에 대한 수요가 증가하면서 저전력 프로세서에 대한 요구

역시 증가했다. 현재 ARM이 전 세계 스마트폰에 사용되는 프로세서의 95% 이상을 점유하고 있다는 점은 ARM의 전략이 적중했다는 증거라 할 수 있다. 이외에도 ARM 프로세서는 낮은 가격에 힘입어 전통적인 8bit 및 16bit 프로세서 시장으로도 영역을 확대하고 있으며 다양한 전자제품에서 사용되고 있다.

RISC vs. CISC

ARM 프로세서는 대표적인 RISC 프로세서 중 하나다. 반면 데스크톱 컴퓨터에서 흔히 사용되는 인텔의 프로세서는 대표적인 CISC(Complex Instruction Set Computer) 프로세서 중 하나다. CISC와 RISC의 가장 큰 차이는 CPU에서 지원하는 명령어 개수에 있다. CPU에서 지원하는 명령어는 어셈블리어(assembly language) 수준에서의 명령어로서, 전용 하드웨어로 처리되는 명령어를 말한다. 하드웨어로 처리되는 명령어에는 사칙연산, 논리연산, 분기, 메모리 읽기와 쓰기 등이 포함된다.

CPU의 발전과 더불어 CPU에서 처리할 수 있는 명령어의 개수는 점차 증가하였다. 명령어 개수의 증가는 명령어 처리를 위한 하드웨어의 추가로 이어지고 이에 따라 CPU는 점차 복잡해졌다. 게다가 명령어 추가 과정에서 하위 호환성을 유지하기 위해서는 이전의 명령어들을 그대로 유지해야 하므로, 새롭게 추가되는 복잡한 명령어와 이전의 간단한 명령어의 처리 시간이 달라지는 경우가 발생하였다. 이처럼 하드웨어가 복잡해짐에 따라 CPU의 기능을 개선하는 일이 쉽지 않을뿐더러 성능 향상에도 걸림돌로 작용하게 되었다. 이러한 문제를 해결하고자 하는 시도 중 하나가 CPU에서 지원하는 명령어의 개수를 줄이는 것이었는데, CISC 구조에서 사용하는 명령어 중 자주 사용되는 간단한 명령어만을 모아놓은 것이 바로 RISC 구조다. RISC 구조는 IBM 연구소의 존 코크(John Cocke)가 'CPU에서 지원하는 명령어 중 20% 정도가 프로그램에서 80% 이상의 일을 처리한다'는 사실을 증명함으로써 1970년대에 알려지기 시작했다.

명령어 개수를 줄이면 CPU 구조가 단순해지며 명령어 처리 속도를 높일 수 있다. 복잡한 명령어 처리를 위해서는 많은 수의 간단한 명령어를 실행해야 하지만, 80%의 일을 더 빨리 처리한다면 나머지 20% 처리에 많은 시간이 걸린다고 해도 전체적으로는 비슷하거나 더 빠른 속도로 명령어를 처리할 수 있다. 한 가지 문제점은 CISC 구조에서는 하나의 명령어로 가능한 작업을 RISC 구조에서는 여러 개의 명령어로 처리해야 하는 경우가 있으므로, 복잡한 명령을 간단한 명령의 집합으로 나누는 작업이 필요하다는 점이다. RISC 구조에서는 이러한 명령어 분해 작업을 소프트웨어적으로, 즉 기계어 파일을 만들어내는 컴파일러가 처리하도록 하고 있다. CISC 구조에서 복잡한 명령의 처리를 하드웨어가 담당한다면, RISC 구조는 소프트웨어가 담당해야 하는 비중을 보다 높인 차이가 있다.

RISC 구조의 대표적인 특징 중 하나는 메모리를 읽고 쓰는 명령이 별도로 존재한다는 점이다. CISC 구조에서는 ① 메모리에서 피연산자를 읽어와 ② 연산을 수행하고 ③ 그 결과를 메모리에 저장하는 과정을 하나의 어셈블리어 명령으로 처리할 수 있다. 반면 RISC 구조에서는 각 단계를 위한 어셈블리어 명령이 필요하다.

표 1-1은 RISC 구조와 CISC 구조를 비교한 것이다. 하지만 최근에 출시되는 CPU는 순수하게 CISC 구조나 RISC 구조만을 사용하는 경우는 거의 없으며 서로의 장점을 채택하고 있으므로 CISC 구조와 RISC 구조의 구분이 모호해지고 있다는 점도 잊지 말아야 한다.

표 1-1 CISC 구조와 RISC 구조의 특징

	CISC	RISC
명령어 개수	많음	적음
프로그램 크기	작음	큼
하드웨어 복잡도	높음	낮음
소프트웨어(컴파일러) 복잡도	낮음	높음
명령어 실행 시간	가변	고정
전력 소모	많음	적음
호환성	높음	낮음
대표적인 CPU 제조/설계사	인텔(Intel)	ARM

ARM이 성공할 수 있었던 다른 이유 중 하나는 개발, 생산, 판매의 전 과정을 하나의 회사에서 진행하던 기존 틀을 벗어나 프로세서의 설계만 하고 설계한 프로세서에 대한 라이선스를 지적재산(IP, Intellectual Property) 형태로 판매한다는 점이다. ARM에서 제공하는 IP는 일종의 설계도로 반도체 회사들은 여기에 필요한 주변장치를 추가해 SoC(System on a Chip) 형태의 반도체 칩으로 제조하여 판매한다. 사물인터넷 및 4차 산업혁명의 발전과 함께 하나의 칩에 시스템의 기능을 집적하는 SoC에 대한 요구는 증가하고 있으며, 이에 따라 SoC를 제작하는 회사들은 그에 적합한 코어를 찾고 있다. 모듈 형식으로 설계된 ARM 프로세서는 기업의 필요에 따라 다양한 방식으로 조합하는 것이 가능하여 기업의 다양한 요구를 충족시켜줄 수 있으므로 ARM 프로세서에 대한 수요는 계속 증가하고 있다. 퀄컴, 삼성전자, 애플, 텍사스 인스트루먼츠(Texas Instruments), 엔비디아(NVIDIA), IBM, AMD 등 대부분의 칩 생산 기업들은 ARM의 IP를 구입하여 칩을 생산하고 있다.

ARM 프로세서를 설명하기 위해서는 몇 가지 단어에 대한 정의가 필요한데, ARM 아키텍처, ARM 코어, ARM 프로세서 등이 대표적이다. 이 단어들이 가리키는 것들은 서로 연관되어 분리하기가 쉽지 않으므로 관련 문서를 읽을 때 이해를 어렵게 하는 원인 중 하나가 된다. 아래는 이 책에서 사용하는 정의들인

데, 일반적으로 사용되고 있는 정의를 따르려고 했지만 다른 문서에서는 일부 다른 의미로 사용될 수도 있다는 것에 주의해야 한다.

- **ARM ISA(Instruction Set Architecture) 또는 ARM 아키텍처**: 프로세서에서 수행할 수 있는 명령어의 종류 및 이에 대한 동작을 정의한 사양(specification)에 해당한다.
- **ARM 코어**: ARM 아키텍처에 따라 구현한 프로세서의 핵심 부분으로 중앙처리장치(CPU, Central Processing Unit)에 해당한다. ARM 아키텍처는 모듈 형식으로 정의되므로 ARM 코어에서는 필요에 따라 ARM 아키텍처를 선택적으로 구현한다.
- **ARM 코어 패밀리**: 유사한 기능을 가지는 코어들을 통칭하기 위해 사용된다. Cortex-M0, Cortex-M1, Cortex-M3 등은 흔히 '코어'라고 이야기하며, 이들 전체를 가리키는 Cortex-M은 '코어 패밀리'라고 통칭하여 구별한다. 하지만 Cortex-M의 공통적인 특징을 강조하기 위해 Cortex-M을 'Cortex-M 코어'라고 지칭하는 경우를 흔히 볼 수 있다. 또한 'Cortex-M 코어 시리즈'라는 이름으로 불리기도 한다.
- **ARM 프로세서**: ARM 코어에 메모리, 입출력 인터페이스 등을 추가하여 만든 설계 또는 이를 SoC 형태로 구현한 칩을 가리킨다. 칩의 경우 제작사에서 별도의 이름을 정하는 경우가 대부분이며, 이 책에서 사용하는 프로세서의 경우 칩을 생산하는 STMicroelectronics에서 정한 STM32F103이라는 이름이 붙어 있다. Cortex-M3는 코어를 가리키지만, 'Cortex-M3 프로세서'와 같이 코어를 사용하여 만든 프로세서를 가리키기 위해 사용되는 경우도 흔히 볼 수 있다.

이 책에서는 STMicroelectronics에서 제작한 STM32F103RBT6 마이크로컨트롤러(또는 ARM 프로세서)를 사용한다. STM32F103은 Cortex-M3 코어를 사용하고 있으며, Cortex-M3 코어는 ARM 아키텍처의 하나인 ARMv7 아키텍처의 세부 아키텍처인 ARMv7-M 아키텍처를 바탕으로 한다.

ARM 프로세서의 핵심은 기능을 정의하는 'ARM 아키텍처', 그리고 정의된 기능을 용도에 맞게 구현한 'ARM 코어'에 있다. ARM에서 2003년 이전에 개발한 코어는 'ARM[n](n은 정수)' 형식의 이름을 가지고 있으며, 이를 '클래식 코어(classic core)'라고 한다. 2004년 이후로는 'Cortex-[X](X는 알파벳)' 형식으로 바뀌면서 X에는 용도에 맞게 A, R, M 중 하나의 알파벳을 사용한다.

- **Cortex-A(Application)**: 스마트폰, 태블릿, 서버 등을 위한 하이엔드 코어로서 ARM의 주력 제품군이다. 1GHz 이상의 높은 클록에서 실행되며, 리눅스와 안드로이드, 마이크로소프트 윈도우 등의 운영체제를 사용할 수 있다.
- **Cortex-R(Real-time)**: 실시간 환경에 적합한 특징을 가지는 코어로서 Cortex-A 코어보다는 성능이 낮다. 실시간 시스템을 위해 Cortex-R 코어는 주로 실시간 운영체제(RTOS, Real Time Operating System)와 함께 사용되며, 자동차 제어 시스템, 센서 네트워크, 대용량 저장 장치 컨트롤러, 네트워크 연결 장치 등에서 사용한다.

- **Cortex-M(Microcontroller)**: 이름 그대로 마이크로컨트롤러를 위한 코어다. 2010년까지 Cortex-M 시리즈는 가격이 싼 8bit나 16bit MCU(Microcontroller Unit)에 매출과 생산 규모에서 뒤처져 있었다. 이후 32bit MCU의 가격 하락에 따라 32bit MCU의 판매량이 증가하면서 32bit MCU 코어가 주력인 ARM의 Cortex-M 기반 마이크로컨트롤러는 MCU 시장에서 주도적 위치를 차지하고 있다.

표 1-2는 ARM에서 발표한 Cortex 코어를 연도별로 나타낸 것으로, 2004년 이전에 발표된 클래식 코어는 나타내지 않았다. 2003년까지 발표된 ARM 코어는 발표 순서에 따라 ARM1(1985년)에서 ARM11(2002년)까지 이름이 정해져 있다.

표 1-2 Cortex 코어

연도	Cortex 코어 패밀리			
	Cortex-M	Cortex-R	Cortex-A(32bit)	Cortex-A(64bit)
2004	Cortex-M3			
2005			Cortex-A8	
2006				
2007	Cortex-M1		Cortex-R9	
2008				
2009	Cortex-M0		Cortex-R5	
2010	Cortex-M4		Cortex-R15	
2011		Cortex-R4 Cortex-R5 Cortex-R7	Cortex-R7	
2012	Cortex-M0+			Cortex-R53 Cortex-R57
2013			Cortex-R12	
2014	Cortex-M7		Cortex-R17	
2015				Cortex-R35 Cortex-R72
2016	Cortex-M23 Cortex-M33	Cortex-R8 Cortex-R52	Cortex-R32	Cortex-R73
2017				Cortex-R55 Cortex-R75
2018	Cortex-M35P			Cortex-R76

Cortex 코어 중에서도 가장 먼저 발표된 것은 마이크로컨트롤러를 위한 Cortex-M3 코어로, 이 책에서 다루는 STM32F103에 사용된 코어이기도 하다. STM32F103 이외에도 STMicroelectronics에서는 Cortex-M3 코어를 사용한 많은 종류의 마이크로컨트롤러를 제작하여 판매하고 있으며, 그림 1-1은 그중 일부를 나타낸 것이다.

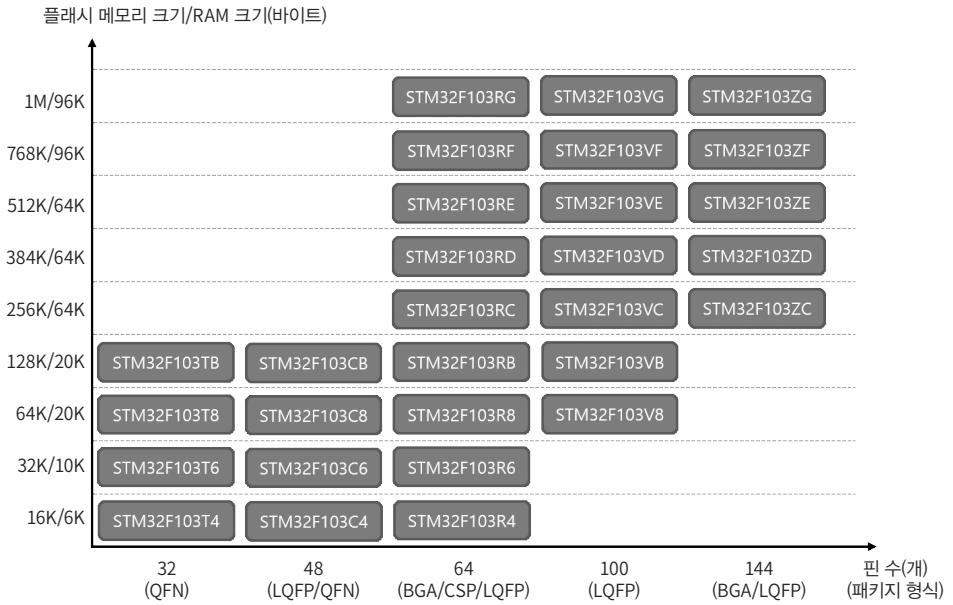


그림 1-1 **STM32F103 시리즈 마이크로컨트롤러**

그림 1-1의 마이크로컨트롤러들은 모두 Cortex-M3를 기반으로 하고 있지만, 메모리 크기와 패키지의 핀 수에는 차이가 있다. 즉, 같은 코어라도 필요에 따라 다른 프로세서(또는 마이크로컨트롤러)를 선택하여 사용할 수 있다. 이처럼 STM32F103 시리즈 마이크로컨트롤러는 핵심이 되는 Cortex-M3 코어에 메모리와 입출력장치 등의 주변장치를 추가하여 다양한 프로세서를 만들 수 있다.

ARM의 코어는 'ARM[n]' 또는 'Cortex-[X]' 형식의 이름을 가진다. 이와 혼동하기 쉬운 것이 바로 'ARMvn(n은 정수)'으로 표시되는 아키텍처다. ARM 아키텍처는 명령어, 레지스터 구조, 메모리 구조 등 프로세서의 기본 구조와 동작 원리를 정의한 것으로, 이 아키텍처에 따라 구현한 프로세서의 핵심 부분을 코어라고 하며, 여기에 메모리, 메모리 관리 장치, 인터페이스 장치 등의 주변장치를 추가한 것이 프로세서에 해당한다. 가장 최근에 발표된 아키텍처는 v8이다. 현재 판매되고 있는 Cortex 코어 대부분은 ARMv7을 사용하고 일부 최신 코어가

ARMv8을 사용한다. 아키텍처 역시 세부 아키텍처로 나눌 수 있으며, Cortex-A는 ARMv7-A, Cortex-R은 ARMv7-R, Cortex-M은 ARMv7-M 아키텍처를 사용하는 경우가 대부분이다.

1.2 Cortex-M

이 책에서 다루는 STMicroelectronics의 STM32F103RBT6 마이크로컨트롤러는 Cortex-M3 코어에 주변장치를 추가하여 만들어진 마이크로컨트롤러다. Cortex-M 시리즈 코어는 마이크로컨트롤러를 위한 코어로 지금까지 9개가 발표되었다. 표 1-3은 Cortex-M 시리즈에 속하는 코어들을 비교한 것이다.

표 1-3 **Cortex-M 코어**

ARM 코어	발표 연도	ARM 아키텍처	비고
Cortex-M3	2004	ARMv7-M	최초의 Cortex 코어
Cortex-M1	2007	ARMv6-M	FPGA 전용
Cortex-M0	2009	ARMv6-M	
Cortex-M4	2010	ARMv7E-M	
Cortex-M0+	2012	ARMv6-M	
Cortex-M7	2014	ARMv7E-M	가장 높은 성능
Cortex-M23	2016	ARMv8-M	TrustZone 보안 기능 지원
Cortex-M33	2016	ARMv8-M	
Cortex-M35P	2018	ARMv8-M	

Cortex-M 시리즈 중 가장 먼저 발표된 코어는 2004년 소개된 Cortex-M3이며, Cortex-M3보다 성능은 낮지만 저전력을 강조한 Cortex-M0 코어가 2009년 소개되었다. Cortex-M3에 실수 연산을 비롯한 몇 가지 기능을 추가한 Cortex-M4는 2010년 발표되었고, 2012년에는 Cortex-M0의 성능을 개선한 Cortex-M0+가 발표되었다. 2014년에는 Cortex-M 시리즈 중 성능이 가장 뛰어난 Cortex-M7이 발표되었으며, 2016년 이후 새로운 아키텍처인 ARMv8을 바탕으로 보안 기능을 강화한 Cortex-M23, Cortex-M33, Cortex-M35P 등이 발표되었다. 이외에도 FPGA(Field Programmable Gate Array) 전용인 Cortex-M1이 있다. Cortex-M 코어는 다음과 같이 3개의 그룹으로 나누어볼 수 있다.[1]

1 Cortex-M23, Cortex-M33, Cortex-M35P 코어는 한두 개 기업에서 이들 코어를 사용한 프로세서의 양산을 준비하고 있으나 아직은 널리 사용되고 있지는 않다.

- Cortex-M0, Cortex-M0+는 낮은 가격과 저전력이 요구되는 애플리케이션에 적합하다.
- Cortex-M3, Cortex-M4는 성능과 전력 효율 모두를 만족시킬 수 있는 범용 애플리케이션에 적합하며, 가장 많이 사용되는 코어다.
- Cortex-M7은 Cortex-M 시리즈 중 가장 높은 성능을 보이는 코어로, 고성능 애플리케이션에 적합하다.

표 1-4는 Cortex-M 코어의 성능을 비교한 것으로, 필요한 연산 능력에 따라 선택하여 사용하면 된다.

표 1-4 Cortex-M 코어의 MHz당 성능

	CoreMark/MHz[2]	DMIPS/MHz[3]
Cortex-M0	2.33	0.87~1.27
Cortex-M0+	2.46	0.95~1.36
Cortex-M3	3.32	1.25~1.89
Cortex-M4	3.40	1.25~1.95
Cortex-M7	5.01	2.14~3.23

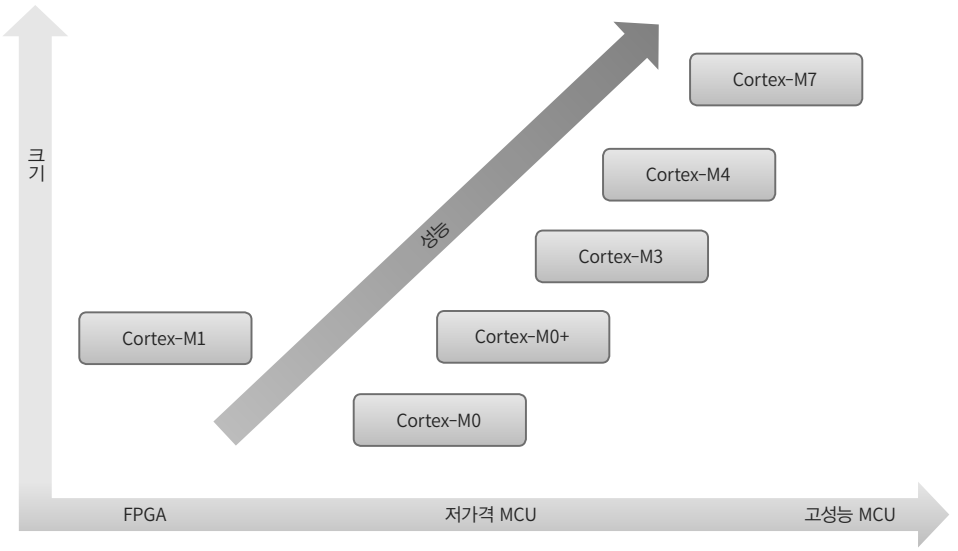


그림 1-2 Cortex-M 코어 성능 비교

2 코어마크(CoreMark)는 EEMBC(Embedded Microprocessor Benchmark Consortium)에서 제공하는 벤치마크로 이전에 많이 사용되던 드라이스톤(Dhrystone)을 대체하는 산업계 표준이다.

3 DMIPS(Dhrystone Million Instructions Per Second)는 코어마크 이전에 주로 사용되던 벤치마크다.

Cortex-M 시리즈 중 가장 많이 사용되는 코어는 가장 오랜 역사를 가진 Cortex-M3 코어로, 이 책에서도 Cortex-M3 코어를 사용한 STM32F103RBT6 마이크로컨트롤러를 사용한다. 2004년 Cortex-M3가 처음 소개된 이후 2006년 텍사스 인스트루먼츠에서 처음으로 Cortex-M3 코어 기반의 마이크로컨트롤러를 발표하였으며, 2007년 STMicroelectronics가 뒤를 이어 지금까지 STM32 시리즈로 다양한 Cortex-M 기반의 마이크로컨트롤러를 생산하고 있다.

1.3 Mbed의 등장

Mbed는 Cortex-M 기반 마이크로컨트롤러를 위한 프로토타이핑 도구로 2005년 시작되었다. Mbed가 등장한 이유는 한마디로 ARM 프로세서를 다루기가 어렵기 때문이다. ARM을 어렵게 만드는 이유 중 하나는 ARM 프로세서가 32bit CPU를 포함하고 있다는 점이다. 학교에서 흔히 마이크로컨트롤러 학습용으로 사용하는 AVR 시리즈 마이크로컨트롤러가 8bit라는 점을 감안하면 ARM이 다루기 어렵다는 생각을 가지는 것은 당연하며, 어느 정도 사실이기도 하다.

ARM을 어렵게 만드는 또 다른 이유는 ARM이 IP만을 판매하고 직접 칩을 생산하지 않기 때문이다. AVR의 경우 마이크로칩(Microchip)에서 설계하고 생산한다. 하지만 ARM 프로세서는 ARM에서 IP를 구입한 많은 회사에서 생산하고 있으므로, 비록 그 핵심이 되는 부분은 같다고 할지라도 회사별로 차이가 존재한다. 여기에 ARM의 개발을 복잡하게 만드는 또 다른 요인은 다양한 개발 환경이 존재한다는 점이다. AVR의 경우도 다양한 개발 환경을 사용할 수 있지만, 기본적으로 동일한 하드웨어를 사용하므로 개발 환경에 큰 차이는 없다. ARM의 경우에는 하나의 개발 환경 내에서도 서로 다른 회사에서 제조한 다양한 하드웨어를 지원해야 하므로 선택해야 하는 가짓수는 기하급수적으로 증가할 수밖에 없다.

AVR 마이크로컨트롤러를 사용하기 위해서는 개발 환경만을 선택하면 되고, AVR의 경우 마이크로칩에서 Atmel Studio라는 강력한 개발 도구를 무료로 제공하고 있으므로 개발 환경의 선택이 어렵다면 Atmel Studio를 선택하면 된다. 하지만 ARM을 사용하기 위해서는 어느 회사에서 만든 어떤 프로세서를 어떤 개발 환경을 사용하여 개발할 것인지 선택해야 한다. 게다가 ARM을 위한 대표적인 개발 환경인 IAR, Keil 등은 모두 유료다. 이클립스(Eclipse)와 같이 무료로 사용할 수 있는 개발 환경도 존재하지만, 이클립스의 경우 다양한 개발 환경을 지원하는 범용 도구이므로 마이크로컨트롤러를 위한 전용 개발 환경으로 보기는 어렵다.

ARM을 위한 개발 환경을 선택하기는 쉽지 않으므로 어느 회사에서 제작한 마이크로컨트롤

러인가에 따라 개발 환경을 선택하는 경우를 흔히 볼 수 있다. 만약 하나의 개발 환경에서 여러 회사에서 제작한 Cortex-M 기반 마이크로컨트롤러를 위한 프로그램을 개발할 수 있다면 어려움이 줄어들지 않을까? 이러한 접근법을 사용하는 대표적인 개발 환경 중 하나가 아두이노(Arduino)다. 아두이노 보드 역시 다양한 마이크로컨트롤러를 사용하여 만들어진다. Mbed에서 사용되는 마이크로컨트롤러는 모두 Cortex-M 기반이라는 공통점이 있다. 하지만 아두이노 보드에 사용되는 마이크로컨트롤러에는 AVR은 물론 Cortex-M까지 포함되어 있으므로 그 종류가 더 다양하다. 하지만 아두이노는 모든 아두이노 보드를 동일한 환경에서 사용할 수 있는 개발 환경을 제공해줌으로써 ARM과 달리 '쉽다'는 인상을 심어주어 아두이노 보급의 일등 공신이 되었다.

Mbed 역시 아두이노와 마찬가지로 모든 Mbed 보드를 위한 프로그램을 동일한 개발 환경에서 작성할 수 있도록 설계되었다. Mbed는 더 나아가 개발 환경을 온라인을 통해 제공하고 있다. 온라인 개발 환경이란 웹상에서 프로그램을 작성하고 이를 컴파일하여 실행 파일을 생성할 수 있는 환경을 말하는데, Mbed 보드를 USB를 통해 컴퓨터에 연결한 후 Mbed 홈페이지에 접속하는 것만으로 프로그램 작성을 위한 준비는 끝난다. 온라인에서 작성한 프로그램은 온라인 컴파일러를 통해 실행 파일로 만들어지고, 실행 파일을 다운로드한 후 USB 메모리에 파일을 복사하는 것과 동일하게 Mbed 보드에 복사해 넣는 것만으로 프로그램이 설치되고 실행 가능한 상태가 된다. 물론 이것이 가능한 이유는 복사해 넣는 방식을 지원하기 위한 전용 하드웨어가 Mbed 보드에 포함되어 있기 때문이다. 사용자는 그저 Mbed 보드의 설계 기준을 만족하는 Mbed 지원 보드를 사용하기만 하면 된다.

이처럼 '쉬운 프로토타입 개발 환경'에서 출발한 Mbed는 사물인터넷의 보급에 따라 쉽게 사물을 제작할 수 있고 제작된 사물을 사용하여 서비스를 제공할 수 있는 'Mbed IoT 디바이스 플랫폼(또는 간단히 'Mbed 플랫폼')'으로 확장되었다. Mbed 플랫폼은 '사물인터넷을 위한 디바이스와 애플리케이션 개발에 필요한 모든 것'을 포함하고 있으며, 다음과 같은 요소들로 구성된다.

① **개발 도구와 Mbed 보드**: Cortex-M 기반 마이크로컨트롤러를 사용하여 만들어진 100개 이상의 개발 보드들이 Mbed를 지원하고 있다. Mbed를 지원한다는 것은 온라인 컴파일러를 사용하여 프로그램 개발이 가능하고 복사해 넣기 방식의 프로그램 설치가 가능하다는 의미다. 또한 ARM에서 무료로 제공하는 개발 도구는 웹 기반 개발은 물론 오프라인 개발도 지원한다.

② **Mbed OS**: 오픈소스 기반의 플랫폼 운영체제로 설계된 Mbed OS는 Cortex-M 기반 마이크로컨트롤러의 기본적인 기능뿐만 아니라 보안 및 네트워크 관련 기능 등 사물인터넷을 위한 디바이스 개발에 필요한 기능들을 포함하고 있다. 또한 Mbed OS는 이러한 기능들을 추상화된 API(Application Programming Interface)를 통해 제공하고 있으므로 Mbed를 지원하는 보드에서 실행 가능한 프로그램을 공통 API를 사용하여 쉽게 개발할 수 있도록 해준다.

③ **Mbed 클라우드(cloud)**: Mbed 클라우드는 다양한 네트워크 환경에서 안전하고 확장 가능한 사물인터넷 디바이스 관리 기능을 제공한다. Mbed 클라우드에서는 Mbed OS를 사용하여 만들어진 디바이스는 물론 Mbed OS를 사용하지 않고 만들어진 디바이스 역시 사용할 수 있다.

④ **Mbed TLS(Transport Layer Security)**: Mbed TLS는 적은 노력으로 암호화와 SSL(Secure Sockets Layer)/TLS 등의 보안 기능을 사용할 수 있도록 해준다.

처음 Mbed가 소개되었을 때에는 Mbed 지원 하드웨어(①)와 API(②)를 통한 프로토타이핑에 집중하였다면, 사물인터넷을 위한 플랫폼으로 확장되면서 통신(③)과 보안(④) 기능들이 추가되었다. 이 책에서는 전통적인 Mbed, 즉 Mbed 환경을 지원하는 보드와 개발 도구를 사용하여 쉽게 사물인터넷을 위한 디바이스를 제작하는 방법을 다룬다. 이를 위해서는 먼저 Mbed 환경에서 사용할 수 있는 보드와 이를 지원하는 개발 환경이 필요하다. Mbed 환경에서 사용할 수 있는 보드는 Cortex-M 기반 마이크로컨트롤러를 생산하는 대부분의 회사에서 제작하여 판매하고 있으며, 그 종류가 100종을 넘는다. Mbed 보드는 Mbed 홈페이지에서 확인할 수 있으며 그림 1-3은 그중 일부를 나타낸 것이다.

Mbed 보드의 공통점이면서 Mbed 플랫폼의 가장 큰 장점 중 하나는 바로 모든 Mbed 보드에서 실행 가능한 공통의 코드를 작성할 수 있다는 점인데, 이를 가능하게 하는 것이 바로 Mbed에서 제공하는 API다. 서로 다른 보드들이 동일한 동작을 수행하기 위해서는 서로 다른 코드가 필요하다. 하지만 Mbed API는 이러한 코드를 추상화시켜 모든 Mbed 지원 보드에서 실행 가능한 코드를 작성할 수 있도록 해준다.

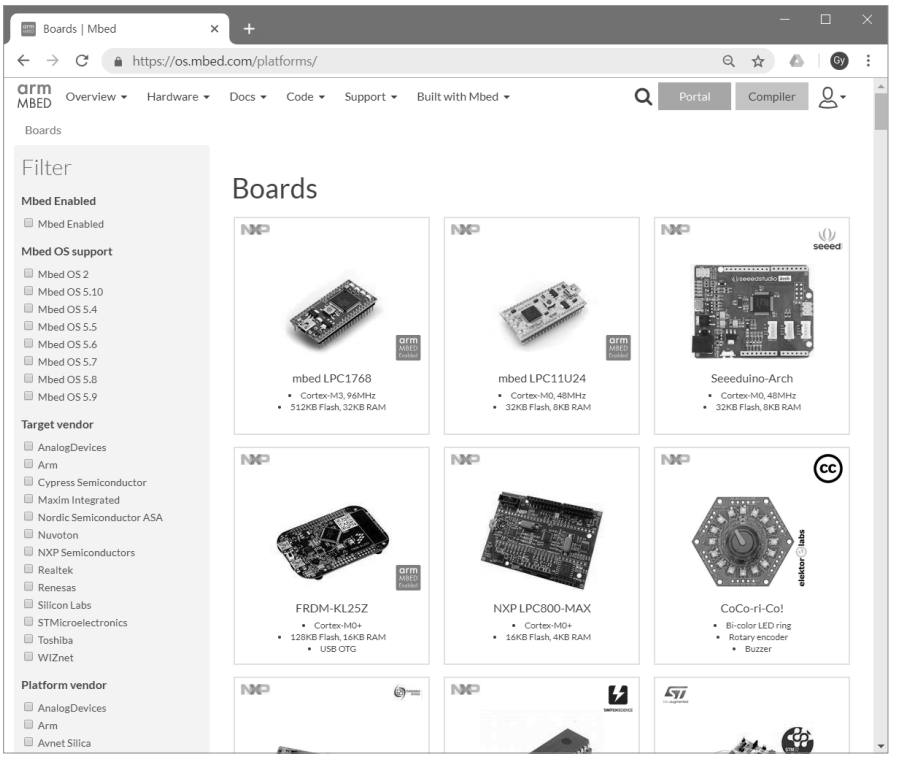


그림 1-3 **Mbed 보드**

Mbed에서 사용할 수 있는 개발 환경 역시 다양하다. 애초 Mbed 플랫폼은 온라인 개발 환경을 특징으로 하였지만, 다양한 개발 환경에 대한 요구가 증가함에 따라 오프라인 개발 환경 역시 제공하고 있다. ARM에서 제공하는 오프라인 개발 환경에는 명령 창을 통해 실행할 수 있는 Mbed CLI(Command-Line Interface)가 있으며, 이외에도 IAR, Keil, Eclipse 등 ARM 프로세서를 위한 통합 개발 환경을 사용하여 Mbed 보드를 위한 프로그램을 개발하는 것도 가능하다. 또한 최근에는 Mbed를 위한 통합 개발 환경인 Mbed Studio의 베타 버전이 공개되는 등 선택의 폭이 넓어졌다. 이 책에서는 온라인 개발 환경을 사용하여 프로그램을 작성하는 방법을 다룬다.

오프라인 개발 환경과 비교했을 때 온라인 개발 환경의 장점은 로컬 컴퓨터에 지루하고 복잡한 개발 환경 설치가 필요하지 않다는 점과 항상 최신의 개발 환경에서 작업할 수 있다는 점을 꼽을 수 있다. 또한 온라인 개발 환경에서는 각종 예제 및 공개된 프로그램을 사용해볼 수 있으며, 온라인 커뮤니티를 통해 다양한 정보를 얻을 수 있으므로 짧은 시간에 원하는 프로그램을 개발할 수 있다.

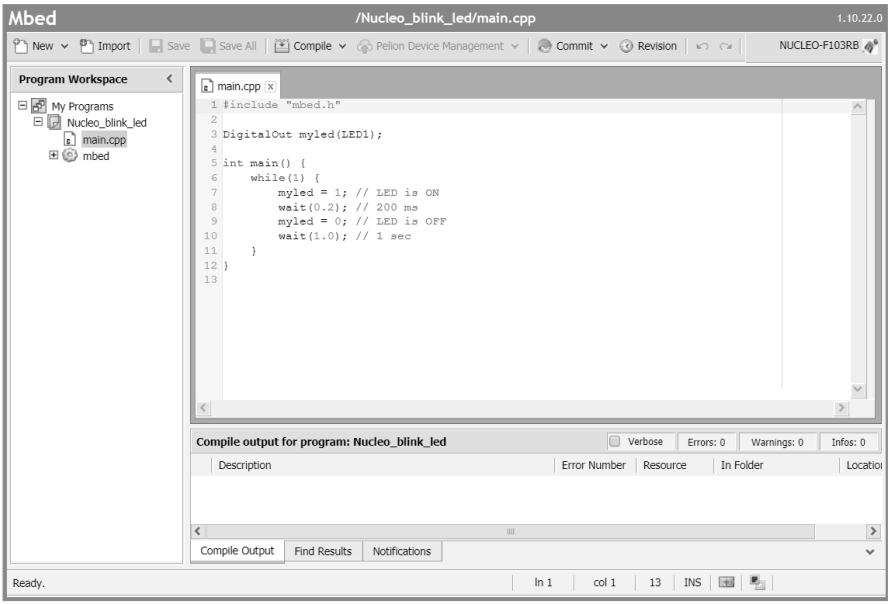


그림 1-4 **Mbed 온라인 개발 환경**

Mbed의 또 다른 특징은 고수준 API 기반의 프로그래밍에 있다. 마이크로컨트롤러 프로그래밍에서 가장 어려운 점은 레지스터 조작이다. Mbed에서는 빠른 프로토타이핑을 위해 저수준의 레지스터 조작을 고수준의 API로 대체하고 있으므로 하드웨어에 대한 깊은 지식 없이도 마이크로컨트롤러를 위한 프로그램을 작성할 수 있다. 즉, Mbed의 API는 하드웨어 종속적인 코드를 하드웨어 독립적인 코드로 바꾸어준다. 또한 Cortex-M 기반 마이크로컨트롤러 보드의 기능들을 계층적으로 추상화하고 있으므로 프로그래머는 동일한 인터페이스를 통해 서로 다른 회사에서 제작한 마이크로컨트롤러를 동일한 방식으로 제어함으로써 프로그램 작성의 부담을 줄일 수 있다. 고수준 API 기반의 프로그래밍은 아두이노의 특징이기도 하다. 아두이노 역시 저수준의 레지스터 조작 없이 고수준의 API만으로 프로그램 작성이 가능하지만, Mbed 환경은 아두이노와 비교했을 때 더 많은 기능을 API를 통해 사용할 수 있도록 해준다.

1.4 Mbed 보드

2005년에 시작된 Mbed 프로젝트에서 처음 Mbed 보드를 제작한 회사는 NXP[4]로, ARM7TDMI-S 코어를 사용하여 최초의 Mbed 보드 Mbed LPC2368을 선보였다. 이후 NXP

4 http://www.nxp.com

는 Cortex-M3 코어를 사용한 LPC1768을 출시하였고, LPC1768은 가장 많이 사용되는 대표적인 Mbed 보드 중 하나로 자리 잡았다. 인터넷에서 Mbed 보드를 검색해보면 여전히 LPC1768 보드가 가장 많이 눈에 띈다.

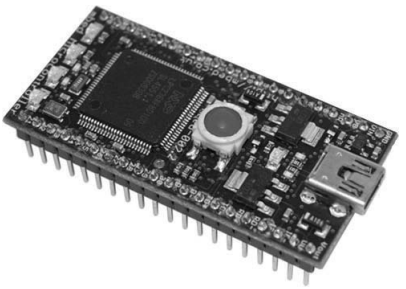


그림 1-5 **Mbed LPC1768 보드**

이후 NXP를 포함하여 프리스케일(Freescale), STMicroelectronics, 노르딕 세미컨덕터(Nordic Semiconductor) 등 주요 마이크로컨트롤러 제작 회사에서 다양한 Mbed 보드를 출시하여 Mbed 공식 홈페이지에 소개된 Mbed 보드만도 100종을 넘어서고 있다. 이들 보드에서 사용한 코어 역시 Cortex-M0에서 Cortex-M7에 이르기까지 모든 Cortex-M 시리즈가 사용되고 있다. 대표적인 Mbed 보드인 Mbed LPC1768에 대해 조금 더 살펴보자. 표 1-5는 Mbed LPC1768의 사양을 요약한 것이다.

표 1-5 **LPC1768 사양**

항목	사양
코어	ARM Cortex-M3(100핀 마이크로컨트롤러)
Mbed 보드 핀 수	40핀(0.1인치, 2.54mm 핀 간격)
동작 주파수	최대 100MHz
플래시메모리	512KB
SRAM	64KB
주변장치	이더넷, USB, SPI, I2C, CAN, UART
ADC	12bit 8채널
범용 입출력	최대 70개
타이머/카운터	32bit 4개

사양을 보면 알 수 있듯이 일반적인 마이크로컨트롤러 보드와 비교했을 때 Mbed LPC1768 보드에 특이한 점은 없다. Mbed 보드 역시 다양한 주변장치를 연결하여 제어장치를 구성하기 위해 사용되는 마이크로컨트롤러 보드의 한 종류일 뿐이다.

온라인 개발 환경과 고수준 API에 의한 프로그래밍이 소프트웨어 측면에서 Mbed의 장점이라면, 하드웨어 측면에서 Mbed의 장점은 Cortex-M 기반 마이크로컨트롤러가 32bit 마이크로컨트롤러라는 데 있다. 현재 시장에서 가장 많이 판매되는 마이크로컨트롤러는 8bit 마이크로컨트롤러이지만, 판매 금액에서는 32bit 마이크로컨트롤러가 8bit 마이크로컨트롤러를 이미 앞질렀으며, 가까운 시일 내에 판매량에서도 8bit 마이크로컨트롤러를 추월할 것으로 예상된다. 이러한 마이크로컨트롤러 시장 변화의 중심에 ARM의 Cortex-M 프로세서가 자리하고 있다. 강력한 성능과 다양한 주변장치를 연결할 수 있는 유연성에 가격 경쟁력까지 갖춘 Cortex-M은 기존 8bit 마이크로컨트롤러의 자리를 차지할 것으로 보인다. 이에 따라 쉽고 빠르게 프로토타입을 제작하고 사용 가능성을 테스트할 수 있도록 해주는 Mbed의 역할이 더욱 중요하다고 할 수 있겠다.

Mbed 보드의 또 다른 장점 중 하나는 간편한 업로드 방식에 있으며, 이를 위해 Mbed 보드는 특별한 업로더(uploader)를 사용한다. 마이크로컨트롤러는 교차 개발 환경에서 개발이 이루어진다. '교차'란 프로그램을 개발하는 환경과 개발된 프로그램이 실행되는 환경이 서로 다르다는 의미다. 윈도우에서 실행되는 프로그램의 개발은 윈도우 환경에서 비주얼 스튜디오와 같은 개발 도구를 사용하여 이루어진다. 하지만 마이크로컨트롤러에는 하드디스크와 같이 대용량 저장 장치가 없고 윈도우와 같은 운영체제도 없으므로 마이크로컨트롤러에서 응용 프로그램을 개발할 수는 없다. 따라서 마이크로컨트롤러를 위한 프로그램 개발은 컴퓨터(개발 시스템)에서 이루어지고, 작성된 프로그램은 마이크로컨트롤러(목적 시스템)로 옮겨서 실행된다. 이처럼 개발 시스템과 목적 시스템이 다른 환경을 교차 개발 환경이라고 한다.

교차 개발이 가능하기 위해서는 개발 시스템에서 동작하면서 목적 시스템에서 실행 가능한 기계어 파일을 생성할 수 있는 교차 컴파일러(cross compiler)와 생성된 기계어 파일을 목적 시스템으로 옮기는 방법이 필요하다. Mbed 환경에서는 웹을 통해 교차 컴파일러를 제공하고 있다. 웹을 통해 프로그램을 작성하고 실행 파일이 생성된 후에는 마이크로컨트롤러로 프로그램을 옮기는 작업이 필요한데, 이 과정을 흔히 '업로드(upload)'라고 한다. 업로드를 위해서 사용하는 장치가 바로 업로더로, AVR에서는 ISP(In System Programming) 방식, ARM에서는 JTAG(Joint Test Action Group) 방식을 주로 사용한다. 업로드를 위해서는 전용 장치가 필요하다.

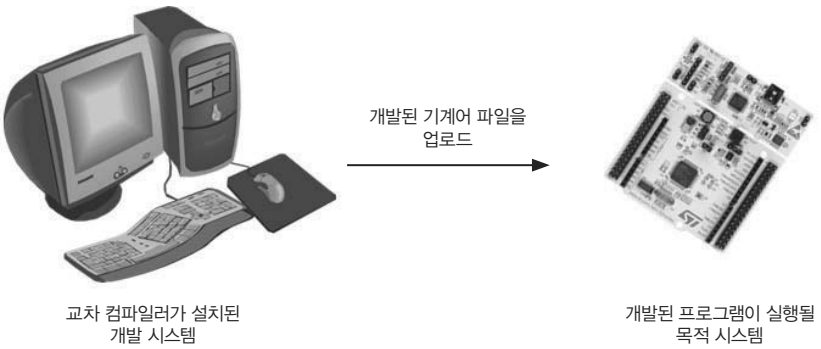


그림 1-6 교차 개발 환경

Mbed 보드는 업로드를 위해 별도의 하드웨어가 필요하지 않으며, Mbed 보드에 내장된 업로드 전용 마이크로컨트롤러를 사용한다. 대부분의 Mbed 보드에는 2개의 마이크로컨트롤러가 포함되어 있다. 그중 하나가 프로그래머가 원하는 작업을 수행하기 위한 메인 마이크로컨트롤러라면, 다른 하나는 프로그램 업로드 전용으로 사용되는 마이크로컨트롤러다. 업로드에 사용되는 마이크로컨트롤러를 USB로 컴퓨터와 연결하면 컴퓨터에서는 USB 메모리와 마찬가지로 외부 저장 장치로 인식한다. 따라서 온라인 컴파일러를 통해 만들어진 실행 파일을 외부 저장 장치에 복사해 넣기만 하면 자동으로 프로그램이 설치되고 실행된다. 이 또한 아두이노와 유사한 점이 있다. 대부분의 아두이노 보드 역시 2개의 마이크로컨트롤러를 포함하고 있으며, 이 중 하나는 프로그램 업로드 전용으로 사용되고 있으므로 추가적인 장치 없이도 아두이노 보드에 프로그램을 설치하고 실행할 수 있다. 물론 아두이노 보드를 컴퓨터와 연결하였을 때에는 아두이노 보드를 저장 장치로 인식하지는 않는다.

1.5 Mbed 보드의 단점

Mbed 보드는 온라인 개발 환경을 사용하므로 개발 환경 구축을 위해 프로그램을 설치할 필요가 없고, 추상화된 공통의 API를 사용하므로 다양한 개발 보드들을 동일한 코드를 통해 제어할 수 있으며, 별도의 하드웨어 장치 없이 프로그램을 업로드할 수 있는 등 많은 장점이 있다. 하지만 단점 하나 없이 좋기만 한 것이 있을 리 만무하다. Mbed 역시 마찬가지다. Mbed의 단점을 꼽자면 널리 보급되지 못했다는 점을 들 수 있다. 특히 국내에서는 사용자가 더 적다. 사용하는 사람이 적다면 관련 정보를 얻기가 수월하지 않은 것은 당연하다. 앞의 설명에서 느꼈겠지만, 아두이노와 Mbed는 주로 사용하는 마이크로컨트롤러의 종류가 다른 점을 제외하

면 개발 철학이 유사하다. 아두이노는 8bit 마이크로컨트롤러를 주로 사용하므로 집요하게 비전문가나 초보자들을 파고들어 독자적인 생태계 구축에 성공했지만, Mbed는 32bit 마이크로컨트롤러를 사용하므로 초보자나 비전문가들이 처음으로 선택할 수 있는 마이크로컨트롤러로는 적당하지 않았던 것이 사실이며, ARM의 지원 또한 미미했다.

하지만 세상은 바뀌고 있다. 최근 사물인터넷이 주목을 받으면서 간단한 제어장치를 만들고 테스트할 수 있는 쉽고 빠른 프로토타이핑 환경의 필요성은 점차 증가하고 있다. 구글, 애플, 삼성 등 주요 IT 기업들은 모두 독자적인 사물인터넷 개발 환경을 발표하였거나 발표를 준비하고 있다. 여기에 발맞추어 ARM 역시 사물인터넷 환경에 맞게 단순한 프로토타이핑 도구에서 사물인터넷을 위한 플랫폼으로 Mbed를 확장하여 다양한 기능을 통합하고 있으므로 향후 행보에 더욱 주목할 필요가 있다.

Mbed가 널리 보급되지 못한 또 다른 원인은 높은 가격도 큰 몫을 했다. 대표적인 Mbed 보드인 LPC1768의 가격은 약 50달러로, 그림 1-5에서 보듯이 별다른 것이 없어 보이는 마이크로컨트롤러 보드의 가격으로 보기에는 비싼 면이 없지 않다. 프로그램 업로드를 위한 전용의 마이크로컨트롤러 등 보이는 것보다는 많은 것이 포함되어 있는 데다 개발 환경 구축이 필요하지 않다는 점 등을 생각하면 50달러의 가격이 그리 비싼 것이 아니지만, 아직은 어렵고 복잡한 것으로 여겨지는 ARM 기반 보드의 가격으로는 접근성이 떨어지는 것이 사실이다. 하지만 최근 이러한 가격 문제는 여러 회사에서 다양한 Mbed 보드를 출시하면서 개선되고 있다. 2014년 상반기에 STMicroelectronics에서 출시한 누클레오(Nucleo) 시리즈 보드 중 Mbed LPC1768과 같은 Cortex-M3 코어를 사용한 보드는 아두이노의 생태계를 활용할 수 있도록 아두이노와 호환되는 핀 헤더까지 포함하고 있으면서 가격은 약 10달러에 지나지 않는다.

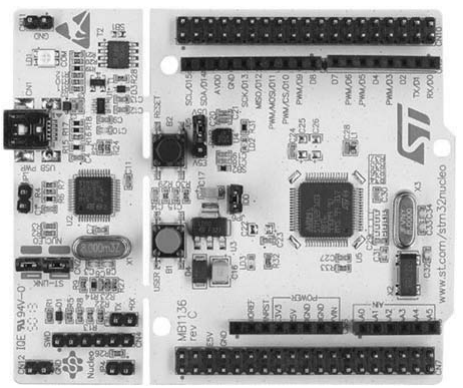


그림 1-7 **STMicroelectronics NUCLEO-F103RB 보드**

위에서 언급한 단점들이 하드웨어 측면이라면 개발 환경, 즉 소프트웨어 측면에서의 단점 역시 존재하는데, 디버깅이 불편하다는 점이 그중 하나다. 컴파일러에 의해 실행 파일이 생성될 때 문법 오류는 검사할 수 있지만, 실행 중에 발생하는 런타임 오류는 디버깅을 위한 장비를 이용하여 코드를 한 줄씩 실행시키면서 찾아내야 한다. 사실 온라인 환경에서의 디버깅은 쉽지 않다. 하지만 오프라인 개발 환경을 사용한다면 전용 장비를 통해 일반적인 Cortex-M 기반 마이크로컨트롤러 보드와 같은 방법으로 디버깅이 가능하다. 이는 아두이노 역시 마찬가지다. 아두이노 역시 디버깅을 위한 도구나 환경을 제공하지는 않지만, 마이크로칩에서 제공하는 개발 환경과 전용 장치가 준비되어 있다면 아두이노 보드 역시 디버깅이 가능하다.

1.6 Mbed와 아두이노

Mbed와 아두이노는 여러 종류의 보드를 위한 공통의 개발 환경, 고수준의 API, 손쉬운 업로드 방식 등 많은 장점을 공유하고 있을 뿐만 아니라 디버깅이 어려운 단점까지도 동일한 것처럼 많은 부분에서 비슷하다. 게다가 우연의 일치겠지만 아두이노는 2005년에 이탈리아에서 처음 소개되었고, Mbed 역시 ARM의 연구원들에 의해 2005년에 시작되었다. 하지만 두 플랫폼은 주로 사용하는 마이크로컨트롤러의 종류에서도 알 수 있듯이 그 출발점이 다르다. 아두이노에도 Cortex-M 마이크로컨트롤러를 사용하여 만들어진 보드가 존재하지만, 애초 아두이노는 비전공자들이 8bit의 AVR 마이크로컨트롤러를 사용하여 쉽고 간단한 제어장치를 구성하기 위한 목적으로 시작되었다. 반면 Mbed의 목적은 ARM의 Cortex-M 마이크로컨트롤러를 사용하여 짧은 시간에 프로토타입을 만들기 위한 것이다.

애초의 목적이 달라서이겠지만 이후의 행보에도 많은 차이가 있다. 아두이노가 초보자와 비전문가를 위한 도구로 확고한 자리매김을 하였다면, Mbed는 지금까지 눈에 띄는 행보를 보여주지 못했다. 그렇다면 이 책에서 Mbed를 다루는 이유는 무엇일까? Mbed의 가치는 어디에서 찾아야 할까? 결론부터 말하자면 아두이노와 Mbed는 출발점이 다르듯이 나아가는 방향이 다르며 시장에서 차지하는 영역 역시 다르다. 아두이노는 즐겁다. 가볍게 재미있는 여러 장치를 만들어보고 싶다면 아두이노를 선택하면 된다. 반면 Mbed는 진지하다. 아두이노만큼은 아니지만 ARM 프로세서를 사용한 시스템을 개발하기 수월할 뿐 아니라 아두이노로 구현이 어려운 고성능의 복잡한 시스템도 Mbed를 사용한다면 만들 수 있다. Mbed로는 가능하지만 아두이노로는 불가능한 일을 찾아보기는 어렵다. 하지만 할 수 없는 것과 하지 않는 것은 차이가 있다. 아두이노를 인터넷에 연결하여 사물인터넷의 일부로 동작하도록 하는 것은 가능하다.

하지만 Mbed 보드 중에는 훨씬 싼 가격으로 동일한 기능을 짧은 시간에 구현할 수 있도록 해주는 보드가 존재한다. 아두이노에 카메라를 연결하여 영상 처리를 하고 싶다면 아두이노 보드보다 비싼 영상 처리 보드가 필요하다. 하지만 Mbed는 아두이노보다 훨씬 쉽게 카메라를 연결하고 영상을 분석할 수 있다. 물론 선택은 독자들의 몫이다. 32bit 마이크로컨트롤러에 관한 관심이 늘어나고 있고, Mbed에 대한 지원 역시 증가하고 있는 상황에서 Mbed로 눈을 돌려보는 것은 나쁘지 않은 선택이 될 것이다. 오히려 사물인터넷 환경에서 필수가 되리라 생각한다. 닭 잡는 데 소 잡는 칼을 쓸 필요는 없지만, 2개의 칼을 가지는 것이, 특히나 Mbed라는 칼이 아주 유용하다면 가지고 싶은 욕심이 생기지 않는가?

CHAPTER 03

마이크로컨트롤러 프로그래밍

마이크로컨트롤러를 위한 프로그램은 C/C++ 언어를 포함하여 여러 가지 프로그래밍 언어로 작성할 수 있다. C/C++은 Java와 함께 프로그래밍 언어 중에서 가장 많이 사용되는 언어 중 하나다. Java가 인터넷의 보급에 따라 다양한 운영체제에서 동작할 수 있는 호환성을 중시한 언어라면, C/C++은 Java의 모태가 된 언어로 이전보다 사용이 줄기는 했지만, 여전히 다양한 분야에서 사용되고 있다. 윈도우와 리눅스 같은 운영체제도 C 언어로 만들어졌으며, 마이크로컨트롤러를 포함하여 하드웨어와 관련된 분야에서는 대부분 C/C++ 언어를 사용한다. Mbed 환경에서도 C/C++ 언어를 사용하며 Mbed API는 C++의 클래스 라이브러리를 바탕으로 하고 있다.

Mbed를 위한 프로그램을 작성하기 위해서는 당연히 C/C++ 언어를 사용할 수 있어야 하고, Mbed API 사용을 위해서는 C++의 클래스에 익숙해져야 한다. C++ 언어는 코드의 재사용성을 높이고 프로그램의 작성 과정을 간단히 해주는 장점이 있지만, C 언어와 비교했을 때 컴파일 후 실행 파일의 크기가 커지고 클래스에 대한 이해가 선행되어야 하는 점 등은 단점이 될 수 있다. 하지만 컴파일러의 성능이 개선되고 마이크로컨트롤러의 플래시메모리 크기가 증가함에 따라 실행 파일 크기는 큰 문제가 되지 않는다. 클래스에 대한 이해가 쉽지 않은 것은 사실이지만 마이크로컨트롤러 프로그래밍에서는 클래스 라이브러리를 직접 만드는 것보다 만들어진 클래스 라이브러리를 사용하는 경우가 대부분이므로 사용 방법에 익숙해지는 것이 먼저다. 아두이노는 쉬운 사용법을 장점으로 내세우고 있는데, 그 일부로 클래스 라이브러리를 제공하고 있다는 점은 클래스 라이브러리를 사용하기가 그리 어렵지 않다는 사실을 보여주는 예

라 할 수 있다. 이 장에서는 Mbed에서 제공하는 예제를 중심으로 마이크로컨트롤러 프로그래밍에 필요한 C/C++ 언어의 기본 구조와 클래스 라이브러리 사용법을 살펴본다. 하지만 C/C++ 언어 자체에 관한 내용은 다루지 않고 C/C++ 언어를 사용하는 일반적인 프로그램과 마이크로컨트롤러를 위한 프로그램의 차이점을 중심으로 살펴볼 것이므로 C/C++ 언어에 대한 자세한 설명은 다른 책을 참고하자.

3.1 블링크 프로그램

Mbed에서 기본적으로 제공하는 예제 중 하나에 블링크(blinky LED hello world)가 있다. 블링크 프로그램은 0.2초 동안 LED를 켜고 1초 동안 LED를 끄는 동작을 반복하는 프로그램으로 코드 3-1과 같다. 먼저 코드 3-1이 일반적인 C/C++ 프로그램과 다른 점이 무엇인지 살펴보자.

코드 3-1 블링크 프로그램

```
1   #include "mbed.h"
2
3   DigitalOut myled(LED1);
4
5   int main() {
6     while(1) {
7       myled = 1;                          // LED 켜짐
8       wait(0.2);                          // 200ms
9       myled = 0;                          // LED 꺼짐
10      wait(1.0);                          // 1s
11    }
12  }
```

코드 3-1은 크게 전처리 문장(#include), DigitalOut 객체 선언, 그리고 main 함수의 세 부분으로 나누어볼 수 있다. Mbed의 API를 사용하기 위해서는 전처리 문장(Line 1)으로 헤더 파일 mbed.h를 반드시 포함해야 한다. 물론 mbed.h 파일을 포함하는 것만으로 Mbed API를 사용할 수 있는 것은 아니며, 컴파일러와 링커를 위한 옵션 설정 등이 필요하지만 이는 개발 환경 내에서 자동으로 처리되므로 걱정하지 않아도 된다.

Line 3은 DigitalOut 객체를 선언하는 부분이다. DigitalOut은 디지털 데이터를 출력할 수 있는 범용 입출력 핀을 제어하기 위한 클래스의 이름이며, myled는 DigitalOut 클래스로 만들어진 객체에 해당한다. 인자로 주어진 LED1은 LED가 연결되어 있는 범용 입출력 핀을 가리키는 상수다. myled 객체에 1을 대입하면(Line 7) 연관된 범용 입출력 핀인 LED1 핀으로 HIGH가

출력되어 LED가 켜지고, 0을 대입하면(Line 9) LED1 핀으로 LOW가 출력되어 LED가 꺼진다.

main 함수는 while 문 이전의 초기화 부분과 while 문으로 나눌 수 있다. C 언어에서 가장 먼저 실행되는 함수는 main 함수이므로 프로그램이 실행될 때 가장 먼저 실행되는 부분은 초기화 부분이다. 초기화 부분에서는 단어 의미 그대로 변수의 초기화나 레지스터 설정 등을 수행한다. 코드 3-1에서는 초기화 부분이 없지만, Line 3의 객체 선언은 초기화 부분으로 옮길 수 있다.

while 문에서 조건 부분에는 '1'이 들어 있다. C 언어에서 영(zero)이 아닌 모든 정수는 참(true)으로 간주되므로 while(1) 문장은 무한 루프를 형성하며 이를 '메인 루프' 또는 '이벤트 루프'라고 한다. 마이크로컨트롤러에는 일반적으로 하나의 프로그램만 설치되고 설치된 프로그램은 전원이 꺼질 때까지 종료하지 않는다. 따라서 마이크로컨트롤러에서 실행되는 프로그램은 메인 루프 내에서 주변장치로부터 데이터를 받아 이를 처리하고 출력하는 일을 반복한다. 즉, 마이크로컨트롤러를 위한 프로그램은 메인 루프를 무한히 반복해서 실행하는 구조를 가지는데, 이는 일반적인 C/C++ 프로그램과는 다른 점이다. 윈도우를 위한 프로그램 역시 메시지 루프라고 불리는 무한 루프를 가지고 있지만, 프로그램 종료를 위한 메시지가 존재하고 프로그램을 종료할 수 있다는 점에서 차이가 있다. 그림 3-1은 마이크로컨트롤러를 위한 프로그램의 구조를 요약한 것이다.

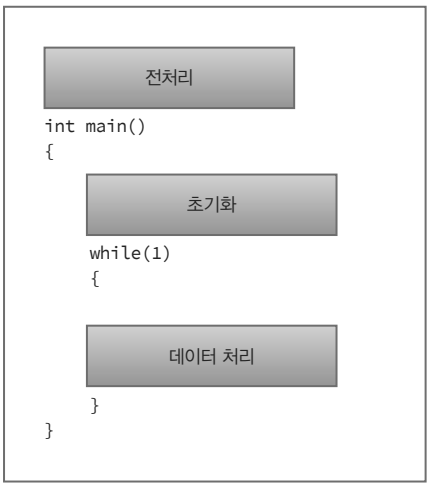


그림 3-1 **마이크로컨트롤러를 위한 프로그램 구조**

3.2 객체의 사용

코드 3-1 Line 3의 DigitalOut 객체를 선언하는 부분은 C/C++ 언어에는 없는 새로운 데이터 유형을 사용할 수 있도록 해주는 방법의 하나로 C 언어에서 정수형 변수를 선언하는 부분과 다르지 않다.

```
DigitalOut myled(LED1);                    // DigitalOut형 변수 선언
int myNumber;                              // 정수형(int) 변수 선언
```

C 언어에서 변수는 하나의 값만을 저장할 수 있다. 이를 확장하여 연관된 여러 개의 값을 저장하기 위해 사용할 수 있는 방법 중 하나로 구조체(structure)가 있다. 구조체는 서로 다른 종류의 값을 가지는 변수들을 연관 지어 사용할 수 있도록 해준다. 코드 3-2는 학생 정보를 나타내기 위해 구조체를 정의하여 사용하는 예를 보여준다.

코드 3-2 구조체 사용

```
1   #include "mbed.h"
2
3   typedef struct{                                // 구조체를 새로운 유형의 변수로 정의
4     int age;                                     // 나이 필드
5     float weight;                                // 몸무게 필드
6   }Student;                                      // 새로운 유형의 이름
7
8   int main() {
9     Student student;                             // Student형 변수 선언
10
11    student.age = 21;                            // 구조체의 각 필드에 변숫값 대입
12    student.weight = 61.5;
13
14    printf("Age : %d, Weight : %f\n", student.age, student.weight);
15
16    while(1) {
17    }
18  }
```

구조체에는 연관된 여러 개의 값을 저장할 수 있으며 각각의 값이 저장되는 공간을 필드(field)라고 한다. 필드에는 C/C++ 언어에서의 기본 데이터 유형인 정수(int), 실수(float, double), 문자(char), 배열 등이 올 수 있다. 배열 역시 여러 개의 값을 저장하기 위해 사용할 수 있지만, 배열은 같은 종류의 값을 저장하기 위해 사용한다면 구조체는 다른 종류의 값을 저장할 수 있는 차이가 있다. 또한 여러 개의 값이 저장되어 있을 때 그중 하나의 값을 알아

내기 위해서 배열에서는 배열 연산자인 '[]'을 사용한다면 구조체에서는 도트 연산자 '.'를 사용한다(Line 11, 12, 14).

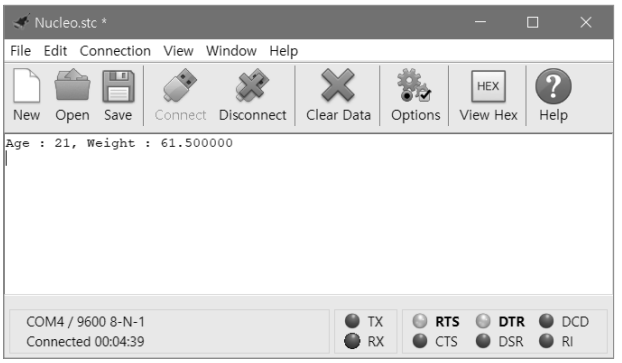


그림 3-2 **코드 3-2 실행 결과**

구조체가 변수들의 묶음이라면 클래스는 연관된 변수에 연관된 함수까지도 함께 포함된 것으로 생각할 수 있다. 이때 연관된 변수들은 멤버 변수(member variable), 연관된 함수들은 멤버 함수(member function)라고 한다. 사실 클래스를 설계하기는 쉽지 않다. 하지만 클래스를 사용하는 방법은 구조체를 사용하는 방법과 크게 다르지 않다. 멤버 변수를 사용하기 위해서는 구조체에서처럼 도트 연산자를 사용하며 멤버 함수 역시 마찬가지다. 하지만 클래스에서 멤버 변수는 도트 연산자를 사용하여 직접 접근하는 방법을 추천하지 않으며 멤버 함수를 통해 접근하도록 권장하고 있다. 코드 3-3은 코드 3-2와 같은 동작을 클래스를 사용하여 구현한 예다. 코드가 길고 복잡해 보일 수도 있지만, Line 3에서 Line 12까지의 클래스 정의를 제외하면 구조체를 사용하는 코드 3-2와 크게 다르지 않다.

코드 3-3 **클래스 사용**

```
1   #include "mbed.h"
2
3   class Student{                                  // 클래스 정의
4   private:                                        // 직접 접근할 수 없는 멤버 변수
5     int age;
6     double weight;
7   public:                                         // 직접 접근할 수 있는 멤버 함수
8     void setAge(int _age) { age = _age; };
9     void setWeight(double _weight) { weight = _weight; };
10
11    int getAge() { return age; };
12    double getWeight() { return weight; };
13  };
```

```
14
15  int main() {
16    Student student;
17
18    student.setAge(21);                  // 멤버 함수를 통한 대입
19    student.setWeight(61.5);
20
21    // 멤버 함수를 통한 값 읽기
22    printf("Age : %d, Weight : %f\n", student.getAge(), student.getWeight());
23
24    while(1) {
25    }
26  }
```

코드 3-3에서는 클래스의 멤버 변수에 값을 대입하기 위해 set로 시작하는 멤버 함수를, 값을 읽어오기 위해 get으로 시작하는 멤버 함수를 사용하고 있다. 하지만 코드 3-1에서는 객체에 직접 값을 대입하는데, 이는 연산자 오버로딩을 통해 가능하다. 대입 연산자(=)는 오른쪽의 값을 왼쪽에 있는 변수에 대입한다. 코드 3-1의 경우 LED 상태를 나타내는 변수 하나만 DigitalOut 클래스에 멤버 변수로 정의되어 있으므로 직관적으로 이해할 수 있지만, 코드 3-3의 Student 클래스에는 2개의 멤버 변수가 정의되어 있으므로 어느 변수에 대입할 것인지 명확하지 않다. 다행히 두 변수는 정수(int)와 실수(double)로 서로 다른 유형의 값을 가지므로 코드 3-1에서와 비슷하게 대입 연산자 오버로딩이 가능하다. 비슷하게 캐스팅(casting) 연산자 오버로딩을 통해 값을 읽어오는 것도 가능하다.

코드 3-4는 대입 연산자와 캐스팅 연산자 오버로딩을 통해 멤버 변수에 값을 대입하고 읽어오는 방법을 보여주고 있다. 한 가지 주의할 점은 코드 3-4에서의 연산자 오버로딩은 Student 클래스에 서로 다른 유형의 멤버 변수가 2개만 존재하기 때문에 가능한 것이며 일반적으로 가능한 것은 아니라는 점이다. age와 weight가 모두 정수형(int) 변수라면 연산자 오버로딩을 통해서는 값을 대입할 수 없으며 멤버 함수를 사용해야 한다.

코드 3-4 대입 연산자 오버로딩

```
1   #include "mbed.h"
2
3   class Student{
4   private:
5     int age;
6     double weight;
7   public:                                // 연산자 오버로딩 정의
8     void operator=(const int &_age) { age = _age; };
```

```
9     void operator=(const double &_weight) { weight = _weight; };
10
11    operator int() { return age; };
12    operator double() { return weight; };
13  };
14
15  int main() {
16    Student student;
17
18    student = 21;                          // 대입 연산자 오버로딩을 통한 대입
19    student = 61.5;
20
21    // 캐스팅 연산자 오버로딩을 통한 값 읽기
22    printf("Age : %d, Weight : %f\n", (int)student, (double)student);
23
24    while(1) {
25    }
26  }
```

코드 3-2, 코드 3-3, 코드 3-4에서 변수에 값을 대입하고 읽어오는 과정을 비교한 것이 표 3-1이다.

표 3-1 구조체와 클래스에서 변수 사용

	변숫값 대입	변숫값 읽기
구조체	student.age = 21;	a = student.age;
	student.weight = 61.5;	w = student.weight;
클래스(멤버 함수)	student.setAge(21);	a = student.getAge();
	student.setWeight(61.5);	w = student.getWeight();
클래스(연산자 오버로딩)	student = 21;	a = (int)student;
	student = 61.5;	w = (double)student;

앞에서도 언급한 것처럼 클래스는 쉽게 사용할 수 있도록 설계하고 구현하는 것이 어려운 것이지 사용하는 것은 그리 어렵지 않다. 표 3-1에서 세 가지 경우의 차이가 그리 크지 않다는 점에서도 눈치챘을 것이다. 코드 3-3과 코드 3-4에서 클래스를 구현하는 방법을 소개하기는 했지만, 이 책에서는 클래스를 구현하지 않고 Mbed API의 사용 방법을 위주로 살펴볼 것이므로 구현 부분을 이해하지 못했다 하더라도 걱정할 필요는 없다.

3.3 맺는말

Mbed 환경에서 프로그램은 C/C++ 언어를 사용하여 만들어진다. 특히 Mbed API는 C++의 클래스 라이브러리를 통해 구현되어 있으므로 클래스의 사용 방법을 알아두는 것은 필수적이다. 앞의 예에서도 볼 수 있듯이 클래스의 사용법은 구조체와 비슷하다. 어려운 점은 필요로 하는 클래스의 기능을 정의하고 이를 쉽게 사용할 수 있도록 구현하는 것이지만, Mbed 환경에서는 다양한 마이크로컨트롤러의 기능을 클래스 라이브러리로 구현해두고 있으므로 그 사용법을 알아두는 것에서 시작할 수 있다. 관심 있는 독자라면 C++ 언어와 객체지향 관련 책을 찾아볼 것을 추천한다. 이를 통해 Mbed API를 깊이 있게 이해하고 나아가 Cortex-M 마이크로컨트롤러에 대한 이해를 넓힐 수 있을 것이다.

CHAPTER 05

Mbed API

5.1 Mbed API

Mbed API(Application Programming Interface)는 Mbed 보드를 위한 프로그램을 작성하기 위해 사용할 수 있는 함수와 클래스의 집합을 가리킨다. Mbed API는 서로 다른 회사에서 제작한 다양한 종류의 마이크로컨트롤러를 같은 방법으로 제어할 수 있도록 계층적으로 구성되어 있다.

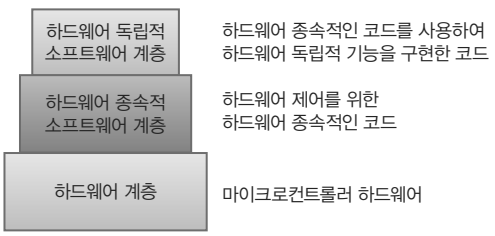


그림 5-1 **Mbed API의 계층 구조**

Mbed API는 크게 3개의 층을 통해 구현되며 가장 아래쪽에는 하드웨어인 마이크로컨트롤러(또는 마이크로컨트롤러의 레지스터)가 위치한다. 하드웨어 계층 위에는 하드웨어에 종속적인 층이 존재하는데, 이는 마이크로컨트롤러에 따라 서로 다른 레지스터를 사용하여 같은 기능을 구현하는 데 필요한 소프트웨어 계층에 해당한다. 하드웨어 종속적인 층은 마이크로컨트롤러의 종류에 따라 다르게 구현된다. 하드웨어 종속적인 층 위에는 마이크로컨트롤러 종류와 무관하게 같은 기능을 같은 API를 통해 제어할 수 있도록 만들어진, 하드웨어와 독립적인 계층이 존

재한다. 따라서 만약 새로운 마이크로컨트롤러를 Mbed 환경에서 사용하고자 한다면 두 번째 계층인 하드웨어 종속적인 소프트웨어 계층을 구현하면 된다.

마이크로컨트롤러 프로그래밍에서는 레지스터 조작이 기본이 된다. 하지만 레지스터를 조작한다는 것은 그리 쉽지 않다. 따라서 레지스터 조작을 간편하게 할 수 있도록 레지스터 조작 작업을 함수로 구현한 하드웨어 종속적인 소프트웨어 계층이 필요하다. 또한 서로 다른 마이크로컨트롤러 사이에 소스 코드의 호환성을 보장하기 위해 서로 다른 마이크로컨트롤러의 서로 다른 레지스터를 같은 함수로 조작할 수 있는 하드웨어 독립적인 소프트웨어 계층이 필요하다.

레지스터란 무엇인가? 마이크로프로세서에서 레지스터란 연산을 수행하는 데 필요한 피연산자와 연산 결과를 임시로 저장하는 용도로 주로 사용되는 CPU(중앙처리장치) 내의 임시 기억장소를 가리킨다. 하지만 윈도우 환경에서 윈도우 응용 프로그램을 작성할 때 레지스터의 값에 관심을 가지는 경우는 거의 없다. 이는 마이크로컨트롤러에서도 마찬가지다. 마이크로컨트롤러에서 중요한 레지스터는 마이크로프로세서에서는 없는 '입출력 레지스터'다. 마이크로컨트롤러는 마이크로프로세서에 해당하는 CPU 이외에도 주변장치와 데이터를 교환하기 위한 장치가 포함되어 있으며, 이를 제어하는 데 필요한 레지스터가 바로 입출력 레지스터다.

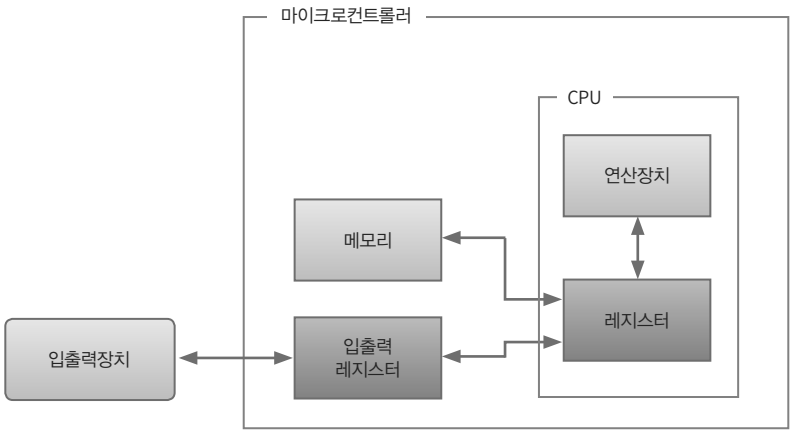


그림 5-2 **마이크로컨트롤러의 구조**

입출력 레지스터는 메모리의 한 종류로 번지가 지정되어 있으며 입출력 레지스터에 쉽게 접근할 수 있도록 이름이 정해져 있다. 따라서 입출력 레지스터는 미리 정의된 변수로 생각할 수 있다. 또한 기능에 따라 연관된 레지스터를 묶어 구조체로 정의하기도 하며, 더 나아가 Mbed

API에서는 객체지향 기법에 따라 클래스를 사용하여 레지스터를 제어할 수 있도록 구현되어 있다. 물론 일부는 클래스가 아닌 함수로 제공되어 복잡하게 보이고 복잡한 것이 사실이다. 그렇다고 해서 걱정할 필요는 없다. 이 책에서는 하드웨어와 관련된 레지스터와 하드웨어 종속적인 소프트웨어는 다루지 않으며 직관적으로 사용할 수 있도록 만들어진, 하드웨어와 독립적인 클래스와 함수를 다룬다. 다만 이 장에서는 하드웨어 독립적인 클래스들이 구현된 과정을 살펴봄으로써 마이크로컨트롤러 프로그래밍이 그리 간단하지는 않다는 점과 Mbed API를 통해 복잡하고 어려운 과정을 생략하고 간단하게 코드를 작성할 수 있다는 점을 살펴보고자 한다.

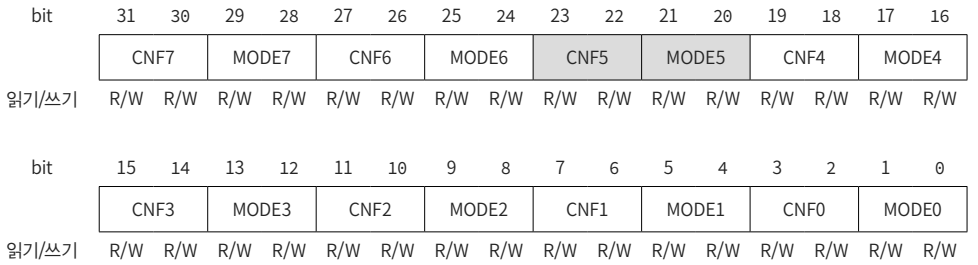
5.2 하드웨어 종속적 프로그래밍

NUCLEO-F103RB 보드에 포함된 LED를 레지스터 조작을 통해 점멸시키는 코드를 작성해보자. NUCLEO-F103RB 보드에는 포트 A의 5번 핀(PA_5, LED1, 또는 D13)에 연결된 LED가 포함되어 있다. 범용 입출력 핀에 연결된 LED 제어는 두 단계로 이루어진다. 입출력 핀은 글자 그대로 (버튼을 통해 데이터를 읽어오는) 입력과 (LED를 통해 데이터를 표시하는) 출력으로 사용할 수 있지만, 입력과 출력으로 동시에 사용할 수는 없다. 따라서 먼저 LED가 연결된 입출력 핀을 출력으로 사용하도록 설정해야 하는데, 이는 해당 포트의 입출력 제어 레지스터를 조작함으로써 가능하다. 출력으로 설정한 이후 LED를 켜거나 끄기 위해서는 해당 핀으로 출력할 데이터를 저장하는 레지스터에 값을 기록하면 된다.

> 마이크로컨트롤러와 주변장치의 데이터 교환은 핀을 통한 비트 단위 데이터를 기본으로 한다. 하지만 C/C++ 언어에서 논릿값은 0과 0이 아닌 값으로 false와 true를 구별한다는 점을 기억할 것이다. 즉, 논릿값을 표현하기 위해 비트 단위 데이터가 사용되지는 않으며, 실제 연산도 비트 단위로 이루어지지 않는다. 따라서 마이크로컨트롤러에서는 핀을 포트 단위로 묶어서 관리한다.
>
> STM32F103RBT6 마이크로컨트롤러에서 포트는 16개 핀으로 구성되며 레지스터 역시 포트 단위의 제어 기능을 제공하고 있다. STM32F103RBT6 마이크로컨트롤러에는 A, B, C, D의 4개 포트가 있다. 각 포트는 16개 핀으로 구성되지만, 포트 D의 경우 마이크로컨트롤러 칩의 핀 수가 부족하여 0~2번의 3개 핀만 할당되어 있다. 따라서 STM32F103RBT6 마이크로컨트롤러에서 사용할 수 있는 범용 입출력 핀의 수는 51(= 16 × 3 + 3)개가 된다.

LED가 연결되어 있는 포트 A의 핀을 출력으로 설정하기 위해 사용되는 레지스터에는 GPIOA_CRL(Port Configuration Register Low)과 GPIOA_CRH(Port Configuration Register High)가

있다. 이 중 GPIOA_CRL 레지스터는 포트 A의 16개 핀 중 0번에서 7번까지 핀의 입출력을 설정하기 위해 사용되고, GPIOA_CRH 레지스터는 8번에서 15번까지 핀의 입출력을 설정하기 위해 사용된다. 따라서 여기서는 GPIOA_CRL 레지스터를 사용한다. GPIOA_CRL은 32bit 레지스터로 4개 비트씩 묶어서 하나의 핀 상태를 설정하므로 5번 핀의 상태를 설정하기 위해서는 GPIOA_CRL 레지스터의 20번에서 23번까지 4개 비트를 사용한다.



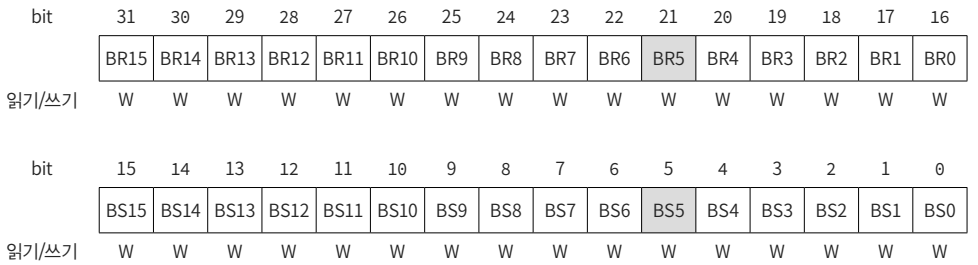
bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	CNF7		MODE7		CNF6		MODE6		CNF5		MODE5		CNF4		MODE4	
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CNF3		MODE3		CNF2		MODE2		CNF1		MODE1		CNF0		MODE0	
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

그림 5-3 **GPIOA_CRL 레지스터 구조**

MODEn(n = 0, ..., 7)은 핀을 입력 또는 출력으로 설정하기 위해 사용되는 비트로 11_2로 설정하면 최대 50MHz의 출력으로 사용할 수 있다. CNFn(n = 0, ..., 7)는 MODEn이 출력으로 설정된 경우 00_2로 설정하면 푸시풀의 범용 입출력으로 사용할 수 있다.

출력으로 설정한 후 해당 핀으로 HIGH 또는 LOW를 출력하기 위해서는 GPIOA_BSRR(Port Bit Set/Reset Register) 레지스터를 사용하면 된다. GPIOA_BSRR 레지스터 역시 32bit 레지스터로 해당 핀으로 HIGH를 출력하기 위해서는 Set 부분인 0~15번 비트 중에서 해당 비트로 1을 출력하면 되고, LOW를 출력하기 위해서는 Reset 부분인 16~31번 비트 중에서 해당 비트로 1을 출력하면 된다. 예를 들어 포트 A의 5번 핀으로 HIGH를 출력하기 위해서는 GPIOA_BSRR 레지스터의 5번 비트를 1로 설정하고, LOW를 출력하기 위해서는 GPIOA_BSRR 레지스터의 21번 비트를 1로 설정하면 된다.



bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
읽기/쓰기	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
읽기/쓰기	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

그림 5-4 **GPIOA_BSRR 레지스터 구조**

GPIOA_CRL 레지스터와 GPIOA_BSRR 레지스터를 통해 포트 A의 5번 핀에 연결된 LED를 제어할 수 있다는 것을 알았다. 하지만 레지스터의 이름인 GPIOA_CRL과 GPIOA_BSRR은 레지스터에 할당된 메모리 주소를 나타낸다는 점을 기억해야 한다. GPIOA_CRL과 GPIOA_BSRR은 코드 작성의 편의를 위해 정의한 이름이므로 레지스터를 조작하기 위해서는 실제 메모리 번지를 알고 있어야 한다. 레지스터에 해당하는 메모리의 번지는 데이터시트에서 확인할 수 있으며 GPIOA_CRL 레지스터의 주소는 `0x40010800`, GPIOA_BSRR 레지스터의 주소는 `0x40010810`이다.

코드 5-1은 레지스터 주소를 사용하여 레지스터를 조작함으로써 포트 A의 5번 핀에 연결된 LED를 0.5초 간격으로 점멸시키는 코드의 예다. 코드 5-1에서는 LED를 제어하는 과정에 집중하기 위해 시간 지연을 위한 함수로 Mbed의 `wait` 함수를 사용하였다. 코드 5-1을 입력하고 컴파일한 후 NUCLEO-F103RB 보드에 업로드하면 0.5초 간격으로 LED가 점멸하는 것을 확인할 수 있다.

코드 5-1 레지스터 주소를 사용한 LED 점멸(1단계)

```
#include "mbed.h"

int main() {
    // 포트 A의 5번 핀을 제어하기 위해 사용하는 마스크
    unsigned int mask_pin5 = 1 << 5;            // Set(ON), Reset(OFF) 제어
    unsigned int mask_pin5_1 = 0b11 << 20;      // MODE5
    unsigned int mask_pin5_2 = ~(0b11 << 22);   // CNF5

    unsigned int *port_A_addr = (unsigned int*)0x40010800;
    // 포트 A의 5번 핀을 출력으로 설정
    *port_A_addr |= mask_pin5_1;                // 최대 50MHz 속도의 출력 모드
    *port_A_addr &= mask_pin5_2;                // 범용 푸시풀 출력 모드

    unsigned int *port_A_set_reset_addr = (unsigned int*)0x40010810;

    while (true) {
        // 포트 A의 5번 핀으로 HIGH값 출력: Set
        *port_A_set_reset_addr |= mask_pin5;
        wait(0.5);

        // 포트 A의 5번 핀으로 LOW값 출력: Reset
        *port_A_set_reset_addr |= (mask_pin5 << 16);
        wait(0.5);
    }
}
```

코드 5-1에서는 레지스터의 값을 변경하기 위해 레지스터의 주소를 사용하고 있다. 하지만 포트 A의 핀을 출력으로 설정하기 위해 사용한 레지스터의 이름인 GPIOA_CRL은 기억이 날지도 모르겠지만 GPIOA_CRL 레지스터의 주소인 0x40010800을 기억하기란 쉬운 일이 아니다. 따라서 레지스터의 주소를 직접 사용하는 경우보다는 레지스터 주소를 계산할 수 있는 상수를 통해 레지스터에 접근하는 것이 일반적이다. 또한 보다 편리한 사용을 위해 연관된 레지스터들을 구조체로 정의하여 사용한다. stm32f103xb.h[1] 파일을 살펴보면 범용 입출력 핀을 위한 상수와 구조체 정의를 찾아볼 수 있다. 코드 5-2는 PA_5 핀에 연결된 LED를 점멸시키는 데 필요한 구조체와 상수를 요약한 것이다.

코드 5-2 하드웨어 종속적인 데이터 구조 및 상수 정의

```
typedef struct {
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;

#define PERIPH_BASE                      0x40000000U
#define APB2PERIPH_BASE                  (PERIPH_BASE + 0x00010000U)
#define GPIOA_BASE                       (APB2PERIPH_BASE + 0x00000800U)
#define GPIOA                            ((GPIO_TypeDef *)GPIOA_BASE)
```

코드 5-2의 정의를 사용하면 코드 5-1에서와 같이 레지스터의 번지를 사용하지 않고 상수와 구조체를 통해 포트 A의 핀들을 제어할 수 있다. 코드 5-1과 같은 기능을 하면서 코드 5-2의 정의를 사용하여 수정한 코드의 예가 코드 5-3이다.

코드 5-3 구조체와 상수를 사용한 LED 점멸(2단계)

```
#include "mbed.h"

int main() {
    // 포트 A의 5번 핀을 제어하기 위해 사용하는 마스크
    unsigned int mask_pin5 = 1 << 5;            // Set(ON), Reset(OFF) 제어
    unsigned int mask_pin5_1 = 0b11 << 20;      // MODE5
```

1 https://github.com/ARMmbed/mbed-os에서 소스 코드를 다운받아 D:에 압축을 해제하였다고 가정하였을 때 해당 파일의 위치는 D:\mbed-os-master\targets\TARGET_STM\TARGET_STM32F1\TARGET_NUCLEO_F103RB\device가 된다. 이후 파일의 위치 역시 D:에 압축을 해제하였다고 가정한 경우의 파일 위치다.

```
    unsigned int mask_pin5_2 = ~(0b11 << 22);    // CNF5

    // 포트 A의 5번 핀을 출력으로 설정
    GPIOA->CRL |= mask_pin5_1;                   // 최대 50MHz 속도의 출력 모드
    GPIOA->CRL &= mask_pin5_2;                   // 범용 푸시풀 출력 모드

    while (true) {
        // 포트 A의 5번 핀으로 HIGH값 출력, 하위 바이트
        GPIOA->BSRR |= mask_pin5;
        wait(0.5);

        // 포트 A의 5번 핀으로 LOW값 출력, 상위 바이트
        GPIOA->BSRR |= (mask_pin5 << 16);
        wait(0.5);
    }
}
```

5.3 하드웨어 독립적 프로그래밍

코드 5-3은 코드 5-1과 비교하면 간단하고 편리하지만 한 가지 문제점이 있다. 바로 코드 5-3에서 사용한 구조체의 이름은 STM32F103RBT6 마이크로컨트롤러에서만 사용할 수 있다는 점이다. 딱히 문제점으로 보이지 않을 수도 있지만, Mbed의 장점 중 하나는 다양한 제조사에서 제작한 Mbed 보드를 공통의 인터페이스를 사용하여 제어할 수 있다는 점이다. 코드 5-3은 STM32F103RBT6의 레지스터 구조에 맞게 정의된 자료 구조와 상수를 사용하므로, 즉 하드웨어 종속적이므로 다른 회사에서 제작한 마이크로컨트롤러 또는 같은 회사에서 제작한 다른 마이크로컨트롤러에는 사용할 수 없다. 코드 5-3은 레지스터의 주소를 기억하지 않아도 되는 편리함은 있지만, 여전히 하드웨어 종속적인 코드에 해당한다. 따라서 다양한 Mbed 보드들을 공통의 인터페이스로 제어할 수 있도록 Mbed에서는 하드웨어 독립적인 층을 추가로 정의하고 있다.

gpio_object.h 파일과 gpio_api.c 파일[2]을 살펴보면 코드 5-2에서 정의한, `GPIO_TypeDef` 구조체를 사용하여 하드웨어와 독립적인 새로운 구조체 `gpio_t`와 새로운 구조체를 사용하는 데이터 입출력 함수가 정의되어 있다.

2 D:\mbed-os-master\targets\TARGET_STM