【 기술 노트 18 】

# AVR 및 AVR-GCC를 사용할 때의 유의 사항

AVR이 뜨고 있다. 1984년에 설립된 신생 반도체 회사인 Atmel에서 1997년에 처음 발표하여 불과 7년 정도의 짧은 역사를 가지고 있는 AVR이 시장에서 기라성같은 선배들을 속속 물리치고 강세를 보이는 괴력을 발휘하고 있다. 이는 마치 1976년에 마이크로프로세서 시장에 혜성처럼 등장하여 Z80으로 돌풍을 일으켰던 Zilog사를 연상케 한다. 그러나 Z80은 당시 불과 몇 개 정도의 경쟁사를 상대로 했고 시대적으로 마이크로프로세서 기술이 조금씩 자리를 잡아가고 있던 걸음마 단계 시절이었으므로 얼마든지 돌풍이 가능할 수도 있었고 핵심 설계자들이 Intel사의 출신들이라는 점을 감안하면 쉽게 상황이 이해되는데 비하여, Atmel은 마이크로프로세서 기술이 활짝 꽃을 피운 치열한 기술 경쟁의 시대에 뒤늦게 후발주자로 나타나 무림의 강호들을 상대로 돌풍을 일으키는 것을 감안하면 그 위력은 그 옛날의 Zilog보다도 더 강하다고 할 수 있다.

AVR이 이와 같이 짧은 기간에 사용자들에게 선호되고 널리 사용될 수 있게 된 것은 무엇보다도 Atmel의 기술적 강점인 플래시 메모리를 MCU 안에 프로그램 메모리로 내장하고 ISP(In-System Programming)라는 간편한 방법으로 이를 라이팅할 수 있기 때문으로 분석된다. 초기에는 이 플래시 메모리를 1,000번까지 재사용하는 것을 보증하였으나, 현재는 대부분의 모델이 이를 10,000번까지 재사용할 수 있는 사양을 가지고 있다. 프로그래밍 장비도 처음에는 Atmel사의 직렬포트용 AVR ISP만을 사용해야 했지만, 나중에는 사용자가 쉽게 자작할 수 있는 병렬포트용이 개발되어 완전 무료가 되었다. 이는 이 소자를 사용하는데 따른 하드웨어 장비의 비용을 덜어주게 되어 아마추어들에게 상당히 매력있는 요소로 작용하였으며, 이것은 이러한 특별한 메모리 구조나 라이팅 방법을 가지고 있지 않은 8051 시리즈 등의 경쟁 제품에 대하여 뚜렷한 강점이었고, 이와 유사한 고속처리 성능을 가지는 RISC 방식이지만 OTP(One-Time Programming)를 기반으로 하여 대부분 1회만 프로그래밍이 가능한 PIC 시리즈에 대하여도 분명한 비교 우위를 가지는 것이었다.

또 한가지 AVR의 성공 비결은 Atmel사의 무료 어셈블러와 GNU 그룹의 무료 C컴파일러인 AVR-GCC의 사용이다. 통상적으로 수백만원에 이르는 개발 소프트웨어를 구입하지 않고도 무료로 사용할 수 있다는 것은 분명 대단한 매력이다. Atmel사는 AVR을 처음 개발할 때부터 IAR사의 C컴파일러 개발자들과 협력관계를 유지하면서 AVR을 C언어를 처리하는데 적합한 구조로 설계하기 위하여 노력하였으며 응용노트와 같은 각종 기술자료에도 IAR C컴파일러를 기준으로 설명하고 있는 것을 보면 이는 Atmel사의 의도와는 다른 결과로 보인다. 초보적인 사용자의 편의를 위하여 매크로 어셈블러는 AVR Studio에 포함시켜 무료로 제공하고 C컴파일러는 IAR C컴파일러를 사용하도록 유도하였지만 결과적으로는 뜻

하지 않게 AVR-GCC의 출현으로 사용자들은 C컴파일러까지 공짜로 사용할 수 있게 된 것으로 추측된다.

어쨌든 이렇듯이 AVR은 하드웨어 개발 툴이나 소프트웨어 개발 툴을 무료로 사용할 수 있게 되므로써 아마추어들에게 AVR은 곧 무료라는 등식으로 간주되는 분위기가 형성되었고 이는 당연히 시장에서 AVR의 폭발적 성공으로 이어지는 결과로 나타났다. 누구에게나 우선 공짜라는 것은 분명 매력적인 것이다...!

그러나, 짧은 시간에 급속하게 발전하여 온만큼 AVR의 사용이나 AVR-GCC의 사용에 필요한 기술자료들은 상당히 부족한 편이다. 특히 AVR-GCC는 모든 것이 비상업용으로 진행된 탓인지 사용자 매뉴얼 등이 매우 부족하거나 부실하여 사용자들이 어려움을 겪고 있다. 이에 본 필자는 AVR-GCC를 기본으로 사용하는 "AVR ATmega128 마스터" 책을 출판하면서 책에 넣지 못한 AVR 또는 AVR-GCC 사용시의 유의 사항을 기술노트로 정리하고자 한다.

## 1. AVR 사용시의 유의 사항

### (1) AVR의 여러 가지 모델 사이에 호환성이 있다고 생각하지 마라.

AVR은 현재 기본적으로 ATtiny 패밀리, AT90 패밀리, ATmega 패밀리 등 3가지의 소계열로 나누어지고 있으며 이들 각각의 소계열에는 다시 많은 모델들이 있다. 그러나, 최근에 들어서 Atmel사는 AT90 패밀리를 점차로 축소하고 저가형인 ATtiny 패밀리와 고성능형인 ATmega 패밀리에 주력하는 인상을 주고 있다. 이 때문에 AVR의 역사는 짧지만 그동안에 벌써 많은 모델들이 단종되었다.

사용자 입장에서는 하나의 AVR 모델을 사용하다가 기능 또는 성능의 향상을 위하여 다른 모델로 업그레이드하거나, 또는 사용하던 모델이 단종되어 다른 모델로 교체하는 경우가 많다. 그러나, Atmel은 새로운 모델을 개발하면서 끊임없이 개선 또는 변화를 시도해 왔기 때문에 기능적으로 100% 호환되는 모델은 거의 없으므로 사용자는 기존의 소프트웨어를 변경하지 않고 AVR 모델을 변경하여 사용하는 것을 기대하면 안된다. 특히 이에 관해 Atmel은 매우 일관성이 없는 정책을 취해온 것으로 분석된다. 즉, 어느 모델의 경우에는 기존의 모델과 호환성을 유지하기 위하여 애쓴 흔적이 있고, 어느 모델의 경우는 전혀 기존 모델과의 호환성을 고려하지 않고 기능을 크게 변경하기도 하였다. 특히 I/O 레지스터가 많이 달라졌고, 같은 이름을 사용하면서도 레지스터 내부의 각 비트 기능이 달라진 경우도 있으며, 같은 기능을 유지하고 있는데 레지스터의 이름만 변경된 경우도 있는 등 종잡을 수가 없다. 따라서, 사용자가 AVR 모델을 변경하는 경우에는 기존의 모델의 의식하지 말고 새로운 모델을 완전히 다시 익혀서 사용한다는 마음으로 꼼꼼하게 데이터 시트를 읽어보는 것이 좋다. 그래야만 예상하지 않았던 엉뚱한 문제로 오랜 시간을 고생하는 일을 사전에 예방할 수

있다.

(2) 데이터 시트를 철저히 읽어라.

AVR의 여러 가지 모델들을 사용하다 보면 데이터 시트 이외의 기술자료를 구하는 것이 어려운 경우가 많다. 비교적 널리 쓰이는 모델은 기술서적이 나와있는 경우도 있으나 많은 모델은 책도 없다. 사실 책이 있어도 그 내용이 별로 도움이 안되는 경우도 많다.

이럴 때 기댈 곳은 결국 데이터 시트밖에 없다. 데이터 시트를 능가하는 기술자료는 없다. 간혹 어느 분야에 따라서는 Atmel사에서 응용노트로 보충자료를 제공하는 경우도 있으나 이는 지극히 한정적인 경우에 불과하다. 그러나, 이 데이터 시트는 물론 영어로 되어 있고, 대체로 300페이지를 넘는 방대한 분량이다. 초보자들은 이것만으로도 충분히 겁먹을 수 있는 내용이다.

더구나, 마이크로프로세서에 어느 정도의 경험을 가진 사람도 Intel사의 데이터 시트에 익숙하다고 하여 AVR 데이터 시트를 쉽게 읽을 수 있는 것은 아니라는 것을 알게 된다. 실제로 읽어보면 Atmel의 AVR 데이터 시트는 상당히 읽기 까다롭고 어려운 표현도 많으며 내용의 오류도 적지 않다.

그러나, 그렇다고 하더라도 데이터 시트는 AVR을 익히는데 피해가서는 안되는 길목이다. 데이터 시트를 제대로 읽어서 이해하는 것만이 그 AVR 모델을 잘 사용할 수 있는 비결이다. MCU에 관한 문제가 발생하면 언제나 데이터 시트에서 해결책을 찾아야 한다.

(3) 16비트 레지스터의 액세스에는 원칙이 있다.

모든 8비트 마이크로콘트롤러가 그렇듯이 AVR에서도 몇가지의 16비트 I/O 레지스터를 가지고 있고, 이 16비트 레지스터는 8비트씩 2차례로 나누어 읽거나 쓴다. 그러나, 다른 마이크로콘트롤러와 달리 AVR에서는 반드시 이 16비트 레지스터를 액세스하는데 지정된 순서가 있다. 즉, 16비트 레지스터를 구성하는 2개의 8비트 레지스터를 액세스할 때 리드 동작은 반드시 하위 바이트를 먼저 리드하고 상위 바이트를 나중에 리드하며, 라이트 동작은 반대로 상위 바이트를 먼저 라이트하고 하위 바이트를 나중에 라이트해야 한다. 이를 어셈블리 루틴으로 예를 들면 다음과 같다.

((예)) 16비트 리드의 경우

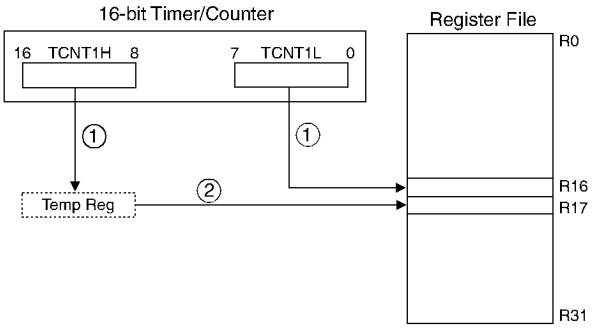
```
IN      R16,TCNT1L          ; R16 ← TCNT1L(low byte first)
IN      R17,TCNT1H          ; R17 ← TCNT1H
```

((예)) 16비트 라이트의 경우

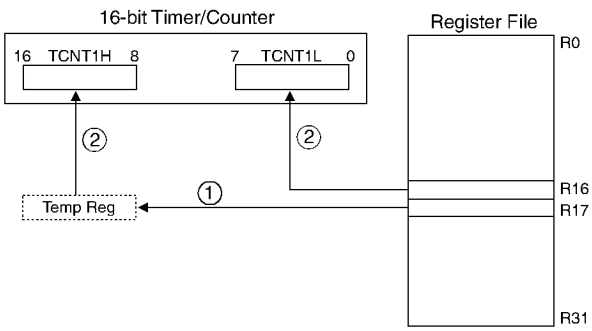
```
LDI     R17,high(10000)     ; R17:R16 ← 10000
LDI     R16,low(10000)
```

```
OUT     TCNT1H,R17          ; TCNT1H ← R17(high byte first)
OUT     TCNT1L,R16          ; TCNT1L ← R16
```

그 이유는 실제 레지스터에서 데이터를 읽거나 쓰는 것이 동시에 16비트로 처리될 수 있도록 버퍼인 임시 레지스터를 사용하기 때문이며, 이를 그림으로 설명하면 <그림 18.1>과 같다. 이러한 16비트 레지스터에는 타이머/카운터 1과 3의 레지스터나 A/D 컨버터의 레지스터 등이 있으며, 이 원칙을 지키지 않고 16비트 레지스터를 액세스하면 임시 레지스터에 남아있던 엉터리 데이터로 처리된다는 점을 기억해야 한다.




(a) 리드의 동작 순서




(b) 라이트의 동작 순서

<그림 18.1> 16비트 레지스터의 리드 및 라이트 동작

## 2. AVR-GCC 사용시의 유의 사항

### (1) AVR-GCC 컴파일러의 신뢰성을 의심하지 마라.

흔히 초보 수준의 AVR 사용자들이 AVR-GCC로 프로그램하다가 잘 모르는 에러가 발생하면 AVR-GCC의 신뢰성을 의심하고 너무나도 가볍게 컴파일러 버그 운운하는 것을 볼

수 있다. 어리석기 짝이 없는 일이다.

AVR-GCC는 상업용 소프트웨어 못지 않은 성능과 신뢰성을 가지고 있다. 상업용이든 셰어웨어든 프리 소프트웨어든 세계적으로 널리 사용되는 소프트웨어에서 내가 버그를 발견할 수 있는 행운은 인생에서 그리 자주 있는 일이 아니다. 그럴 확률은 선동렬 선수가 코리안 시리즈 7차전에서 9회말 투아웃에 대타로 나와서 초구에 역전 만루홈런을 칠 확률과 비슷하다고 보면 된다...^.^ 프로 소프트웨어 개발자들은 초보 사용자가 어찌어찌 하다가 에러를 발견할 만큼 그렇게 허술하게 제품을 내놓지 않는다.

내가 작성한 프로그램에 문제가 있다고 생각하지 않고 AVR-GCC를 의심할수록 해결책을 찾아내기는 어려워진다. AVR-GCC는 충분히 믿을만한 컴파일러이다. 100% 내 잘못이라는 전제하에 문제 해결을 위하여 접근하라.

(2) AVR-GCC에서 컴파일 에러가 발생하면 무조건 에러가 발생한 행만 체크하지 말고 좀더 넓게 보라.

필자의 경험으로 볼 때 AVR-GCC의 한가지 결점이라면 에러 메시지가 애매한 경우가 많다는 것이다. 얼핏 메시지 내용만을 봐서는 이게 왜 에러가 발생하였다는 것인지 알기 어려운 경우가 많다.

그런 경우에는 에러가 발생한 행만 집중적으로 살펴보지 말고 그 앞의 행들을 조사하여 보라. 앞의 행에서 잘못한 사항이 뒤 줄에 에러로 나타나는 경우가 많기 때문이다. 예를 들어

```
 1 : #include <avr/io.h>
 2 :
 3 : int main(void)
 4 : {
 5 :   DDRB = 0xFF;
 6 :
 7 :   while(1)
 8 :     { PORTB = 0x55
 9 :       PORTB = 0xAA;
10 :     }
11 : }
```

라는 프로그램 test.c를 AVR-GCC로 컴파일하였을 경우

```
test.c : 9 : error : called object is not a function
```

이라는 에러 메시지가 발생하게 된다. 이를 마무리 해석해봐야 헷갈리기만 한다. 우선 9번 행에는 에러가 없을 뿐만 아니라 누가 무엇을 호출했다는 것이며 호출된 것이 함수가 아니라니 이게 무슨 말인지 머리가 빙빙 돈다.

그러나, 이것은 그 앞 줄인 8번 행을 보면 금방 에러라는 것을 알 수 있다. 8번 행이 세미콜론(;)으로 종료처리가 안되어 8번 행이 9번행과 이어진 것으로 인식되고 그렇게 되니 문법 에러가 되는 것이다. 굳이 해석하자면 PORTB에 데이터를 출력하는 함수가 잘못되었다는 것이다. PORTB가 그 앞줄의 0x55와 붙어있으니까... 이 명령은 함수처럼 보이지 않으나 함수다. 현재의 AVR-GCC는 IAR C컴파일러 형식의 이런 표현을 함께 사용하지만 이는 원래 outp(0xAA, PORTB)처럼 함수로 사용했던 것이다.

(3) 컴파일이나 링크에서 에러가 없었는데 원하는 실행 결과가 나오지 않으면 프로그램을 디버깅하기 전에 AVR의 데이터 시트를 다시 보고 MCU의 기능을 먼저 살펴보라.

AVR의 기능을 제대로 이해하지 않고 엉뚱하게 프로그램한 것을 두고 소스만 들여다 보아야 해결될 턱이 없다. 실행 결과가 뜻대로 나오지 않으면 프로그램을 의심하기 전에 내가 AVR을 제대로 이해하고 있는지를 의심하라.

물론 이것도 넓은 의미에서 보면 프로그램의 오류이다. 그러나, 프로그램 자체의 오류가 아니라 AVR을 잘못 이해하여 발생한 프로그램 작성 오류이므로 이 해결책은 AVR의 데이터 시트를 다시 읽어 해결해야 한다. 혹시라도 이를 두고 프로그램의 버그(bug)라고는 하지 마라. 이런 것은 버그가 아니다. 내가 당연히 알고 있는 사실을 단지 프로그램 소스에서 실수로 잘못 처리한 것은 버그일 수 있지만 내가 몰라서 프로그램에 오류를 범한 것은 버그라고 할 수 없다.

필자의 경우 이런 오류를 찾아내는데 좀더 도움이 되는 방법은 프로그램 소스를 프린터로 인쇄하여 놓고 보는 것이다. 아무래도 컴퓨터 모니터 상에서 소스를 위아래로 스크롤하면서 보는 것은 산만하다. 쉽게 눈에 띄지 않는 오류를 찾는데는 반드시 소스를 프린터로 인쇄하여 놓고 보라. 그러면서 데이터 시트를 읽고 소스를 한 줄씩 확인해나가도록 하라. 이때 또한 꼭 필요한 것이 빨간색 펜과 파란색 펜이다. 본 필자는 빨간색과 파란색의 0.5mm 중성펜을 1년에 각각 5~6개씩은 사용한다. 이런 펜조차 일본제 찾지말고 국산을 사용하시라. 동아에서 나오는 중성펜도 얼마든지 좋다. 흡연을 즐기시는 분은 이런 때 담배 한 개피에서 여유를 찾아도 좋다. 넓게 볼 때는 넓게 보아야 하고, 깊게 볼 때는 깊게 보아야 한다. 프린터로 인쇄된 소스를 볼 때는 넓게 보는 것과 깊게 보는 것이 모두 용이하다. 그러나, 모니터 상에서 소스를 보는 것은 넓게 보는데도 불편하고 깊게 보는데도 불편하다.

(4) 에러 메시지도 없었고 AVR을 사용하는데도 의심의 여지가 없으면 본격적으로 소스를 파고 들어라.

이러한 경우는 이제 프로그램의 오류를 찾기 위한 본 게임에 들어가는 것이다. 어느 소프트웨어나 그렇듯이 모든 오류를 잡아내주지는 못한다. 미묘한 오류는 에러로 표시되지 않

으므로 사용자가 찾아내야 한다. 우선 프로그램을 전체적으로 요약하여 의심스러운 부분을 압축해야 한다. 어느 부분을 새로 추가하고 나서 문제가 발생하였다면 문제가 되는 부분을 쉽게 찾아낼 수 있다. 그러나, 그렇지 않은 경우에는 의심스러운 부분을 정확히 추정하는 눈썰미가 중요하다. 의심스러운 부분을 주석으로 처리하여 실행되지 않도록 제외한 후에 프로그램을 실행시켜 보는 것도 좋은 디버깅 방법이다.

이런 오류를 찾을 때 가장 도움이 되는 것은 역시 AVR-GCC의 매뉴얼이다. 그러나, 유감스럽게도 이는 매우 부실하다. 가장 도움이 되는 것은 avr-libc Reference Manual이다. 이 문헌의 내용은 반드시 깊이 있게 읽어야 한다. 이밖에 AVR용이 아닌 일반 GCC에 대한 매뉴얼도 있지만 너무나도 방대하여 사실 읽어볼 용기가 나지 않는다. 필자도 이것은 매우 부분적으로만 읽었다.

그 다음에 또 사용할 수 있는 방법은 인터넷에서 질문/답변 게시판을 검색하는 것이다. 매뉴얼이 충실하지 않을수록 질문/답변의 게시판이 중요하다. 나처럼 답답한 경우를 경험한 선배들이 이미 질문과 해결책을 흔적으로 남기고 갔을 가능성이 높다. 나는 그것을 찾아서 읽어보기만 하면 되는 것이다. 이때 중요한 것이 눈썰미 있게 키워드를 선정하여 검색하는 것이다. 키워드를 잘못 사용하면 내게 필요한 내용이 있어도 찾아내지 못한다. 그렇다고 수천개의 게시물을 모두 하나씩 읽어나갈 수는 없는 노릇이다. 이러한 유용한 인터넷 게시판으로는 우키의 AVR World 홈페이지(http://micro.new21.org/avr/), 황해권 님의 테라뱅크 홈페이지(http://www.terabank.co.kr/), 하이텔의 디지털동호회 홈페이지(http://club.hitel.net/dig), 그리고 필자의 홈페이지 “윤교수의 마이크로프로세서 월드” 등이 있고, 외국의 AVR 관련 사이트로는 http://www.avrfreaks.net/ 가 유용하다.

한가지 더 추천할 수 있는 심도있는 디버깅 방법으로는 컴파일된 어셈블리 소스를 보는 것이다. 컴파일러가 어떤 처리를 하는지 의심스러울 때는 이 방법이 가장 강력하다. 이는 한마디로 AVR-GCC 컴파일러의 뱃속을 들여다보는 것이라고 말할 수 있다. 이를 보면 컴파일러가 소스를 어떻게 번역하는지, 그래서 AVR이 어떻게 동작할지를 정확히 살펴볼 수 있으므로 웬만한 문제는 그냥 다 잡힌다. 이 어셈블리 소스는 컴파일 후에 생성되는 .lst 파일을 보면 된다. 이 파일은 방대한 길이를 가지므로 원하는 부분을 빨리 찾아내어 살펴보는 요령이 필요하다.

이 방법을 많이 사용하면 더불어 어셈블리 프로그래밍 실력이 크게 향상된다. 그리고 물론 C언어 소스를 작성하는 효율적인 방법도 터득하게 된다. 실제로 AVR-GCC는 매우 효율적인 어셈블리 코드를 생성한다. 필자도 ATmega128의 어셈블리 언어로 작성한 것보다 C언어로 작성한 프로그램이 오히려 실행속도가 빠르게 되는 것을 경험한 적이 있다. 그만큼 AVR-GCC의 컴파일 능력은 뛰어나다.

(5) 전역변수를 사용할 때는 주의가 필요하다.

전역변수는 SRAM에 저장되어 있다가 이를 사용할 때마다 레지스터로 읽혀지며 사용이 끝나면 다시 SRAM에 저장되므로 처리속도가 늦은 변수이므로 꼭 필요한 경우가 아니면 가급적 사용을 피하는 것이 좋다. 이에 비하여 지역변수는 레지스터로 처리되므로 항상 처리속도가 빠르다. 다른 MCU에서는 지역변수를 스택에 저장하므로 전역변수에 비하여 그다지 유리하지 않은 경우도 있지만, AVR의 경우에는 C언어에 적합한 구조를 가지도록 여러 가지 사항이 고려되어 설계되었는데 범용 레지스터가 32개나 존재하는 것도 그중의 하나이다. 이 때문에 AVR은 상당한 정도의 지역변수를 레지스터로 처리하는데 아무 문제가 없다.

그런데, 전역변수에는 이러한 사항 이외에 또다른 문제가 발생할 수 있다. 즉, 컴파일러가 이를 번역할 때 이를 변수로 처리하지 않고 그 상황에서 이 변수가 갖는 값에 해당하는 상수로 처리하는 사례가 있는 것이다. 이렇게 되면 여러 함수에서 이 변수를 공유하여 사용하는데 문제를 야기한다. 이는 상황에 따라 상당히 다르고 최적화 옵션을 어떻게 지정하는지에 따라서도 영향을 받는다. 이러한 문제는 특히 인터럽트 서비스 루틴 함수와 그 밖의 함수 사이에 공유하는 전역변수의 경우에 발생하는 경우가 많다.

이를 방지하려면

```
unsigned char count1, count2;
```

와 같은 전역변수를

```
volatile unsigned char count1, count2;
```

처럼 volatile 속성으로 선언해 주면 된다. 개념적으로 이 volatile은 const와 반대의 의미로 생각하면 쉽다.

(6) 최적화 옵션을 올바르게 지정하라.

AVR-GCC에서는 최적화 옵션을 0~3레벨까지 지정할 수 있으며 이 숫자가 커질수록 최적화의 정도가 높아진다. -O0은 최적화를 전혀 수행하는 않는 것이며, -O3은 가장 높은 최적화를 수행하는 것이다. 각 단계별로 어떤 내용의 최적화가 수행되는지는 GCC 매뉴얼에 설명되어 있다.

그러나, 여기서 주의할 사항이 있다. 최적화라는 것은 프로그램의 실행속도를 높이는 측면과 실행 코드 사이즈를 작게 하는 측면으로 생각할 수 있다. 최적화가 0, 1, 2로 높아질수록 이 2가지를 점점 더 높은 단계로 최적화하게 된다. 그러나, 최적화 옵션을 -O3으로 지정하면 -O2에 대하여 -finline-functions 옵션과 -frename-registers 옵션이 추가로 동작한다. 이중에서 -finline-functions 옵션은 함수를 호출할 때 이를 서브루틴처럼 처리하지 않고 매

크로 확장의 형태로 처리하는 기능이다. 따라서, 최적화 옵션을 -O3으로 지정하면 실행 속도는 빨라지지만 함수 호출이 많은 프로그램의 경우에는 실행 코드 사이즈가 엄청나게 증가하는 결과가 된다.

따라서, 프로그램의 실행 코드 사이즈를 증가시키지 않으면서 최적화의 정도를 가장 높게 하는 것은 옵션을 -O3이 아니라 -O2로 지정하는 것이다. 특별히 실행 코드의 사이즈를 가장 작게 하는 최적화 옵션으로는 -Os를 사용한다.

AvrEdit V3.6은 디폴트로 -O3으로 설정되어 있다. 사용자가 이를 다른 옵션으로 수정하려면 설정→환경설정→컴파일러 메뉴에서 Compiler flags라는 행을 수정하면 된다.

(7) 대용량의 상수 데이터 테이블을 프로그램 메모리에 저장하는데는 정확한 처리가 요구된다.

용량이 큰 상수 데이터를 SRAM에 저장하는 것은 불리하다. ATmega128의 경우 내부 SRAM은 4KB에 불과하고 프로그램 메모리인 플래시 메모리는 128KB나 되기 때문에 가능하면 대용량 상수 데이터는 프로그램 메모리에 저장해야만 SRAM이 메모리 변수나 스택으로 사용될 수 있도록 여유를 가지게 된다.

하지만, 프로그래머의 의도와는 달리 자칫하면 이러한 상수 데이터가 SRAM으로 저장되므로 사용 가능한 SRAM이 줄어들고 그 결과로 뜻하지 않게 스택 오버플로우가 발생하여 프로그램이 오동작함으로써 고생하는 사례가 보고되고 있다.

예를 들어 보자. 다음 프로그램 test1.c는 D/A 컨버터로 출력할 100바이트의 상수 데이터를 SRAM에 변수로 저장한 예이다. 그 결과 이를 AvrEdit V3.6에서 최적화 옵션을 -O3으로 주고 컴파일하면 플래시 메모리 용량은 1612바이트이고, SRAM 용량은 134바이트로 된다.(그러나, 최적화 옵션을 -Os로 주고 컴파일하면 플래시 메모리 용량은 650바이트이고, SRAM 용량은 134바이트로 된다.)

<프로그램 test1.c> 100개의 바이트 데이터를 SRAM에 메모리 변수로 저장한 경우

```
#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

int main(void)
{ unsigned char i;
  unsigned char tri_table[] = {                  // triangular wave data table
    0x80, 0x84, 0x88, 0x8C, 0x90, 0x94, 0x98, 0x9C, 0xA0, 0xA4,
    0xA8, 0xAC, 0xB0, 0xB4, 0xB8, 0xBC, 0xC0, 0xC4, 0xC8, 0xCC,
    0xD0, 0xD4, 0xD8, 0xDC, 0xE0, 0xE4, 0xE0, 0xDC, 0xD8, 0xD4,
    0xD0, 0xCC, 0xC8, 0xC4, 0xC0, 0xBC, 0xB8, 0xB4, 0xB0, 0xAC,
    0xA8, 0xA4, 0xA0, 0x9C, 0x98, 0x94, 0x90, 0x8C, 0x88, 0x84,
    0x80, 0x7C, 0x78, 0x74, 0x70, 0x6C, 0x68, 0x64, 0x60, 0x5C,
    0x58, 0x54, 0x50, 0x4C, 0x48, 0x44, 0x40, 0x3C, 0x38, 0x34,
    0x30, 0x2C, 0x28, 0x24, 0x20, 0x1C, 0x20, 0x24, 0x28, 0x2C,
```

```
    0x30,0x34,0x38,0x3C,0x40,0x44,0x48,0x4C,0x50,0x54,
    0x58,0x5C,0x60,0x64,0x68,0x6C,0x70,0x74,0x78,0x7C };

  MCU_initialize();                          // initialize MCU
  Delay_ms(50);                              // wait for system stabilization
  LCD_initialize();                          // initialize text LCD module

  LCD_string(0x80,"  DAC0800 D/A   ");       // display title
  LCD_string(0xC0,"Triangular Wave ");

  while(1)
    { for(i=0; i<100; i++)                   // output D/A
        { PORTB = tri_table[i];
          Delay_us(10);
        }
    }
}
```

그러나, 이를 상수 데이터가 프로그램 메모리인 플래시 메모리에 저장되도록 수정하면 다음의 프로그램 test2.c와 같이 된다. 이를 AvrEdit V3.6에서 최적화 옵션을 -O3으로 주고 컴파일하면 플래시 메모리 용량은 1692바이트이고, SRAM 용량은 34바이트로 된다.(그러나, 최적화 옵션을 -Os로 주고 컴파일하면 플래시 메모리 용량은 726바이트이고, SRAM 용량은 34바이트로 된다.)

<프로그램 test2.c> 100개의 바이트 데이터를 프로그램 메모리에 상수로 저장한 경우

```
#include <avr/io.h>
#include <avr/pgmspace.h>
#include "c:\AvrEdit\Ok128c\Ok128.h"

prog_uchar tri_table[] = {                   // triangular wave data table
    0x80,0x84,0x88,0x8C,0x90,0x94,0x98,0x9C,0xA0,0xA4,
    0xA8,0xAC,0xB0,0xB4,0xB8,0xBC,0xC0,0xC4,0xC8,0xCC,
    0xD0,0xD4,0xD8,0xDC,0xE0,0xE4,0xE0,0xDC,0xD8,0xD4,
    0xD0,0xCC,0xC8,0xC4,0xC0,0xBC,0xB8,0xB4,0xB0,0xAC,
    0xA8,0xA4,0xA0,0x9C,0x98,0x94,0x90,0x8C,0x88,0x84,
    0x80,0x7C,0x78,0x74,0x70,0x6C,0x68,0x64,0x60,0x5C,
    0x58,0x54,0x50,0x4C,0x48,0x44,0x40,0x3C,0x38,0x34,
    0x30,0x2C,0x28,0x24,0x20,0x1C,0x20,0x24,0x28,0x2C,
    0x30,0x34,0x38,0x3C,0x40,0x44,0x48,0x4C,0x50,0x54,
    0x58,0x5C,0x60,0x64,0x68,0x6C,0x70,0x74,0x78,0x7C };

int main(void)
{ unsigned char i;

  MCU_initialize();                          // initialize MCU
  Delay_ms(50);                              // wait for system stabilization
  LCD_initialize();                          // initialize text LCD module

  LCD_string(0x80,"  DAC0800 D/A   ");       // display title
  LCD_string(0xC0,"Triangular Wave ");
```

```
    while(1)
      { for(i=0; i<100; i++)                    // output D/A
          { PORTB = pgm_read_byte(&tri_table[i]);
            Delay_us(10);
          }
      }
}
```

AVR-GCC에서 이와 같이 상수 데이터를 프로그램 메모리에 저장하려면 다음의 3가지를 명확히 처리해야 한다.

ⅰ) 프로그램 메모리를 처리하는 헤더파일 pgmspace.h를 인클루드해주어야 한다.
ⅱ) 상수 데이터를 반드시 전역변수처럼 함수의 밖에서 정의해야 한다. 이때 상수가 바이트 데이터라면 prog_char 또는 prog_uchar로 정의해야 한다.
ⅲ) 상수 데이터를 읽는 것은 반드시 pgm_read_byte() 함수를 사용해야 한다.

만약, 상수 데이터를 prog_char 또는 prog_uchar로 정의하더라도 이를 함수 내에서 정의하면 SRAM 변수로 처리되는데 유의하라. 또한, 함수의 밖에서 올바르게 데이터를 정의했더라도 pgm_read_byte() 함수를 사용하지 않고 그냥 일반적인 변수처럼 tri_table[i]로 액세스하면 올바르게 읽혀지지 않는다.

### (8) 수학함수를 지원하는 라이브러리 파일을 링크하려면 사용자가 지정해주어야 한다.

AVR-GCC에는 2개의 기본적인 라이브러리 파일이 있는데, 일반 C언어 기능을 처리하는 라이브러리 파일 libc.a와 수학함수를 처리하는 라이브러리 파일 libm.a가 그것이다. 이중에서 기본 라이브러리인 libc.a는 사용자가 일부러 링크하도록 지정하지 않아도 자동으로 링크되지만, sqrt()나 sin()과 같은 수학함수를 처리하는 라이브러리 파일 libm.a는 사용자가 필요할 때 링크하도록 지정하여야 한다. 링커 옵션에서 이 라이브러리 파일들은 lib를 떼어내고 그 다음의 파일명만을 사용하므로 수학함수 라이브러리는 m이 된다. 따라서, 라이브러리 파일을 지정하는 옵션이 -l이므로 라이브러리 파일 libm.a를 링크하도록 지정하려면 옵션을 -lm으로 주면 된다. AvrEdit V3.6에서는 이를 기본적으로 지정하지 않고 있으므로 사용자가 이를 링크하도록 지정하려면 설정→환경설정→컴파일러 메뉴에서 Linker flags라는 행에 ,-lm을 추가하면 된다.

### (9) AVR-GCC에도 스타트업 파일이 있다.

AVR-GCC에도 스타트업 파일(C startup file)이 있으며, 이는 사용자가 지정하지 않더라도 링커에 의하여 자동으로 링크된다. 이 스타트업 파일은 인터럽트 벡터 테이블을 설정하

고 스택 포인터를 RAMEND(ATmega128의 경우 SRAM의 끝번지인 0x10FF)로 설정하며, 워치독과 MCUCR 레지스터를 초기화하고 .data 세그먼트와 .bss 세그먼트를 초기화한 후에 main()으로 점프한다. 원래는 main() 함수를 호출하는 것이 바람직하지만 AVR에서는 main() 함수를 종료하고 리턴할 필요성이 없으므로 기본 제공되는 스타트업 파일에서는 스택을 절약하는 의미에서 점프로 처리하고 있다.

스타트업 파일은 각 MCU 모델에 따라 다르며 WinAVR에서는 이를 오브젝트 파일로만 제공하므로 사용자가 이를 수정하기는 어렵다. ATmega128의 경우에는 이 스타트업 파일이 crtm128.o라고 되어 있다. 이를 굳이 수정하려면 gcrt1.S 파일을 수정하여 다시 어셈블하면 되는 것으로 알려져 있다.

사용자 프로그램을 컴파일한 후에 생성되는 메모리 맵 파일을 보면 프로그램을 링크하는 순서는 스타트업 파일, 사용자 프로그램 파일, 기본 라이브러리 파일 libc.a, 수학함수 라이브러리 파일 libm.a의 순서로 처리되는 것을 볼 수 있다.

(10) main()도 함수이며, void 함수를 명확히 정의해야 한다.

C언어는 함수를 기본으로 하는 프로그래밍 언어이며, 당연히 main()도 하나의 함수이다. 그런데, AVR-GCC에서는 스타트업 파일에서 main() 함수를 호출하지 않고 점프하면서도 정작 main() 함수의 속성은 반드시 int형으로 지정하도록 하고 있다. 그러나, 이것은 컴파일러의 요구사항이므로 사용자는 이유를 따지지 말고 그대로 해주면 된다.

또한 대부분의 C언어에서는 리턴값도 없고 입력 파이미터도 없는 함수의 경우에 그냥 function() 이라고 정의하면 되었지만, AVR-GCC에서는 반드시 void function(void)와 같이 void를 생략하지 말고 적어주도록 되어 있다. void 함수인 것을 명시하지 않으면 컴파일러는 이를 디폴트로 int 함수로 처리하며, 이때 함수내에서 리턴값하는 값이 없으면 에러가 된다. 함수의 파라미터가 없을 경우에도 이를 void로 명시해주지 않으면 경고 메시지가 발생된다. 따라서, 이러한 사항들은 사용자의 입장에서 이유를 따지지 말고 C컴파일러의 원칙대로 처리해주면 된다.

main() 함수에서 또 한가지 주의할 사항은 main() 함수의 실행이 완료되고 나서 이를 이탈하지 않도록 하라는 것이다. main() 함수도 하나의 함수이므로 운영체제가 있는 시스템에서는 이것이 실행 종료되면 이를 호출한 곳으로 리턴하지만 AVR에서는 이런 상황을 가정하지 않는다. 따라서, main() 함수를 무한 루프 등으로 종료되지 않도록 처리해주지 않으면 컴파일러는 “control reaches end of non-void function”이라는 경고 메시지를 출력한다.

**【 참고문헌 】**

1. 윤덕용, AVR ATmega128 마스터, Ohm사, 2004