리눅스 I/O 포트 프로그래밍 미니 하우투

저자: Riku Saikkonen <*Riku.Saikkonen@hut.fi* `<mailto:Riku.Saikkonen@hut.fi>`> 최종 수정일: 1997년 3월 30일

번역: 이기동 <*kidong@opera.cse.cau.ac.kr* `<mailto:kidong@opera.cse.cau.ac.kr>`>

이 문서의 저작권은 Riku Saikkonen씨에게 있다. 자세한 것은 리눅스 하우투 저 작권 정보를 참조하기 바란다. 이 하우투 문서는 하드웨어 I/O 포트 프로그래밍과 사용자 모드의 인텔 x86에서 돌아가는 리눅스 프로그램에서 짧은 기간의 시간을 기다리는 것을 설명한다. 이 문서는 IO 포트 미니 하우투에서 나온 것이다. 수정하거나 첨가할 사항이 있다면, 주저하지 말고 전자메일을 보내 주기 바란다. (*Riku.Saikkonen@hut.fi* `<mailto:Riku.Saikkonen@hut.fi>`)... 이전 버전에서의 변동 사항 (1996년 8월 26일) 저자의 전자우편 주소가 바뀌었다. 필자가 생각한 대로 ioperm() 권한은 fork()를 통해서 전달되지 않는다. 많은 주제에 대한 정보를 다룬 웹페이지(URL)들이 추가됨 그 외에 다른 작은 변화

목 차

제 1 절 C 프로그램에서 I/O 포트, 일반적인 방법

I/O를 읽고 쓰기 위한 루틴은 /usr/include/asm/io.h에 있다. (또는 커널 소스 배 포본의 linux/include/asm-i386/io.h) 이 루틴은 인라인 매크로이기 때문에, #include <asm/io.h> 만으로도 충분하다; 다른 라이브러리를 추가할 필요는 없 다.

gcc(현재 최소한 2.7.2.1또는 그 아래 버전)의 제한 때문에, 여러분은 소스 코드 를 컴파일해야 한다. 최적화 옵션을 사용하여 (gcc -O1 or 더 높게), 또는 다른 방법으로 #including <asm/io.h> 앞의 #define extern을 비워 두어야 한다. 디버깅을 위해서, 여러분은 "gcc -g -O"를 사용할 수 있다. (최소한 요즘 나오 는 버전의

gcc에서는), 최적화 때문에 때때로 디버거가 조금 이상하게 동작할 수 도 있지만. 그것이 방해된다면, 다른 소스 파일에 나누어서 I/O 포트를 읽고 쓰 기 위한 루틴을 넣는다. 그리고 그 소스만 최적화 옵션을 주고 컴파일한다.

어떤 포트를 읽고 쓰기 전에, 반드시 프로그램에 해당하는 권한을 주어야 한다. 이는 여러분 프로그램의 시작 지점에서 가까운 어딘가 에서 (여러분이 어떤 I/O 포트를 읽고 쓰기 전에) ioperm(2) 함수를 호출함으로써 이루어진다. (unistd.h에 선언되어 있고, 커널에서 정의되어 있다). 문법은 ioperm(from,num,turn_on) 이다, 여기에서 from은 읽고 쓰려는 첫 번째 포트 번호이다. 예를 들어 ioperm(0x300,5,1); 은 포트 0x300에서 0x304 (모두 5개의 포트)에 엑세스할 권한 을 줄 것이다. 마지막 인자는 Boolean 값으로 프로그램에서 포트를 엑세스할 권 한을 줄 것인지(true (1)) 아니면 제거할 것인지 (false (0))를 정의한다. 떨어져 있는 포트 여러 개를 사용하기 위하여 ioperm을 여러 번 호출할 수 있다. 문법 에 대하여 자세한 사항은 ioperm(2) 매뉴얼 페이지를 참조하기 바란다.

ioperm() 호출은 여러분의 프로그램이 루트 권한을 가지고 있기를 요구한다; 따 라서 여러분은 프로그램을 루트 사용자로써 실행하던지 setuid 루트로 해결할 수도 있다. 여러분은 포트를 사용 가능하도록 ioperm()을 호출한 후에 루트 권한 을 버릴 수 있다. 여러분은 ioperm(...,0)로 포트 엑세스 권한을 반드시 버려야 할 필요가 있는 것은 아니다. 이는 프로그램이 끝나면, 자동적으로 이루어진다. setuid()가 루트가 아닌 사용자에게 ioperm()에서 주어지는 포트 엑세스 금지하 는 것이 아니고, fork()가 그러한 일을 한다.

Ioperm()은 0x000에서 0x3ff까지의 포트의 엑세스만 허용한다; 더 상위 번지에 있는 포트는, iopl(2) (이는 한번에 모든 포트에 대해 엑세스 할 수 있도록 한다) 을 사용할 필요가 있다. 프로그램에 모든 I/O 포트를 읽고 쓰는 기능을 제공하 기 위하여 레벨 인자 3(예를 들어 "iopl(3);")을 사용한다 (잘못된 포트를 읽고 쓰는 것은 컴퓨터에 심한 문제를 일으킬 수 있으므로 주의한다). 다시, 여러분은 iopl()을 호출하려면 루트 권한이 필요하다.

다음으로, 모든 포트를 실제로 읽고 쓰는데, 포트에서 한 바이트를 입력받기 위 해서는, inb(port);를 호출하면 되는데, 이는 받은 바이트를 리턴한다. 포트에 한 바이트를 출력하기 위해서는, outb(value, port);를 호출한다. (매개변수의 순서에 주의한다). 포트 x와 x+1(각각의 한 바이트에서 워드를 형성한다, 어셈블러 명령 의 INW와 같이) 에서 한 개의 워드(16비트)를 입력받기 위해서는, inw(x);를 호 출한다. 두 포트에 한 개의 워드를 출력하기 위해서는, outw(value,x)를 호출한 다. 여러분이 어떤 포트 명령(바이트/워드)을 사용해야 하는지 분명하지 않다면, 여러분은 아마도 inb() 와 outb()를 필요로 할 것이다. - 대부분의 장치들은 바 이트 순서로 포트를 읽고 쓰도록 설계되어 있다. 모든 포트 명령은 최소한 실행 하는데 1 마이크로 초가 걸린다는 것을 기억한다.

inb_p(), outb_p(), inw_p(), and outw_p() 매크로는 위에서 본 것들과 동일하게 수행되지만, 이들은 포트를 읽고 쓴 후에 짧은 (1 마이크로초 정도 되는) 지연 시간을 갖는다; 여러분은 <asm/io.h>를 #include하기 전에 REALLY_SLOW_IO 를 #define으로 정의함으로써 4 마이크로초 동안 지연하도록 할 수 있다. 이들 매크로는 일반적으로 (#define SLOW_IO_BY_JUMPING 하지 않는 한, 이것은 그렇게 정확하지 않다) 0x80번 포트에서의 출력을 사용하므로, 여러분은 0x80번 포트를 읽고 쓸 수 있도록 할 필요가 있다. ioperm() 먼저 (0x80 포트의 출력은 시스템의 다른 부분에 영향을 미치지 않아야 한다). 지연할 수 있는 다른 다양한 방법은 뒤에서 볼 수 있다.

최근에 발표된 리눅스 맨페이지 배포본에는 ioperm(), iopl()와 위의 매크로에 대 한 매뉴얼 페이지가 있다.

제 2 절 I/O 포트를 읽고 쓰는 또다른 방법

I/O 포트를 읽고 쓰는 또다른 방법은 open() /dev/port (문자 장치, 주번호 1, 부 번호 4) 읽고 쓰기 위해서 (stdio.h의 f*(), 즉 표준 입출력 파일 처리 함수는 내 부적인 버퍼링을 가지고 있어서, 이들을 피한다??). 그 다음 lseek()는 파일에서 적절한 지점에 위치시키는데 쓰이고(파일 위치 0 = 포트 0, 파일 위치 1 = 포트 1, 등등), 여기서 온 바이트나 워드를 읽고 쓰기 위하여 read() 또는 write()를 사 용한다.

물론, 여러분의 프로그램은 /dev/port를 읽고 쓸 필요가 있다. 이 방법은 보통 쓰 이는 방법보다는 조금 느리지만, 최적화나 ioperm()를 필요로 하지 않는다. (여 러분이 /dev/port에 일반 사용자 권한이나 그룹 권한을 주었을 때는, 루트로 실 행할 필요가 없다.)

제 3 절 인터럽트(IRQs)와 DMA 엑세스

여러분은 사용자-모드 프로그램에서 직접 IRQ나 DMA를 사용할 수 없다. 여러 분은 커널 드라이버를 작성할 필요가 있다면 자세한 사항과 커널 소스 코드 예 제는

`Linux Kernel Hacker's Guide`

(<*www.redhat.com:8080/HyperNews/get/khg.html* `<http://www.redhat.com:8080/HyperNews/get/khg.html>`>)를 읽어보라. 또한 사용자-모드 프로그램에서는 인터럽트를 금지시킬 수 없다.

제 4 절 고해상도 타이밍: 지연 시간

우선, 리눅스의 선점형 멀티태스킹 때문에 사용자 모드의 프로세스가 타이밍을 정확히 제어하는지 보증할 수 없다는 것에 주의한다. 무엇보다도, 사용자 모드의 프로세스는 멀티 태스킹과 리눅스의 선점적인 특성 때문에 정확한 타이밍을 보 장받을 수 없다는 것을 말하고 싶다. 여러분의 프로세스가 10밀리 초에서 (로드가 크게 걸리는 시스템에서) 수 초동안 스케쥴링에서 제외될 수 있다. 그러나, I/O 포트를 사용하는 대부분의 응용 프로그램에서 이는 별로 문제가 되지 않는 다. 이러한 문제를 최소화 하려면 여러분의 프로세스를 nice 명령어로 높은 우선 순위를 부여할 수 있다. (nice(2)의 매뉴얼 페이지를 참조하기 바란다).

일반 사용자 모드의 프로세스를 정확한 타이밍으로 수행시키려 한다면, 사용자 모드의 '리얼 타임' 기능이 지원되어야 한다. 리눅스 2.x 커널에서는 소프트 리 얼 타임 기능이 지원되어야 한다; 자세한 것은 sched_setscheduler(2) 맨페이지 를 참조한다. 하드 리얼 타임을 지원하는 특별한 커널이 있다. 이에 대한 더 많 은 정보는 <URL:*luz.cs.nmt.edu/˜rtlinux/* `<http://luz.cs.nmt.edu/~rtlinux/>`>를 참조한다. 이제, 더 쉬운 타이밍 호출을 시작해 보자. 몇초동안 지연하려면, 최선의 선택은 sleep(3)를 사용하는 것이다. 최소한 수십 초를 지연하려면 (최소 지연 시간이 10 밀리초 정도 될 때), usleep(3)가 그렇게 동작할 것이다. 이러한 기능은 CPU에게 다른 프로세스를 수행하도록 하므로, CPU 타임이 낭비되는 일이 없다. 자세한 것은 매뉴얼 페이지를 참조한다.

50밀리초 이하로 지연할 때는 (프로세서나 머신, 시스템 부하에 좌우되지만), 리 눅스 스케쥴러는 제어권을 돌려 받기 전에 최소한 10-30 밀리초 정도 소모하기 때문에 CPU를 포기할 수는 없다. 이러한 이유 때문에, 아주 작은 지연 시간을 둘 때, usleep(3)은 매개변수에 지정한 것 보다 최소 10밀리초 정도 더 지연을 한다.

짧은 지연 시간을 할당할 경우 (보통 50밀리초 정도 될 것이다), 다양하게 쓰일 수 있는 방법은 udelay()를 사용하는 것인데, /usr/include/asm/delay.h (linux/include/asm-i386/delay.h) 에 정의되어 있다. udelay()는 매개변수를 하나 만 주었을 때 지연하는데 몇 마이크로 초 정도의 시간이 걸리고, 아무것도 리턴 하지 않는다. 매개변수에서 지정한 것보다 몇 마이크로초 정도 더 걸릴 수도 있 는데, 이는 얼마나 기다려야 하는지 계산하는 오버헤드에서 비롯된 것이다 (delay.h에 자세한 사항이 있다).

커널 밖에서 udelay()를 사용하기 위해서, 여러분은 정확한 값으로 정의한 unsigned long 변수인 loops_per_sec를 필요로 할 것이다. 필자가 아는 한, 이 값을 커널에서 얻는 가장 빠른 길은 /proc/cpuinfo의 BogoMips를 읽어 500000을 곱하는 것이다.

리눅스 커널 2.0.x 시리즈에서, 새로운 시스템 호출인 nanosleep(2) (맨페이지 참 조)는 매우 짧은 시간 동안 잠들거나 지연하도록 할 수 있다. 이것은 프로세스가 소프트 리얼 타임 스케쥴링(sched_setscheduler(2)을 사용)을 지정한다면 지연 시간이 2 밀리초 이하일 때 udelay(2)를 사용하고, 다른 경우에는 (usleep()와 같이) sleep을 호출한다. 여러분은 nanosleep()를 사용하기 위해서 loops_per_sec 변수를 필요로 하지는 않을 것인데, 이 시스템 호출이 그 값을 커널에서 가져오 기 때문이다.

수 마이크로초동안 지연하는 또다른 방법은 포트 I/O이다. 포트 0x80에 몇 바이 트를 읽고 쓰려면 프로세서 종류나 속도에 관계없이 정확히 1 마이크로초 정도 기다려야 한다. 몇 마이크로초동안 기다리기 위해서 여러 번 호출할 수 있다. 이 포트 출력은 표준 머신에 대해서는 심각한 부작용이 없다고 확신한다.) (그리고 커널 드라이버도 이를 사용한다.) 이는 {in|out}[bw]_p()가 지연하는 방법이다 (asm/io.h를 참조).

사실, 대부분의 범위 0-0x3ff번의 포트에서 쓰는 포트 I/O 명령은 거의 정확히 1 마이크로초 정도 점유하므로, 예를 들어 여러분이 병렬 포트를 정확히 사용하려 면 포트에 지연 시간을 주기 위해서 추가로 inb()를 호출한다. 여러분이 프로그램이 돌아갈 프로세서의 종류나 클럭 속도를 알고 있다면, 특정 한 어셈블러 명령에서 오는 지연 시간보다 더 짧은 시간을 써넣을(hard-code) 수 있다(그렇지만 기억할 것이 있다면 프로세스는 언제나 수행될 수 있으므로, 그 지연 시간은 실제로 더 길어질 수 있다는 것이다). 아래의 표에서 내부 프로 세서 속도는 클럭 사이클의 수를 결정한다. 예를 들어, 50 MHz의 프로세서 (486DX-50 또는 486DX2-50)의 클럭 사이클은 1/50000000이다.

명령	i386 클럭 사이클	i486 클럭 사이클
nop	3	1
xchg %ax,%ax	3	3
or %ax,%ax	2	1
mov %ax,%ax	2	1
add %ax,0	2	1

(미안하지만, 펜티엄의 경우는 잘 모른다. 아마도 486과 비슷할 것이다.) (필자는 i386에서 한 사이클을 사용하는 명령을 찾을 수 없었다) 표의 nop와 xchg 명령은 부작용이 없다. 나머지 명령은 플래그 레지스터를 변경 할 수 있지만, gcc가 그것을 발견한다고 문제가 되지는 않는다. 이를 사용하려면, 프로그램에서 asm("명령"); 을 호출한다. 위의 테이블에서 문 법에 있는 명령을 준다; 여러 명령을 넣으려면asm("명령 ; 명령 ; 명령"); 이 된 다. asm()은 gcc가 인라인 어셈블리로 변환하여 함수 호출 오버헤드가 없다. 펜티엄에서, 다음과 같은 C 코드로, 최근에 리부트 하였을 때부터 경과한 클럭 사이클의 수를 얻을 수 있다.

extern __inline__ unsigned long long int rdtsc() { unsigned long long int x; __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x)); return x; }

인텔 x86아키텍처에서 한 클럭 사이클보다 더 짧은 지연 시간을 내기는 불가능 하다.

제 5 절 고해상도 타이밍: 시간 측정하기

초단위로 시간을 정확하게 측정하는 방법으로, time(2)를 사용하는 것이 가장 쉬 운 방법이 될 것이다. 더 정확한 시간은, gettimeofday(2)는 마이크로초 단위의 정확도를 지원한다 (스케쥴링에 대해서는 윗부분을 참조한다.) 펜티엄에서 위의 코드는 한 클럭 사이클에서 정확하다. 여러분의 프로세스가 일정 시간이 지난 후에 시그널을 얻기를 원한다면, setitimer(2)를 사용한다. 매뉴얼 페이지에 자세한 사항이 나와 있다.

제 6 절 다른 프로그래밍 언어

이 설명은 C 프로그래밍 언어에 중점을 둔다. 이는 C++과 Objective C에도 직 접 적용되어야 한다. 어셈블러에서, 여러분은 C의 ioperm()과 iopl()를 호출해야 하지만, 직접 읽고 쓸 수 있는 I/O 포트를 사용할 수 있게 된 후에 그렇게 될 것이다.

다른 언어로, 프로그램에 인라인 어셈블러나 C 코드를 삽입하지 않는다면, 가장 쉬운 방법은 여러분이 필요로 하는 I/O 포트 접근에 사용할 함수에 간단한 C 소 스 파일을 작성하고 나머지 프로그램과 컴파일하고 링크하는 것이 될 것이다. 또는 위에서 설명 한대로 /dev/port를 사용한다.

제 7 절 몇몇 쓸만한 포트

여기에는 범용 TTL 로직 I/O에 직접 쓰이는 일반적인 포트에 대한 프로그래밍 정보가 있다.

패러랠 포트 (BASE = 0x3bc for /dev/lp0, 0x378 for /dev/lp1, and 0x278 for /dev/lp2): (여러분이 표준 프린터 처럼 제어하는 방법만 필요하다면, Printing-HOWTO를 보라) 뒤에 설명한 표

준 출력 전용 모드에 덧붙여서, 대부분의 프린터 포트에는 '확장 된' 양방향 모드가 있다. 이 새로운 ECP/EPP 모드(일반적으로 IEEE 1284 표준 이라 한다)에 대한 정보는, <*www.fapo.com/* <http://www.fapo.com/>> 와 <*www.senet.com.au/˜cpeacock/parallel.htm* <http://www.senet.com.au/˜cpeacock/parallel.htm>>를 참조하면 된다. 여러 분이 사용자-모드 프로그램에서 IRQ 또는 DMA를 사용할 수 없기 때문에, 여러 분은 ECP/EPP를 사용하는 커널 드라이버를 만들어야만 할 것이라는 것을 기억 한다; 필자는 누군가가 이러한 드라이버를 만들고 있을 것이라 생각하지만, 자세 한 것은 알지 못한다.

포트 BASE+0 (데이터 포트)은 포트 (D0 to D7은 각각 0에서 7번 비트에 해당 한다. 상태 0 = low (0 V), 1 = high (5 V))의 데이터 신호를 제어한다. 이 포트 에 데이터를 쓰면 포트의 핀에 데이터를 보내게 된다. 포트를 읽게 되면 표준 또는 확장된 쓰기 모드로 마지막에 쓰여진 데이터 또는 확장된 읽기 모드의 또 다른 장치의 핀에서의 데이터를 리턴한다. 포트 BASE+1 (상태 포트)은 읽기 전용으로, 다음과 같은 입력 신호의 상태를 리턴한다.

Bits 0 과 1 은 예약되어 있다. Bit 2 IRQ 상태 (핀이 아니다. 필자도 어떻게 돌아가는지는 알지 못한다.) Bit 3 ERROR (1=high) Bit 4 SLCT (1=high) Bit 5 PE (1=high) Bit 6 ACK (1=high) Bit 7 -BUSY (0=high) (high와 low 상태에 대해서는 확신할 수 없다.) Port BASE+2 (Control port)는 쓰기만 가능하고(읽기를 시도하면 마지막 써넣은 값이 리턴된다), 다음과 같은 상태 시그널을 제어한다: Bit 0 -STROBE (0=high) Bit 1 AUTO_FD_XT (1=high) Bit 2 -INIT (0=high) Bit 3 SLCT_IN (1=high) Bit 4 병렬 포트 IRQ를 사용 가능하도록 한다 (which occurs on the low-to-high transition of ACK) when set to 1. Bit 5 확장 모드 방향을 제어한다 (0 = 쓰기, 1 = 읽기), 쓰기만 가능하다(이 비 트를 읽는 것은 전혀 쓸모가 없다). Bits 6과 7은 예약되어 있다.

(다시 한번 말하지만, high와 low상태의 순서에 대해서는 확신하지 못한다.) 핀 배열 (포트의 25-핀 암컷 D-쉘 커넥터) (i=input, o=output):

1io -STROBE, 2io D0, 3io D1, 4io D2, 5io D3, 6io D4, 7io D5, 8io D6, 9io D7, 10i ACK, 11i -BUSY, 12i PE, 13i SLCT, 14o AUTO_FD_XT, 15i ERROR, 16o -INIT, 17o SLCT_IN, 18-25 Ground

IBM 명세서에서 1, 14, 16, 17(제어 출력)번 핀은 개방된 collector 드라이버를 4.7 킬로오옴 저항을 통해서 5V로 끌어 올린다고 한다. (sink 20 mA, source 0.55 mA, 고수준 출력 5.0 V - pullup). 나머지 핀들은 sink 24 mA, source 15 mA, and 고수준 출력은 최소 2.4 V. 양쪽 모두 low 상태에서는 최대 0.5 V 이 다. IBM 아키텍처가 아닌 것의 병렬 포트는 거의 표준을 따르지 못할 것이다. 더 자세한 정보른 아래를 참조한다.

<*www.hut.fi/˜then/circuits/lptpower.html* <http://www.hut.fi/˜then/circuits/lptpower.html>>. 마지막으로, 경고: 접지 할 때 주의하도록 한다. 필자는 컴퓨터가 켜 있을 때 연 결을 해서 병렬 포트 몇 개를 망가뜨렸다. 이러한 경우에는 보드에 통합되어 있 지 않은 병렬 포트를 사용하는 것이 좋을 것으로 보인다. (여러분은 두번째 저렴 한 표준 '멀티-IO'카드에서 병렬 포트를 구할 수 있다; 필요 없는 포트는 사용 을 금지한다, 그리고 카드의 병렬 포트 주소를 비어 있는 주소로 잡는다. 병렬 포트의 IRQ에 대해서 걱정할 필요는 없다, 사실 거의 쓰이지 않기 때문이다. 게임 (조이스틱) 포트 (0x200-0x207 포트): (일반 조이스틱을 제어하므로, 커널- 수준의 조이스틱 드라이버가 있다. *te.unc.edu/pub/Linux/kernel/patches/* <http://te.unc.edu/pub/Linux/kernel/patches/>ystick-* 을 참조한다.) 핀 배열 (포트의 15핀 암컷 D-쉘 커넥터):

1,8,9,15: +5 V (파워) 4,5,12: 접지 2,7,10,14: 각각 BA1, BA2, BB1, BB2의 디지털 입력 3,6,11,13: 각각 AX, AY, BX, BY, 아날로그 입력

+5 V 핀은 보드의 전원 선에 직접 연결되는 것으로 보인다, 그래서 많은 전력의 출처를 명시(?)할 수 있어야 한다. 보드에 의존하는, 파워 서플라이와 게임 포트 디지털 입력은 포트에 연결하는 두 조이스틱의 버튼에 쓰인다. (조이스틱 A와 조이스틱 B, 각각 버튼 두개) 이들은 보통 TTL-수준의 입력이 되어야 하고, 상태 포트에서 직접 상태를 읽을 수 있다(아래를 참조). 실제 조이스틱은 버튼을 누르면 low(0V) 상태를, 다른 경우에는 high (1킬로 오옴 저항을 통하여 전원 핀에서의 5V)를 리턴한다.

이른바 아날로그 입력은 저항을 측정한다. 게임 포트에는 네 개의 입력에 연결 된 네 개의 원샷 다중 진동기가 있다. 각 입력에는, 입력 핀과 다중 진동기 사이 에 2.2 킬로오옴 저항이 있다. 다중 진동기와 접지. 실제 조이스틱은 각 축마다 전위차계를 지니고 있고 (X와 Y), +5V와 알맞은 입력 핀 사이에 연결되어 있다.(조이스틱 A에는 AX 또는 AY가 있고, 조이스틱 B에는 BX 또는 BY가 있다). 다중 진동기가 활성화 되었을

때는 출력 선을 High로 놓고 각각의 출력 선의 전압이 낮아지기 전에 타이밍 콘덴서가 3.3V가 될 때까지 기다린다. 따라서 다 중 진동기의 높은 전위에서의 간격은 조이스틱의 전위차계에 있는 저항에 비례 한다.(예를 들면, 각 축에서 조이스틱의 위치는) 다음과 같다:

R = (t - 24.2) / 0.011,

R은 전위차계의 저항이고 t는 높은 전위의 유지 시간으로 초를 단위로 한다. 따라서, 아날로그 입력을 읽으려면, 우선 다중 진동기를 활성화 시켜야 하고 (port white와 함께, 아래를 참조) (포트를 계속 읽으면서) 네 축의 상태가 높은 전위에서 낮게 떨어질 때까지 기다려야(polling) 한다. 이러한 polling은 CPU 시 간을 많이 점유하고 리눅스와 같이 실시간 멀티태스킹을 지원하지 않는 시스템 에서는, 포트를 계속 기다릴 수 없으므로 결과가 그리 정확하지 않다. (여러분이 커널 레벨 드라이버를 사용하여 polling 하는 동안 인터럽트를 금지하지 않는 동 안은 그렇지만, 이 방법은 CPU 시간을 더 만이 소모한다.) 신호가 오랜 시간(수십 밀리초)을 점유할 것이라는 것을 안다면, polling하기 전에 CPU 시간을 다른 프로세스에 주기 위하여 usleep()를 호출할 수 있다.

여러분이 읽고 쓰기를 원하는 I/O 포트가 0x201 뿐이다(다른 포트는 완전히 동 일한 방식으로 동작하거나 아무것도 하지 않는다). 이 포트에 쓰기를 하면 (무엇 을 쓰는 지는 중요하지 않다) 다중 진동기가 활성화 된다. 이 포트를 읽으면 입력 시그널의 상태를 리턴한다.

Bit 0: AX (status (1=high) of the multivibrator output) Bit 1: AY (status (1=high) of the multivibrator output) Bit 2: BX (status (1=high) of the multivibrator output) Bit 3: BY (status (1=high) of the multivibrator output) Bit 4: BA1 (digital input, 1=high) Bit 5: BA2 (digital input, 1=high) Bit 6: BB1 (digital input, 1=high) Bit 7: BB2 (digital input, 1=high)

직렬 포트: 여러분이 지원하려는 장치가 RS-232와 닮았다면, 여러분은 직렬 포 트를 사용할 수 있어야 한다. 리눅스 직렬 드라이버는 대부분의 모든 응용 프로 그램이 필요한 것으로는 충분하다. (여러분은 직렬 포트를 직접 조작하지는 말아 야 한다; 어쨌든, 여러분은 이러한 일을 하는 커널 드라이버를 작성해야 할 것이다.); 이 포트는 매우 다양한 용도로 쓰이는데, 비표준 bps 비율을 사용하여도 문제를 일으키지 않는다.

유닉스 시스템의 직렬 포트 프로그래밍에 대한 더 자세한 정보는 termios(3) 메 뉴얼 페이지, 시리얼 드라이버 소스 코드(/usr/src/linux/drivers/char/serial.c), <*www.easysw.com/~mike/serial/index.html* `<http://www.easysw.com/~mike/serial/index.html>`>를 참조한다. 여러분이 좋은 아날로그 I/O를 원한다면, 여러분은 ADC와 DAC칩을 병렬 포트 에 연결할 수 있다. (힌트: 파워를 위해서는 게임 포트 커넥터 또는 컴퓨터 케이 스의 바깥에 연결하는 예비 디스크 드라이브의 파워 커넥터를 사용한다, 여러분 이 낮은 전력 장치를 사용하거나 파워 때문에 병렬 포트 자체를 사용할 수 있는 것이 아니라면) 또는, AD/DA 카드를 구입한다(더 느린 카드들 대부분은 I/O 포 트가 제어한다). 부정확하고 영점 불안정함에도 한 개나 두 개의 채널에 만족한 다면, 리눅스 사운드 드라이버에서 지원되는 저가의 사운드 카드도 할 수 있다. (이는 충분히 빠르다.)

다른 힌트: 여러분이 리눅스용 프린트된 회로 보드 디자인 소프트웨어를 찾고 있다면, Pcb라 불리는 공개의 X11 응용 프로그램이 있는데, 잘 동작할 것이다. 최악의 경우 여러분이 복잡해서 아무것도 할 수 없을 수도 있다. 이 프로그램은 많은 리눅스 배포본에 포함되어 있고, *ite.unc.edu/pub/Linux/apps/circuits/* `<http://ite.unc.edu/pub/Linux/apps/circuits/>`pcb-*에서 구할 수 있다.

제 8 절 문제 해결

Q1. 포트를 제어하려 할 때 segmentation faults 에러가 난다.

A1. 여러분의 프로그램이 루트 권한을 가지고 있지 않거나 ioperm() 호출이 다 른 이유로 실패하였을 것이다. ioperm()의 리턴값을 점검해보라. 또한, 실제로 ioperm()으로 읽고 쓰기가 가능한 포트에 접근하는 것인지 점검한다.

Q2. 어디에서도 in*(), out*()로 정의된 함수를 찾을 수 없다. gcc는 undefined references라고 한다.

A2. 여러분은 최적화 옵션(-O)을 켜지 않고 컴파일 하였다, 따라서 gcc는 asm/io.h 매크로를 분석할 수 없다. 또는 #include <asm/io.h>를 하지 않았다.

Q3. out*()은 아무것도 하지 않거나, 이상하게 동작한다.

A3. 매개변수의 순서를 점검한다; 도스에서는 outportb(port,value)가 아닌 outb(value,port)가 되어야 한다.

Q4. 표준 RS-232 포트/병렬 프린터/조이스틱을 제어하고 싶다.

A4. 여러분은 이미 존재하는 드라이버를 (리눅스 커널 또는 X 서버 어딘가에 있 는) 제거하는 것이 좋다. 그 드라이버는 매우 다양한 목적으로 쓰일 수 있어서, 비표준 장치도 동작시키기도 한다. 이 문서가 제시하고 있는 표준 포트에 대한 정보를 참조한다.

제 9 절 소프트웨어 예제

여기에는 I/O 포트를 엑세스할수 있는 간단한 예제 코드가 있다.

```
/*
 * example.c: 간단한 포트 입출력 예제
 *
 * 이 코드는 특별히 쓸만한 건 없고 , 포트에 쓰고, 잠시 멈춘 다음,
 * 포트를 읽는다. ‘gcc -O2 -o example example.c’로 컴파일한다.
 */
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>
#define BASEPORT 0x378 /* lp1 */
int main()
{
  /* Get access to the ports */
  if (ioperm(BASEPORT,3,1)) {perror("ioperm");exit(1);}

  /* Set the data signals (D0-7) of the port to all low (0) */
  outb(0,BASEPORT);

  /* Sleep for a while (100 ms) */
  usleep(100000);

  /* Read from the status port (BASE+1) and display the result */
  printf("status: %d\n",inb(BASEPORT+1));
  /* We don’t need the ports anymore */
  if (ioperm(BASEPORT,3,0)) {perror("ioperm");exit(1);}
  exit(0);
}
/* end of example.c */
```

제 10 절 감사드릴 분들

일일이 나열하기에는 너무나 많은 분들이 도움이 되었다, 하지만 매우 감사한다. 조언 받은 모든 분들께 답장을 하지는 못하였다. 이에 대해 죄송스럽게 생각하 고, 다시 한번 도움을 주심에 감사드린다. 리눅스 I/O 포트 프로그래밍 미니 하우투 끝