논문 2011-48SC-5-10

ARM9기반의 Nested Software Interrupt의 구현 및 Parameter의 전달 방식

(Implementation of Nested Software Interrupt and Passing Way of Parameters based on ARM9)

한 길 종*, 류 경 식*, 이 정 원**, 김 용 득***

(Gil Jong Han, Kyeung Seek Lew, Jung Won Lee, and Yong Deak Kim)

요 약

SWI 함수의 다중 호출과 매개변수의 효율적인 전달방법으로 일반적인 소프트웨어 인터럽트의 사용방법의 문제점을 해결하고자 한다. 소프트웨어 인터럽트는 강제로 소프트웨어적인 인터럽트를 발생하여 시스템 함수를 호출하거나 시스템 리소스를 사용하기 위해 이용되기 때문에 무분별한 접근들로부터 보호되어야 하지만, SWI명령어의 제한된 사용방식으로 효율적인 처리가 어렵다. 본 논문에서는 이를 해결하기위한 SWI의 Nested Call과 SWI의 서비스 루틴에 매개변수를 효율적으로 전달하는 방식을 설계하고 구현하였다. 즉, 단일 호출방식에서 다중 호출방식으로, 좀 더 유연하게 호출할 수 있도록 하였고, 두 가지 매개변수 이용 방식의 장단점을 비교분석하였다. 두 매개변수 이용 방식의 가장 큰 차이점은 속도와 가독성이다. 비록 스택 포인터 얻는 방식이 가독성이 매우 뛰어난 장점을 보였지만 많은 오버헤드를 발생시켰다. 이에 반해, 스택 포인터 전달 방식은 오버헤드의 발생을 줄여 속도적인 측면에서 약 19 %의 효율성을 보였다.

Abstract

I try to solve the problem of the usage of the general software interrupt with the nested call of the software interrupt and the effective passing way of the parameters. The software interrupt should be protected against the indiscriminate access because it is used to call the system functions or to use the system resources by generating a software interrupt. But, it is difficult to effectively handle the SWI instruction because of its limited usage. I designed and implemented nested call of the software interrupt and the effective way that handle the parameters in the software interrupt service routine to solve this problem in this paper. In other words, from the single SWI call to the nested SWI call, I improved the software interrupt use all the more flexibly, and I compared and analyzed the strong and weak points of the two passing ways of the parameters. The main differences between these two ways are speed and readability. The stack pointer getting way incurred a lot of overhead although it has a very great readability. But, the stack pointer passing way producted 19% of the effectivity in speed by reduce overhead

Keywords : Software Interrupt(SWI), Nested call, APCS, Mode & Exception, Embedded

Ⅰ. 서 론

일반적으로 소프트웨어 인터럽트(Software Interrupt: SWI)sms 강제로 소프트웨어적인 인터럽트를 발생하여 프로그램의 모드(Mode)를 비특권(Non-privileged) 모드에서 특권(Privileged)모드로 바꾸어, 시스템 함수(System Function)를 호출하거나 시스템 리소스(System Resource) 그리고 공유된 데이터를 사용하기 위해 이용된다. 시스템 함수, 시스템 리소스 그리고 공유된 데이터 등을 사용하기 위해서는 이들에 대한 접근권한을 가져야 하는데 멀티-태스킹(Multi-Tasking)의 경우, 서로 다른 태스크들이 이러한 접근권한을 독점하려고 시도한다. 이 때문에, 무분별하게 접근권한을 독점

* 학생회원, ** 정회원, *** 평생회원,
아주대학교 전자공학부
(Dept. of Electronics Engineering Ajou University)
접수일자: 2010년10월29일, 수정완료일: 2011년7월12일

하려는 시도들로부터 보호되어야 하고, SWI의 사용 역시 더욱 신중하고 체계적으로 구현 및 처리되어야 한다.

SWI는 또한 독립적으로 컴파일된(Compiled) 프로그램의 함수를 호출하기 위해 사용된다. 일반적으로 중/대형 프로그램에서 지속적으로 업데이트되는 모듈프로그램이 이에 해당하는데, 업데이트될 때마다 다시 빌드(Re-build)하는 것은 비효율적인 일이다. 따라서 메인 프로그램은 SWI를 이용하여 독립적으로 컴파일 된 모듈 프로그램의 함수를 마치 API처럼 호출하고, 실제의 수행은 모듈프로그램 내에서 자체적으로 처리하도록 한다. 이 또한 중요한 개념이며 유용하게 사용된다.[1]

본 논문에서는 이러한 SWI의 중요성에도 불구하고, 단일(Single) SWI호출과 매개변수(Parameters) 전달 방식에 존재하는 유연하지 못하고 제한적인 사용방식을 개선하는 방식을 제안 및 구현하였다. 이에 Nested Call과 SWI의 서비스 루틴(ISR)에 매개변수들을 효율적으로 전달하는 방식을 제안한다.

이후의 구성은 다음과 같다. Ⅱ장에서 본론으로 들어가 SWI의 기본적인 특징과 SWI의 Nested Call, 그리고 매개변수를 효율적으로 전달하는 방식을 제안 및 구현한다. 이어서 Ⅲ장에서는 제안 및 구현된 Nested Call과 매개변수 이용 방식이 적용된 SWI의 실험결과를 보인다. 마지막으로 Ⅳ장에서는 실험을 통한 결과를 통해 결론을 내린다.

Ⅱ. 본 론

1. 기본적인 SWI의 호출 및 처리 과정

ARM SWI 명령어의 32bit binary 형태는 그림 1과 같다.

SWI 명령어를 구성하는 32bit 중, 상위 4bit([31:28])는 프로세스의 현재 상태를 나타내는 상태 필드이고, 그 다음 4bit([27:24])은 SWI 명령어임을 나타내는 비트 패턴이다. 마지막 하위 24bit([23:0])는 일반적으로 소프트웨어 인터럽트의 특정 서비스 루틴을 실행하기 위해 호출할 서비스 루틴의 번호의 정보를 갖는 코멘트 필드이다. 다음은 0번에 해당하는 소프트웨어 인터럽트 서비스 루틴을 호출하는 예이다.

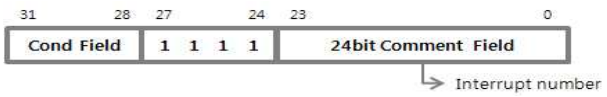


그림 1. ARM SWI 명령어
Fig. 1. ARM SWI Instruction.

```
swi         0
```

SWI 명령어를 실행할 시점의 사이클에서의 PC는 다음, 다음 명령어의 주소이다. 즉, 소프트웨어 인터럽트 서비스 루틴을 수행한 후, 복귀할 주소는 PC-8이다. SWI 명령어에 의해 감독자 모드(SVC Mode)로 진입 시, APCS(ARM Procedure Call Standard)[7]에 의해 복귀주소 값인 PC-4가 현재 프로그램 모드(Program Mode)의 LR에 저장된다. 소프트웨어 인터럽트 핸들러에서는 어떠한 소프트웨어 인터럽트 서비스 루틴을 수행할 것인지, SWI 명령어에서 해당 서비스 루틴의 번호를 추출해야 한다. SWI 명령어는 복귀할 주소의 이전 명령어이기 때문에, LR-4로 구할 수 있다.

```
LDR         r0, [lr, #-4]
```

소프트웨어 인터럽트 서비스 루틴의 번호는 SWI 명령어의 하위 24bit에 있다. 따라서 상위 8bit를 마스킹(Masking)함으로써 구할 수 있다.

```
BIC         r0, r0, #0xFF000000
```

본 연구에서 제안하고 구현한 코드 시퀀스의 구성은 그림 2와 같이 크게 4부분으로 나눌 수 있다. SWI 명령어를 API처럼 호출하는 Main 부분, 실제 SWI 명령어를 호출하고 APCS과 같은 역할을 하는 API 부분, SWI 핸들러로 분기하지 전, ISR의 번호를 구하고, 해당 ISR으로 분기하는 SWI 핸들러 부분 마지막으로 실제 ISR이 설계된 부분으로 구성되어 있다.

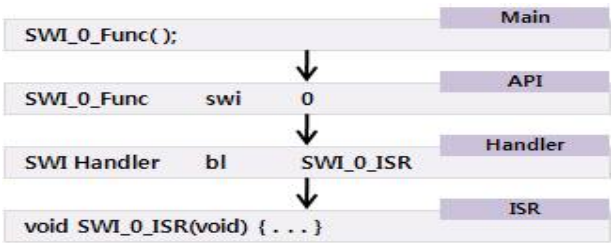


그림 2. 코드 시퀀스
Fig. 2. Code sequence.

2. SWI의 Nested Call

SWI의 ISR은 일반적으로 단일로 처리된다. 하지만, 수행되는 동작의 유연성을 위해 경우에 따라 SWI의

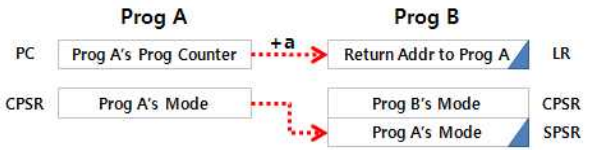


그림 3. 프로그램 A에서 프로그램 B로의 환경 전환
Fig. 3. Context Switching from Program A to Program B.

ISR에서 또 다른 SWI 명령어를 호출하는 경우가 있을 수 있다. 즉, 현재 프로그램이 SWI의 ISR이 동작하는 감독자(Supervisor : SVC) 모드인 상태에서 또 다시 SWI 명령어를 호출하여 SWI의 ISR이 중첩되어 처리하는데, 이것을 SWI의 Nested Call이라 한다.

그림 3과 같이, 프로그램 A에서 프로그램 B로 모드가 전환될 경우, 현재의 프로그램 상태를 나타내는 레지스터 CPSR(Current Process Status Register)에는 프로그램 B의 상태가 저장되어 있고, 프로그램 B가 호출되기 직전의 프로그램 상태를 저장하고 있는 레지스터 SPSR(Saved Process Status Register)에는 프로그램 A의 상태가 저장되어 있을 것이다. 또 프로그램 B의 ISR의 수행 후, 프로그램 A로 복귀하기 위한 복귀주소는 프로그램 B의 LR(Link Register)에 저장될 것이다.

SWI의 Nested Call의 경우, 현재 SVC 모드의 상태에서 또 다시 SWI 명령어를 호출하여 SVC 모드로 진입하기 시작한다. 이때, 중첩으로 호출된 SVC 모드의 LR과 SPSR 등의 레지스터 값이 중첩되기 직전의 SVC 모드의 해당 레지스터 값들을 덮어써 손상시킬 수 있다. 때문에, 중첩으로 호출하기 전에 LR_svc와 SPSR_svc 값을 잃지 않도록 저장해야 하는데, 일반적으로 스택을 이용한다.

```
stmfd     sp!, {lr}
mrs       r0, SPSR
stmfd     sp!, {r0}
```

SWI의 ISR을 수행한 후, Nested Call 이전의 환경으로 복구하기 위해 저장 해 두었던 LR과 SPSR을 반드시 복구해야 한다.

```
ldmfd     sp!, {r0}
msr       SPSR, r0
ldmfd     sp!, {rc}^
```

3. Parameter 전달 방식

그림 4는 APCS[7]에 따른 매개변수의 일반적인 전달

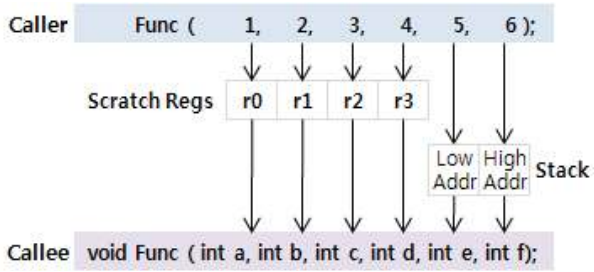


그림 4. 매개변수의 일반적인 전달과정
Fig. 4. General procedure passing parameters.

과정(단, 각각의 매개변수가 4Byte이며 Integer 타입으로 총 6개가 전달)이다. 매개변수의 처음 16Byte는 r0-r3에 저장되고, 그 이후의 매개변수들은 현재 프로그램 모드의 스택에 저장되어 전달된다. 정확하게는 Caller의 스택에 저장된다. 하지만 호출한 측(Caller)과 호출된 측(Callee) 간에 함수의 프로토타입(Prototype)이 같기 때문에, 매개변수의 처음 16Byte가 r0-r3을 통해 전달되고, 그 이후의 매개변수는 스택을 통해 전달되지만, Callee는 이에 대해 고려할 필요 없이 프로토타입에 따라 매개변수를 이용할 수 있다.

위의 전달과정과 같이 Caller와 Callee의 프로그램 모드가 같을 경우, 같은 스택과 스택 포인터를 사용하기 때문에 스택에 저장된 매개변수들을 이용하기 쉽다. 하지만 반대로, Caller와 Callee의 프로그램 모드가 다를 경우, 스택에 저장된 매개변수를 참조하기위해서 Callee는 Caller의 스택에 접근할 수밖에 없다. Callee에서 Caller의 스택에 접근하기 위해 스크래치 레지스터(Scratch Register)인 r0-r3을 이용하거나 Caller의 스택 포인터(Stack Pointer : SP)를 이용하는 등의 다양한 방식을 이용해야 한다.

이에 본 연구에서는 Caller와 Callee의 프로그램 모드가 다른 경우에서의 두 가지의 매개변수 이용 방식을 제안한다. 먼저, Callee인 SWI의 ISR에서 Caller의 스택 포인터를 구하여 Caller에 저장된 매개변수에 접근하는 스택 포인터 얻는 방식(Stack Pointer Getting Way)에 대해서 소개하고, 이어서 Caller의 스택 포인터가 Callee인 SWI의 ISR의 매개변수로 전달되어 직접 접근하는 스택 포인터 전달 방식(Stack Pointer Passing Way)을 소개한다.

가. 스택 포인터 얻는 방식

스택 포인터 얻는 방식은 그림 5와 같이, Callee에서 Caller의 프로그램 상태를 알 수 있는 SPSR을 참조하

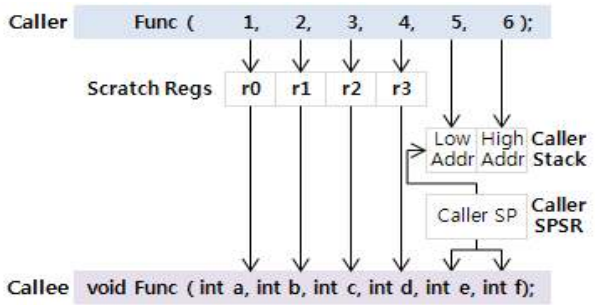


그림 5. 스택 포인터 얻는 방식의 과정
Fig. 5. Procedure of Stack Pointer Getting Procedure.

여 Caller 프로그램의 모드와 해당 스택 포인터를 구한 뒤, 그 스택 포인터로 나머지 매개변수들을 참조하는 방식이다. 가령, 사용자 모드에서 감독자 모드로 전환이 일어날 경우, 감독자 모드의 SPSR을 통해 사용자 모드의 스택 포인터를 구할 수 있다.

스택 포인터 얻는 방식의 두 가지 장점은 첫째, 매개변수로의 용이한 접근성과 스택의 오버플로에 대한 위험 감소이다. 먼저, 매개변수로의 용이한 접근성은 SWI의 ISR 함수와 이를 호출하는 API 함수의 프로토타입을 동일하게 설계함으로써, 가독성이 뛰어나 매개변수에 쉽게 접근한 수 있다는 점이다. 다음으로 스택의 오버플로에 대한 위험 감소는 가령, Callee인 감독자 모드의 프로그램이 크기가 상당히 제약적인 감독자 모드의 스택이 아닌, Caller인 사용자 모드의 스택을 이용하여 상대적으로 크기가 크게 설계된 사용자 모드의 스택을 사용함으로써, 그만큼 스택의 오버플로에 대한 위험을 줄일 수 있다. SWI의 ISR 함수와 이를 호출하는 API 함수의 프로토타입을 동일하게 설계하기 위해서는 약간의 오버헤드가 발생한다. Caller에서 Callee로 매개변수를 전달하는 과정에서 총 16Byte 크기 이상을 전달할 경우 초과된 매개변수는 스택에 저장되어 전달되는데, 이는 Caller의 스택에 저장되어 전달된다. 이에 Callee는 Caller의 스택에 접근하기 위해한 스택 포인터를 Caller의 상태를 저장하고 있는 SPSR을 이용하여 Caller의 모드와 스택 포인터를 구할 수 있다. 아래는 Callee의 SPSR을 이용하여 Caller의 스택 포인터를 구하는 코드의 일부분이다.

```
; save callee's sp
      mov       r6, sp
; change to caller's mode
      mrs       r7, spsr
      msr       cpsr_cf, r7
; get caller's sp
      add       r5, sp
; return to callee's mode
      msr       cpsr_cf, #SVC_Mode
; set caller's sp
      mov       sp, r5
      ...                 ; SWI ISR
; restore callee's sp
      mov       sp, r6
```

그리고 이 방식은 매개변수 참조뿐만 아니라 Callee의 프로그램을 Caller의 스택에서 동작시키기 때문에, Caller의 스택에 저장되어있는 매개변수 이외의 다른 데이터를 손상시키지 않아야 한다. 현재 설계된 스택의 구조를 보면 스택에 저장된 매개변수 위에 SWI 함수를 호출하는 API 함수가 실행이 끝난 뒤의 복귀 주소가 저장되어 있다. 따라서 이 Callee는 Caller의 스택을 사용하기 전에 이 API 함수의 복귀주소를 보호하는 과정이 필요하다. 아래의 코드는 스택 포인터 얻는 방식이 SWI 함수를 호출하는 API 함수의 프로토타입과 동일하게 설계되어 가독성이 뛰어나 매개변수 참조가 용의한 경우를 보여주는 한 예이다.

```
void SWI_Func(char, double, int, char *);
```

```
void SWI_Func_ISR(char c, double f,
                  int d, char * str) {

      Uart_Printf("%c,%f,%d,%s\n",
                        c,f,d,str);
}
```

나. 스택 포인터 전달 방식

스택 포인터 얻는 방식이 SPSR을 이용해 Caller의 스택 포인터를 얻는 방식이었다면, 스택 포인터 전달 방식은 그림 6과 같이 전달하고자 하는 모든 매개변수 (즉, APCS에 따라 일반적으로 r0-r3에 저장되는 전달

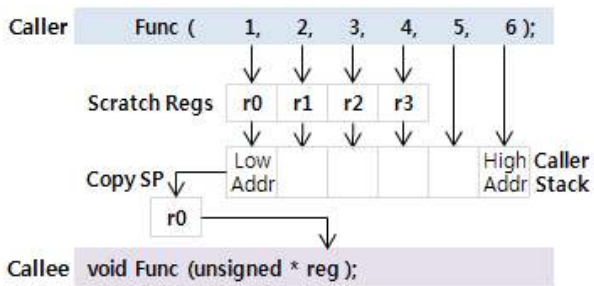


그림 6. 스택 포인터 전달 방식의 과정
Fig. 6. Procedure of Stack Pointer Passing Way.

되는 매개변수)를 Caller의 스택에 저장한 뒤, 다시 이 스택 포인터를 매개변수로 전달하여 모든 저장된 매개변수를 참조하는 방식이다.

스택 포인터 전달 방식은 Caller로부터 전달된 모든 매개변수가 연달아 저장된 스택의 포인터를 매개변수로 전달함으로써 C로 작성된 핸들러와 함수에서 모든 매개변수를 유연하게 참조할 수 있다. 이 경우, 스택 포인터 얻는 방식과 다르게, Callee인 SWI의 ISR가 매개변수 참조만 Caller 스택에서 하고 프로그램은 Callee 스택을 사용하는 정상적인 방식으로 수행 가능하다.

스택 포인터로 모든 매개변수를 참조하기 위해서는 스택에 저장하는 과정이 추가적으로 필요하다. Caller는 스크래치 레지스터 r0를 통해 스택 포인터를 전달하기 때문에, r0-r3을 통해 전달되는 매개변수들을 손상되지 않도록 미리 스택에 저장해야 한다. 아래의 코드는 스택 포인터를 전달하기위한 모든 매개변수를 스택에 저장하고 이 스택에 대한 새로운 포인터를 구하는 과정의 일부분이다.

```
; save the parameters
      STMFD      sp!, {r0-r3}
; get the new sp
      MOV        r1, sp
      SWI        0
; return to the original position
      ADD        sp, sp, #16       ; 16Bytes
```

SWI 함수를 호출하는 API 함수의 모든 수행이 끝난 뒤, 이 API 함수를 호출한 프로그램으로 복귀할 주소를 미리 스택에 저장해야 한다. 이 작업이 없을 경우, 어셈블리로 작성된 API 함수는 복귀 명령(프로그램 카운터(PC)가 바뀌는 동작)이 일어날 때까지 계속해서 다음 명령을 수행하기 때문에, 오작동이 일어날 수 있다. 아래의 코드는 스택 포인터 전달 방식이 매개변수로 전달된 스택 포인터를 이용하여 Caller의 스택에 저장된 매개변수들을 참조하는 간단한 예이다.

```
void SWI_Func(char, double, int, char *);

void SWI_Func_ISR(void *reg) {
      unsigned * pInt = ((unsigned *)reg);
      char       c;
      double     f;
      int        d;
      char *     str;

      c = *(char*)pInt);          pInt++;
      f = *(double*)pInt);        pInt+=2;
      d = *(int*)pInt);           pInt++;
      str = *(char**)pInt);       pInt++;

      Uart_Printf("%c,%f,%d,%s\n",
                        c,f,d,str);
}
```

Ⅲ. 실 험

1. SWI의 Nested Call 실험

그림 7은 SWI의 Nested Call을 실험하기 위한 간단한 예이다. 먼저, 문자열을 출력하기 위한 SWI 함수의 API를 호출하게 되면, 해당 ISR에서 전달받은 매개변수인 문자열의 각 문자들을 출력하는 또 다른 SWI 함수의 API를 내부에서 호출한다. 각각의 문자를 출력하기 위한 또 다른 SWI 함수의 ISR이 문자열 길이만큼 호출 되고 모든 Nested Call된 SWI 함수는 종료된다. 이 실험에 사용되는 매개변수 사용 방식은 스택 포인터 전달 방식이다.

그림 8과 같이, SWI의 Nested Call의 결과를 확인할 수 있다. 비교적 간단한 실험이지만, 정확한 동작 결과를 살펴보기에 매우 적합하다.

그림 7. 중 소프트웨어 인터럽트
Fig. 7. Simple sequence of SWI Nested Call.

```
Now Program Run!

Nested Software Interrupt
Hello, World!

Returned from SWI
```

그림 8. 다중 소프트웨어 인터럽트 실행 결과
Fig. 8. The Result Of Nested Software Interrupt.

2. SWI의 Nested Call 실험

매개변수 이용 방식 실험은 그림 2와 같이 크게 4부

분으로 나누어진 코드 시퀀스로 설계된 두 매개변수 이용 방식에 몇 가지 예제를 각각 실험하여 각 코드 부분의 수행시간을 측정하고 비교, 분석한다. 수행시간은 총 명령어에 대한 사이클 수와 타이머를 이용한 시간을 각각 측정하는데, T32라는 인-서킷 에뮬레이터로 측정한다. 그리고 APCS를 준수한 스택정리와 SWI 명령어 실행 그리고 인터럽트 서비스 루틴을 호출하기 위한 핸들러부분을 통틀어 탑-레벨 핸들러라고 칭한다.

가. 탑-레벨 핸들러의 수행시간 비교

SWI를 발생하여 해당 ISR이 동작하기 까지의 과정에서 SWI 함수를 호출하는 API 함수 부분과 SWI의 ISR을 호출하는 핸들러로 이루어진 가장 중요한부분인 탑-레벨 핸들의 수행시간을 두 매개변수 이용 방식에 대하여 수행시간을 각각 비교한다. 이 실험에 사용된 예제는 SWI를 이용하여 문자열을 출력하는 함수이다. SWI 함수를 호출하는 과정에서 탑-레벨 핸들러는 각각의 매개변수 이용 방식에서 모두 동일하게 설계되어 있기 때문에 예제의 성격에 영향을 받지 않는다.

(1) 스택 포인터 얻는 방식

그림 9와 같이, 스택 포인터 얻는 방식에서는 APCS & SWI의 부분에서는 비교적 간단하나 SPSR을 이용하여 Caller의 스택 포인터를 구하는 오버헤드가 발생하여 인터럽트 핸들러 과정이 스택 포인터 전달 방식에 비해 비교적 복잡하다.

```
        20          stmfd   sp!, {lr}
SR:30002C2C E92D4000  SWI_Putc: stmdb   r13!,{r14}
        21          swi     1
SR:30002C30 EF000001          swi     0x1
        22          ldmfd   sp!, {pc}
SR:30002C34 E8BD8000          ldmia   r13!,{pc}
```

(a) APCS & SWI Call

```
SR:00000050 E92D47F0          stmdb   r13!,{r4-r10,r14}
SR:00000054 E14F7000          mrs     r7,spsr
SR:00000058 E92D0080          stmdb   r13!,{r7}
SR:0000005C E1A0600D          mov     r6,r13
SR:00000060 E129F007          msr     cpsr,r7
SR:00000064 E28D5004          add     r5,r13,#0x4
SR:00000068 E5154004          ldr     r4,[r5,#-0x4]
SR:0000006C E329F013          msr     cpsr,#0x13
SR:00000070 E1A0D005          mov     r13,r5
SR:00000074 E51EA004          ldr     r10,[r14,#-0x4]
SR:00000078 E3CAA4FF          bic     r10,r10,#-0x1000000
SR:0000007C E59F91A4          ldr     r9,0x228
SR:00000080 E1A0E00F  I_Bit:  mov     r14,pc
SR:00000084 E799F10A          ldr     pc,[r9,+r10,lsl #0x2]
SR:00000088 E5054004          str     r4,[r5,#-0x4]
SR:0000008C E1A0D006          mov     r13,r6
SR:00000090 E8BD0080          ldmia   r13!,{r7}
SR:00000094 E169F007          msr     spsr,r7
SR:00000098 E8FD87F0          ldmia   r13!,{r4-r10,pc}^
```

(b) Interrupt Handler

그림 9. 스택 포인터 얻는 방식의 탑-레벨 핸들러 수행시간

Fig. 9. Execution time of Top-Level handler on Stack Pointer Getting Way.

APCS & SWI Call 과정에서 3개의 명령어 즉 9개의 사이클이 소요되고, 인터럽트 핸들러에서 19개의 명령어 즉 64개의 사이클이 소요되어, 총 73개의 명령어 사이클이 소요되었다.

(2) 스택 포인터 전달 방식

그림 10과 같이, 스택 포인터 전달 방식에서는 스택 포인터를 전달하기 위해 기존의 r0-r3에 해당하는 매개변수를 미리 스택에 저장하고 새로운 스택 포인터를 전달하는 오버헤드가 발생하여 APCS & SWI과정이 스택 포인터 얻는 방식에 비해 비교적 복잡하다.

```
        68          STMFD   sp!, {r0-r3}
SR:30002C90 E92D000F  SWI_Puts: stmdb   r13!,{r0-r3}
        69          MOV     r1, sp
SR:30002C94 E1A0100D          mov     r1,r13
        70          STMFD   sp!, {lr}
SR:30002C98 E92D4000          stmdb   r13!,{r14}
        71          SWI     4
SR:30002C9C EF000004          swi     0x4
        72          LDMFD   sp!, {r2}
SR:30002CA0 E8BD0004          ldmia   r13!,{r2}
        73          ADD     sp, sp, #16
SR:30002CA4 E28DD010          add     r13,r13,#0x10
        74          MOV     pc, r2
SR:30002CA8 E1A0F002          mov     pc,r2
```

(a) APCS & SWI Call

```
SR:00000050 E92D4000          stmdb   r13!,{r14}
SR:00000054 E14F2000          mrs     r2,spsr
SR:00000058 E92D0004          stmdb   r13!,{r2}
SR:0000005C E51E0004          ldr     r0,[r14,#-0x4]
SR:00000060 E3C004FF          bic     r0,r0,#-0x1000000
SR:00000064 EB000261          bl      0x9F0
SR:00000068 E8BD0004          ldmia   r13!,{r2}
SR:0000006C E169F002          msr     spsr,r2
SR:00000070 E8FD8000          ldmia   r13!,{pc}^
```

(b) Interrupt Handler

그림 10. 스택 포인터 전달 방식의 탑-레벨 핸들러 수행시간

Fig. 10. Execution time of Top-Level handler on Stack Pointer Passing Way.

APCS & SWI Call 과정에서 7개의 명령어 즉 22개의 사이클이 소요되고, 인터럽트 핸들러에서 9개의 명령어 즉 22개의 사이클이 소요되어, 총 44개의 명령어 사이클이 소요되었다.

스택 포인터 전달 방식이 스택 포인터 얻는 방식보다 APCs & SWI Call 과정에서 속도적인 성능면에서 59% 뒤지지만, 인터럽트 핸들러 과정에서는 오히려 66% 더 좋다. 두 과정의 총 수행시간을 고려해보았을 경우, 스택 포인터 전달 방식이 스택 포인터 얻는 방식보다 속도적인 성능에서 40% 앞선다.

나. 동일한 데이터 형의 매개변수의 수행시간 비교

이 실험은 탑-레벨 핸들러를 제외한 ISR의 수행 시간을 측정하는데, 전달되는 매개변수의 데이터 타입이 동일하여 복잡한 캐스팅이 필요 없는 경우이다.

실험 결과, 스택 포인터 얻는 방식이 전달 방식은 그

```
                45 void SWI_6Int_ISR(int d1, int d2, int d3, int d4, int d5, int d6)
SR:30000A24 E92D41F0  SWI_6Int_:stmdb   r13!,{r4-r8,r14}
SR:30000A28 E1A08001            mov     r8,r1           ; r8,d2
                46         Uart_Printf("%d\n", d1);
SR:30000A2C E1A01000            mov     r1,r0           ; d2,d1
SR:30000A30 E24F0024            sub     r0,pc,#0x24     ; d1,pc,#36
SR:30000A34 E59D4018            ldr     r4,[r13,#0x18]  ; d5,[r13,#24]
SR:30000A38 E59D501C            ldr     r5,[r13,#0x1C]  ; d6,[r13,#28]
SR:30000A3C E1A07003            mov     r7,r3           ; d4,r3
SR:30000A40 E1A06002            mov     r6,r2           ; d3,r2
SR:30000A44 EBFFFECA            bl      0x30000574      ; Uart_Printf
                47         Uart_Printf("%d\n", d2);
SR:30000A48 E1A01008            mov     r1,r8           ; r1,d2
SR:30000A4C E24F0040            sub     r0,pc,#0x40
SR:30000A50 EBFFFEC7            bl      0x30000574      ; Uart_Printf
                48         Uart_Printf("%d\n", d3);
SR:30000A54 E1A01006            mov     r1,r6           ; r1,d3
SR:30000A58 E24F004C            sub     r0,pc,#0x4C
SR:30000A5C EBFFFEC4            bl      0x30000574      ; Uart_Printf
                49         Uart_Printf("%d\n", d4);
SR:30000A60 E1A01007            mov     r1,r7           ; r1,d4
SR:30000A64 E24F0058            sub     r0,pc,#0x58
SR:30000A68 EBFFFEC1            bl      0x30000574      ; Uart_Printf
                50         Uart_Printf("%d\n", d5);
SR:30000A6C E1A01004            mov     r1,r4           ; r1,d5
SR:30000A70 E24F0064            sub     r0,pc,#0x64
SR:30000A74 EBFFFEBE            bl      0x30000574      ; Uart_Printf
                51         Uart_Printf("%d\n", d6);
SR:30000A78 E1A01005            mov     r1,r5           ; r1,d6
SR:30000A7C E8BD41F0            ldmia   r13!,{r4-r8,r14}
SR:30000A80 E24F0074            sub     r0,pc,#0x74
SR:30000A84 EAFFFEBA            b       0x30000574      ; Uart_Printf
```

(a) 스택 포인터 얻는 방식의 ISR

```
                47 void SWI_6Int_ISR(unsigned *reg) {
SR:30000A04 E92D41F0  SWI_6Int_:stmdb   r13!,{r4-r8,r14}
                48         int d1=reg[0], d2=reg[1], d3=reg[2], d4=reg[3], d5=reg[4],
SR:30000A08 E89001F2            ldmia   r0,{r1,r4-r8}   ; reg,{r1,r4-r8}
                56         Uart_Printf("%d\n", d1);
SR:30000A0C E24F0020            sub     r0,pc,#0x20     ; reg,pc,#32
SR:30000A10 EBFFFECF            bl      0x30000554      ; Uart_Printf
                57         Uart_Printf("%d\n", d2);
SR:30000A14 E1A01004            mov     r1,r4           ; d1,d2
SR:30000A18 E24F002C            sub     r0,pc,#0x2C
SR:30000A1C EBFFFECC            bl      0x30000554      ; Uart_Printf
                58         Uart_Printf("%d\n", d3);
SR:30000A20 E1A01005            mov     r1,r5           ; r1,d3
SR:30000A24 E24F0038            sub     r0,pc,#0x38
SR:30000A28 EBFFFEC9            bl      0x30000554      ; Uart_Printf
                59         Uart_Printf("%d\n", d4);
SR:30000A2C E1A01006            mov     r1,r6           ; r1,d4
SR:30000A30 E24F0044            sub     r0,pc,#0x44
SR:30000A34 EBFFFEC6            bl      0x30000554      ; Uart_Printf
                60         Uart_Printf("%d\n", d5);
SR:30000A38 E1A01007            mov     r1,r7           ; r1,d5
SR:30000A3C E24F0050            sub     r0,pc,#0x50
SR:30000A40 EBFFFEC3            bl      0x30000554      ; Uart_Printf
                61         Uart_Printf("%d\n", d6);
SR:30000A44 E1A01008            mov     r1,r8           ; r1,d6
SR:30000A48 E8BD41F0            ldmia   r13!,{r4-r8,r14}
SR:30000A4C E24F0060            sub     r0,pc,#0x60
SR:30000A50 EAFFFEBF            b       0x30000554      ; Uart_Printf
```

(b) 스택 포인터 전달 방식의 ISR

그림 11. 단순한 데이터 형의 매개변수 전달
Fig. 11. Simple Data Type Parameter Passing.

림 11과 같이, 22개의 명령어 즉 73개의 사이클이 소요되었고, 스택 포인터 전달 방식은 20개의 명령어 즉 66개의 사이클이 소요되었다. 스택 포인터 전달 방식이 스택 포인터를 이용하여 매개변수를 참조하기 위해 캐스팅과정을 겨처 오버헤드가 발생함에도 불구하고 스택 포인터 얻는 방식보다 9.6%효율적이다.

다. 서로 다른 데이터 형의 매개변수의 수행시간 비교

이 실험의 경우, 스택 포인터 얻는 방식은 SWI 함수를 호출하는 API 함수와 프로토타입이 동일하기 때문에 전달되는 매개변수의 데이터 형에 대해 고려하지 않아도 되지만, 스택 포인터 전달 방식은 스택 포인터로만 전달되기 때문에 API 함수의 프로토타입과 동일하게 캐스팅과정이 필요하다.

실험 결과, 스택 포인터 얻는 방식이 전달 방식은 그림 12와 같이, 23개의 명령어 즉 70개의 사이클이 소요되었고, 스택 포인터 전달 방식은 21개의 명령어 즉 57개의 사이클이 소요되었다. 스택 포인터 전달 방식이 스택 포인터를 이용하여 매개변수를 참조하기 위해 캐

```
                   void SWI_Various_DataType_ISR(char c,
                                                 int d,
                                                 double f,
                34                               char * str) {
SR:300009D8 E92D40F8  SWI_Vari_:stmdb   r13!,{r3-r7,r14}
SR:300009DC E1A07001            mov     r7,r1           ; r7,d
                35         Uart_Printf("%c\n", c);
SR:300009E0 E1A01000            mov     r1,r0           ; d,c
SR:300009E4 E28F0038            add     r0,pc,#0x38     ; c,pc,#56
SR:300009E8 E59D4018            ldr     r4,[r13,#0x18] ; str,[r13,#24]
SR:300009EC E1A06003            mov     r6,r3
SR:300009F0 E1A05002            mov     r5,r2
SR:300009F4 EBFFFEDE            bl      0x30000574      ; Uart_Printf
                36         Uart_Printf("%d\n", d);
SR:300009F8 E1A01007            mov     r1,r7           ; r1,d
SR:300009FC E28F0024            add     r0,pc,#0x24
SR:30000A00 EBFFFED8            bl      0x30000574      ; Uart_Printf
                37         Uart_Printf("%f\n", f);
SR:30000A04 E1A02006            mov     r2,r6
SR:30000A08 E1A01005            mov     r1,r5
SR:30000A0C E28F0018            add     r0,pc,#0x18
SR:30000A10 EBFFFED7            bl      0x30000574      ; Uart_Printf
                38         Uart_Printf("%s\n", str);
SR:30000A14 E1A01004            mov     r1,r4           ; r1,str
SR:30000A18 E8BD40F8            ldmia   r13!,{r3-r7,r14}
SR:30000A1C E28F000C            add     r0,pc,#0x0C
SR:30000A20 EAFFFED3            b       0x30000574      ; Uart_Printf
SR:30000A24 000A6325            dcd     0xA6325
SR:30000A28 000A6425            dcd     0xA6425
SR:30000A2C 000A6625            dcd     0xA6625
SR:30000A30 000A7325            dcd     0xA7325
                   }
```

(a) 스택 포인터 얻는 방식의 ISR

```
                37 void SWI_Various_DataType_ISR(unsigned *reg) {
SR:300009C4 E92D40F8  SWI_Vari_:stmdb   r13!,{r3-r7,r14}
                           char    c;
                           int     d;
                           double  f;
                           char *  str;

                43         c       = *(char*)reg;          reg++;
SR:300009C8 E4D01004            ldrb    r1,[r0],#0x4
                44         d       = *(int*)reg;           reg++;
                45         f       = *(double*)reg;        reg+=2;
                46         str = *(char**)reg;             reg++;
SR:300009CC E89000D0            ldmia   r0,{r4,r6-r7}   ; reg,{r4,r6-r7}
SR:300009D0 E590500C            ldr     r5,[r0,#0x0C]

                48         Uart_Printf("%c\n", c);
SR:300009D4 E28F002C            add     r0,pc,#0x2C     ; reg,pc,#44
SR:300009D8 EBFFFED0            bl      0x30000554      ; Uart_Printf
                49         Uart_Printf("%d\n", d);
SR:300009DC E1A01004            mov     r1,r4           ; c,d
SR:300009E0 E28F0024            add     r0,pc,#0x24
SR:300009E4 EBFFFED4            bl      0x30000554      ; Uart_Printf
                50         Uart_Printf("%f\n", f);
SR:300009E8 E1A02007            mov     r2,r7
SR:300009EC E1A01006            mov     r1,r6
SR:300009F0 E28F0018            add     r0,pc,#0x18
SR:300009F4 EBFFFED6            bl      0x30000554      ; Uart_Printf
                51         Uart_Printf("%s\n", str);
SR:300009F8 E1A01005            mov     r1,r5           ; r1,str
SR:300009FC E8BD40F8            ldmia   r13!,{r3-r7,r14}
SR:30000A00 E28F000C            add     r0,pc,#0x0C
SR:30000A04 EAFFFED2            b       0x30000554      ; Uart_Printf
SR:30000A08 000A6325            dcd     0xA6325
SR:30000A0C 000A6425            dcd     0xA6425
SR:30000A10 000A6625            dcd     0xA6625
SR:30000A14 000A7325            dcd     0xA7325
                }
```

(b) 스택 포인터 전달 방식의 ISR

그림 12. 복잡한 데이터 형의 매개변수 전달
Fig. 12. Complex Data Type Parameter Passing.

스팅과정을 겨처 오버헤드가 발생함에도 불구하고 스택 포인터 얻는 방식보다 19%효율적이다.

Ⅳ. 결 론

본 논문에서는 SWI 함수의 다중 호출과 매개변수의 효율적인 이용방법으로 일반적인 소프트웨어 인터럽트의 사용방법의 문제점을 해결하고자 Nested Call과 SWI의 서비스 루틴에 매개변수를 효율적으로 전달하는 방식을 설계하고 구현하였다.

SWI의 Nested Call은 Callee의 SPSR을 이용하여 Caller에 대한 프로그램 정보를 저장함으로써 구현이 가능하였고 간단한 예로 동작이 올바르게 동작하는 것을 살펴보았다. 매개변수 이용 방법에서 스택 포인터

얻는 방법은 SPSR을 이용하여 Caller의 스택 포인터를 구하면서 오버헤드가 발생하였고, 스택 포인터 전달 방식은 스택 포인터를 전달하기 위해 매개변수를 스택에 저장하고 스택 포인터를 재지정하는 과정에서 오버헤드가 발생하였다. 결국은 두 매개변수 이용 방식의 가장 큰 차이점은 속도와 가독성이다. 스택 포인터 전달 방식은, 가독성이 떨어지지만 속도적인 측면에서는 성능이 약 19% 더 좋은 것을 알 수 있었다. 앞으로는 본 논문에서 제시한 스택 포인터 전달 방식의 가독성이 떨어지는 부분에 대한 연구를 할 예정이다.

참 고 문 헌

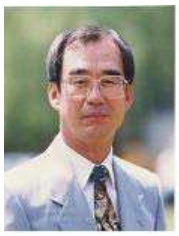
[1] 히언, “EMBEDDED RECIPES-3쇄”, 코너북, 2009.
[2] ARM Limited, “ARM DDI 0100E: ARM Architecture Reference Manual”, 1996.
[3] 안효복, “ARM으로 배우는 임베디드 시스템”, 한빛미디어, 2006년
[4] ARM Limited, “ARM DUI 0040D: ARM Developer Suite Developer Guide”, pp. 9_16-9_17, 1999.
[5] 친절한 임베디드 시스템 개발자 되기 강좌, “http://recipes.egloos.com/”
[6] ARM Limited, “ARM DAI 0030A: Application Note 30”, 1996.
[7] ARM Limited, “ARM IHI 0042D: Procedure Call Standard for the ARM Architecture”, 2009.
[8] ARM Limited, “ARM DDI 0180D: ARM9TDMI Technical Reference Manual Rev3”, 2000.

저 자 소 개

한 길 종(학생회원)
2009년 인천대학교 임베디드 시스템 공학과 학사
2009년 아주대학교 전자공학과 석사과정
<주관심분야 : 임베디드시스템, 플랫폼 설계, 펌웨어 설계>

류 경 식(학생회원)
1991년 아주대학교 전자공학과 학사
1993년 아주대학교 전자공학과 석사
2006년 아주대학교 전자공학과 박사수료
2000년～현재 (주)월텍 대표이사
1993년～1997년 (주)인켈 기술연구소
1997년～1998년 (사)고등기술연구원 정보통신연구실
1998년～2000년 유레카시스템 대표
1998년～2000년 용인송담대학 겸임교수
<주관심분야 : 임베디드시스템, 통신>

이 정 원(정회원)
1993년 이화여자대학교 전자계산학과 학사
1995년 이화여자대학교 전자계산학과 석사
2003년 이화여자대학교 컴퓨터학과 박사
1995년～1997년 LG종합기술원 주임연구원
2003년～2006년 이화여자대학교 컴퓨터학과 BK교수, 전임강사(대우)
2006년～현재 아주대학교 정보통신대학 전자공학부 조교수
<주관심분야 : SOA, 유비쿼터스 컴퓨팅, 임베디드 소프트웨어>



류 경 식(정회원)
1971년 연세대학교 전자공학과 학사
1973년 연세대학교 전자공학과 석사
1978년 연세대학교 전자공학과 박사
1979년～현재 아주대학교 전자공학부 교수
1973년～1974년 불란서 E.S.E 전자공학 연구실
1973년～1974년 미국 Stanford대학교 연구교수
1981년～1982년 한국전자통신연구소 위촉연구위원
1994년～1998년 ITS 연구기획단연구위원 전자부문 총괄
<주관심분야 : 통신, 컴퓨터, ITS>