https://doi.org/10.5392/JKCA.2020.20.10.014

윈도우10에 실시간 성능을 제공하기 위한 타이머 구현 및 성능 측정

Timer Implementation and Performance Measurement for Providing Real-time Performance to Windows 10

이정국, 이상길, 이철훈
충남대학교 컴퓨터공학과

Jeong-Guk Lee(jeonggguk.lee.k@o.cnu.ac.kr), Sang-Gil Lee(sk0137@cnu.ac.kr), Cheol-Hoon Lee(clee@cnu.ac.kr)

요약

실시간 성능이란 정확한 주기에 정확한 결과값을 반환하거나, 일정 주기마다 정해진 일을 수행하는 것이다. 윈도우는 실시간 성능을 지원하지 못하므로 RTX나 INtime과 같은 고가의 서드파티를 사용하여 실시간 성능을 지원한다. 본 논문은 윈도우에 디바이스 드라이버 형태로 동작하는 실시간 커널인 RTiK을 통해 윈도우에 실시간 성능을 지원하고자 한다. 윈도우 7에서 RTiK은 x86 하드웨어에서 지원하는 Local APIC를 이용한 타이머를 사용하였다. 하지만 윈도우 10에서 KPP(Kernel Patch Protection)으로 인해 Local APIC 타이머를 사용하는 것이 불가능해졌다. 이에 Local APIC IPI를 사용하여 정해진 주기를 알리는 타이머를 구현하였고 성능 측정을 수행하여 주기가 오차범위 내에서 정상 동작함을 확인하였다. 이를 통해 윈도우 10에서 실시간 성능 제공을 가능하게 하였다.

■ 중심어 : | 실시간 시스템 | 실시간 운영체제 | RTiK |

Abstract

Real-time performance is to return the exact result value to the correct cycle, or to perform the specified work at a certain cycle. Windows does not support real-time performance, so it supports real-time performance using expensive third parties such as RTX and INtime. This paper aims to support real-time performance of Windows through RTiK, a real-time kernel that operates in the form of a device driver in Windows. In Windows 7, RTiK used a timer using local APIC supported by x86 hardware. However, due to the Kernel Patch Protection (KPP) on Windows 10, it became impossible to use a local APIC timer. Therefore, a timer is implemented to inform the determined cycle using Local APIC IPI, and performance measurement is performed to confirm that the cycle operates normally within the error range. This enables real-time performance on Windows 10.

■ keyword : | Real-Time System | RTOS | RTiK |

I. 서 론

실시간 성능이란 정확한 주기에 정확한 결과 값을 반환하거나, 일정한 주기마다 정해진 일을 수행하는 것이다[1]. 이러한 실시간 성능을 보장하기 위해 실시간 운영체제(Real-Time Operating System, RTOS)를 탑재한다[2]. 실시간 운영체제는 정해진 시간 동안 작업을 완료해서 결과를 도출하기 위해 태스크의 문맥교환에

* 이 연구는 충남대학교 학술연구비에 의해 지원되었음
접수일자 : 2020년 07월 24일
수정일자 : 2020년 08월 19일
심사완료일 : 2020년 08월 20일
교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

걸리는 시간이 짧고, 안정적이면서 스케줄러가 태스크들을 실행시키는 시간에 대한 오버헤드가 적다[3].

이러한 실시간 운영체제는 실시간 성능을 요구하는 시스템의 운영체제로서 해당 시스템의 하드웨어는 별도로 제작하는 경우가 많다. 이러한 하드웨어를 설계 및 제작하여 RTOS를 포팅하는 것은 개발 비용과 개발 기간이 증가한다는 문제가 있다. 따라서 표준화된 x86 기반의 하드웨어에 윈도우를 설치하는 것이 개발 비용의 절감과 개발 기간을 단축하는데 도움이 될 수 있을 것이다.

하지만 윈도우의 경우 범용 운영체제로서 다양한 서비스 및 프로그램을 균등하게 수행할 수 있도록 하는 라운드 로빈(Round-Robin) 스케줄링 알고리즘을 사용하기 때문에 요청되는 작업에 대한 논리적 정확성 뿐만 아니라 시간적 제약을 만족시키는 실시간 성능을 보장할 수 없다[1][4]. 이러한 이유로 윈도우에 실시간 성능을 제공하기 위해 RTX나 INTime과 같은 서드파티(Third Party)를 사용하게 된다[5][6]. 하지만 이러한 서드파티들은 고가의 제품 구입비용과 라이선스 비용 등을 요구하기 때문에 제품 개발에 있어 개발 비용이 증가한다는 문제점이 있다. 따라서 고가의 개발비용을 절감하기 위해 이러한 서드파티들을 대체하여 윈도우에 실시간 성능을 제공할 수 있는 방법에 대한 연구가 필요하다[7].

위와 같은 서드파티들을 대체하기 위해 RTiK은 윈도우 기반 장비에 실시간 성능을 제공할 수 있도록 개발되었다. RTiK은 윈도우 XP에 디바이스 드라이버 형태로 설치되어 인텔 x86의 Local APIC(Advanced Programmable Interrupt Controller)의 Timer 인터럽트를 사용해 윈도우와 독립적인 타이머를 구현하여 윈도우에 실시간 성능을 제공할 수 있었다[8].

이러한 RTiK은 윈도우 XP 기반의 유도탄 점검장비에 실시간 성능을 지원할 수 있도록 개발되어 0.1ms의 주기의 성능을 제공할 수 있었다[9]. 그리고 윈도우 XP 기반 로봇 컴포넌트에 실시간 성능을 지원할 수 있도록 개발되어 1ms 주기를 만족시킬 수 있었다[10]. 그리고 싱글프로세서뿐만 아니라 멀티프로세서 윈도우 XP상에서도 실시간 성능을 지원할 수 있도록 개발되었다[7].

2014년 4월 8일 윈도우 XP에 대한 지원이 종료될 예정이었기 때문에 RTiK을 개발할 윈도우의 버전이 윈도우 XP에서 윈도우 7으로 변화하였다[11]. 윈도우 7은 윈도우 XP처럼 APIC의 Timer 인터럽트를 사용해 타이머를 구현하여 실시간 성능을 제공한다. 멀티프로세서 기반의 윈도우 7이 설치되어 있는 유도무기 휴대용 점검장비가 실시간 성능을 제공할 수 있도록 RTiK이 설계 및 구현되었고, 성능을 검증하기 위해 유도무기 휴대용 점검장비와 오실로스코프를 사용한 실험 환경에서 주기를 측정하였더니 1ms 주기를 만족하는 결과가 나왔다[12].

하지만 윈도우 7은 2020년 1월 14일부로 지원이 종료되어 마이크로소프트 사는 윈도우 10으로 업그레이드 할 것을 권장했다[13].

그러나 윈도우 10으로 업그레이드 하는 경우 KPP(Kernel Patch Protection)으로 인해 IDT를 수정하는 Local APIC Timer 인터럽트를 사용하는 것이 불가능하게되어 기존 방법과는 다른 방향으로 타이머 구현이 필요하게 되었다[14].

본 논문에서는 윈도우 10에 실시간 성능을 제공하기 위해 IPI(InterProcessor Interrupts)를 사용하여 새로운 타이머를 구현하고 성능을 측정하였다. 2장에서는 관련 연구인 서드파티에 대해 설명한다. 3장에서는 윈도우 10에 실시간 성능을 제공하기 위한 타이머의 변경 방안을 설명한다. 4장에서는 구현된 타이머의 성능을 측정하기 위한 방법을 소개하고 성능을 측정하여 5장에서 결론을 맺는다.

II. 관련 연구

1. 서드파티

1.1 RTX

IntervalZero사의 RTX(Real-Time eXtension) 실시간 소프트웨어는 윈도우를 실시간 운영체제로 변환한다. RTX 자체는 순수한 실시간 운영체제가 아니라 개발자들에게 있어 익숙하고 편리한 윈도우의 환경 하에서 풍부한 GUI Library의 장점을 최대한 이용하여 윈도우가 제공할 수 없는 실시간 성능의 제약사항을 보완해 주는 소프트웨어이다.

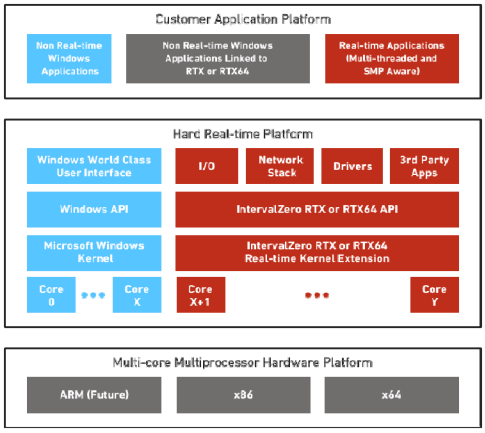

그림 1. RTX의 구조[5]

RTX는 [그림 1]과 같이 멀티프로세서 기반의 윈도우에 실시간 성능을 제공하기 위해 Dedicated Mode를 사용하여 하나 이상의 코어를 RTX가 독점하여 실시간 성능을 제공한다. 독점된 코어에서는 RTX의 실시간 성능을 제공하는 프로그램만 동작하게 된다[5].

이러한 RTX는 산업전자, 국방, 항공, 계측기, 시뮬레이션, 의료 및 Robotics 등의 다양한 Application에 적용되어 온 검증된 소프트웨어이다.

1.2 INTime

TenAsys사의 INTime은 윈도우와 함께 실시간 응용 프로그램을 나란히 실행하기 위해 구현되었다. 윈도우의 INTime은 멀티 코어 호스트 하드웨어를 [그림 2]와 같이 실행 코어, RAM, I/O리소스를 전용하는 처리 노드로 분할하여 실시간 어플리케이션에 필요한 리소스를 제공한다.

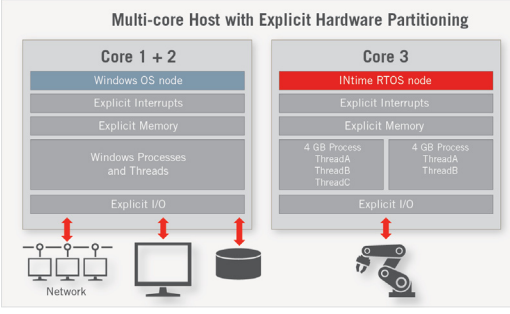

그림 2. INTime의 하드웨어 분할[6]

INTime RTOS 노드와는 별도로 SMP(Symmetric Multi-Processing;대칭 멀티프로세싱)를 사용하여 다른 코어들이 윈도우의 애플리케이션을 계속 실행하도록 한다.

그리고 INTime은 RTOS 애플리케이션이 서로 독립적으로 실행될 수 있도록 AMP(Asymmetric Multi-Processing;비대칭 멀티프로세싱) 접근법을 사용한다. INTime 역시 [그림 2]에서 볼 수 있듯이 RTX처럼 하나 이상의 코어를 INTime용으로 전용하여 실시간 성능을 제공하는 프로세스만 동작할 수 있도록 한다[6].

III. 윈도우10에 실시간 성능 제공을 위한 타이머 변경 방안

기존 윈도우XP와 윈도우7에서는 Local APIC의 타이머 인터럽트를 통해 윈도우와는 독립적인 타이머 인터럽트를 발생시킬 수 있었다. 하지만 윈도우 10에서는 KPP로 인해 기존의 방식을 사용할 수 없게 되어 Local APIC의 IPI를 이용하여 타이머를 구현한다.

1. 기존구조

기존 윈도우7에서 동작하던 RTiK은 다음과 같은 구조를 가지고 있다.

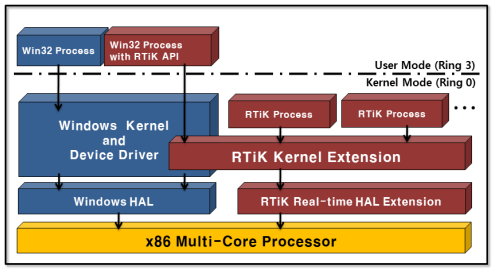

그림 3. RTiK의 구조도

[그림 3]에서와 같이 RTiK은 윈도우 디바이스 드라이버의 형태로 이식된다. 이를 통해 RTiK이 윈도우 커널 레벨에서 동작하여 하드웨어로의 접근이나 윈도우 커널 자원에 대한 접근을 가능하게 한다. RTiK의 하드

웨어 추상화 계층(Hardware Abstration Layer)은 x86 기반 하드웨어 플랫폼에 액세스하여 Local APIC를 제어할 수 있게 하고, Local APIC 타이머 레지스터의 제어는 윈도우와 독립적으로 타이머 인터럽트를 발생시킬 수 있다. 인터럽트 발생 시 RTiK의 타이머 인터럽트 서비스 루틴이 실행되어 중요한 작업을 수행하고 나머지 작업은 지연처리호출(Deferred Procedure Call : DPC)에서 CPU의 권한을 받아 수행하게 된다[4][10].

Local APIC 타이머 관련 레지스터는 다음 [그림 4]과 같다.

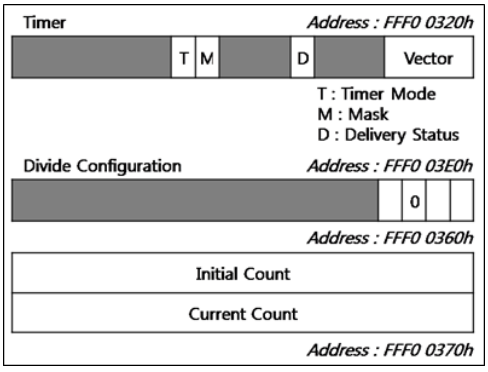

그림 4. Local APIC 타이머 관련 레지스터

타이머 레지스터는 타이머 인터럽트 모드, 마스크 상태, 인터럽트 전달상태 및 인터럽트 벡터 번호와 같은 타이머 상태정보를 가지고 있는 레지스터이다.

Initial Count 레지스터는 타이머의 주기를 결정하는 레지스터로서 카운트 값을 설정한다. Current Count 레지스터는 Initial Count 레지스터의 카운트 값을 복사하여 레지스터의 카운트 값이 버스 클럭에 의해 감소하여 0에 도달하면 타이머 인터럽트를 발생하게 된다. Local APIC로부터 인터럽트 발생을 수신한 프로세서는, 타이머 레지스터의 Vector 비트, 즉 IDT의 주소를 나타내는 인덱스를 참조함으로써 타이머 인터럽트 서비스 루틴을 수행한다[15][16]. 타이머 레지스터의 타이머 모드가 Periodic(0x01)이면 타이머 인터럽트가 발생하게 되었을 때 Initial Count 레지스터의 카운터 값이 Current Count 레지스터로 다시 복사되고 동일한 프로세스를 반복하게 된다[12]. 이렇게 윈도우와는 독립된 타이머 인터럽트를 발생하는 과정을 통해 주기적인 동작을 보장할 수 있었다.

이러한 타이머 인터럽트를 처리하기 위해 IDT에 인터럽트 오브젝트를 등록시켜야 한다. 하지만 윈도우 10에서는 KPP로 인해 드라이버가 IDT를 수정하는 것이 금지되어 기존구조에 대한 변경이 필요하게 되었다[14].

2. IPI를 이용한 타이머의 동작구조

2.1 IPI

Local APIC는 x86하드웨어의 PIC(Programmable Interrupt Controller)의 확장으로 프로그램 가능한 인터럽트 컨트롤러이다[9]. IPI는 프로세서 간에 전송되는 인터럽트를 의미한다[17]. IPI는 Local APIC의 하위 ICR(Interrupt Command Register)와 상위 ICR의 설정으로 구성되어 있다[18]. ICR은 64비트이며, 구성과 필드별 역할은 다음 [그림 5]과 같다.

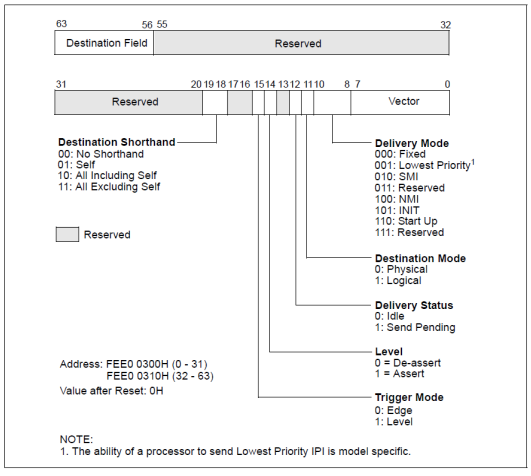

그림 5. ICR의 구조[19]

하위 ICR에 데이터를 쓰게 되면 상위 ICR의 Destination Field에 설정한 Local APIC의 ID를 갖는 프로세서로 IPI를 전달하게 된다. IPI가 발생하면 해당 인터럽트를 받은 프로세서는 인터럽트 서비스 루틴을 수행한다. IPI는 29의 IRQL을 부여받아 높은 우선순위를 갖는다[20]. 따라서 인터럽트를 빠르게 처리할 수 있다.

2.2 타이머의 동작구조

타이머는 다음 [그림 6]와 같이 동작한다.

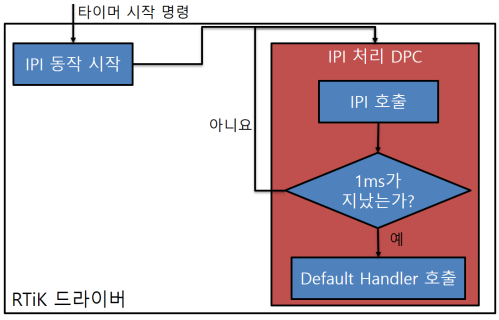

그림 6. IPI 타이머 동작 과정

타이머 시작 명령을 받으면 Local APIC IPI를 발생시킨다. IPI를 수신한 프로세서에서는 인터럽트 서비스 루틴이 수행되고 인터럽트 서비스 루틴에서는 지연처리를 위해 DPC루틴을 등록한다. 인터럽트 서비스 루틴이 완료되면 차후에 DPC Dispatcher가 수행되고, 인터럽트 서비스 루틴에서 등록한 DPC가 수행된다.

DPC 내부에서 다시 IPI를 발생하도록 하여 다시 인터럽트가 발생되게 하고 DPC의 수행을 반복하도록 한다.

폴링 방식으로 동작하여 DPC가 실행될 때마다 해당 시점의 시간을 매번 검사해 바로 직전의 Default Handler를 호출한 이후로 1ms가 지났는지 확인하고 1ms가 지났다면 다시 Default Handler를 호출하도록 한다[21].

Default Handler는 내부 함수를 호출하여 프로그램에서 원하는 주기가 되었는지 확인하고 원하는 주기에 도달하면 이벤트를 발생시켜 프로그램에 전송하도록 하는 함수이다.

1ms를 판단하는 방법은 다음과 같다.

1) IPI를 처리하는 DPC내에서 100ns의 intervals를 갖는 KeQuerySystemTimePrecise()함수[22]를 사용하여 현재 시점을 변수에 저장함

2) 직전의 Default Handler를 호출한 시점과 저장된 현재 시점의 차이가 10,000(1ms)보다 큰 경우 현재 시점을 Default Handler를 호출한 시점으로 저장하고 Default Handler를 호출함

3) 1), 2) 과정을 반복한다.

이러한 방식으로 1ms 주기의 타이머를 설계하였다. 타이머는 1ms의 주기마다 Default Handler를 호출하도록 하여 원하는 주기 작업을 수행할 수 있도록 하였다.

3. 타이머 구현방안

타이머는 시작 명령을 받으면 해당함수를 호출하여 IPI를 발생한다. DPC로 인터럽트 처리가 넘어가게 되면 현재 시점을 저장하고 다시 IPI를 발생시키고 직전 Default Handler의 호출시점과 현재 시점을 비교해 1ms이상이면 Default Handler를 호출하는 방식으로 타이머를 구현하였다. 구현을 위한 소프트웨어 환경은 다음 [표 1]과 같다.

표 1. 구현을 위한 소프트웨어 환경

항목	사양
CPU	Intel 8700K 3.70 GHz
RAM	16GB
OS	Windows 10 Enterprise 1809
IDE	Visual Studio 2017

```
case RTiK_IOCTL_ENABLE:
{
    pRTikPassData passData = (pRTikPassData)buffer;
    unsigned int vector;
    vector = passData->vector;
    status = rtik_systimer_start(vector);
}
```

그림 7. 타이머 시작 명령을 받으면 처리하는 코드

[그림 7]과 같이 디바이스 드라이버가 타이머 시작 명령을 받으면 rtik_systimer_start()함수를 호출한다.

```
int rtik_systimer_start(unsigned int vector)
{
    ...
    rtik_systimer_start_request(vector);
    ...
    return 0;
}

int rtik_systimer_start_request(unsigned int vector)
{
    rtik_timer_startHWTimer(vector);
    return 0;
}
```

그림 8. 타이머 관련 처리 코드

[그림 8]과 같이 호출된 rtik_systimer_start()함수에서 rtik_systimer_start_request()함수를 호출한다.

rtik_systimer_start_request()함수에서는 rtik_timer_startHWTimer()함수를 호출하여 IPI를 발생시키도록 한다.

```
int rtik_timer_startHWTimer(unsigned int vector)
{
    ...
    if(*apic_id >> 24 == 2){
        *apic_icr_high = 0x1 << 24;
        *apic_icr_low = ICR_LOWER_REGISTER(MappedVector);
    }
    else{
        *apic_icr_high = 0x2 << 24;
        *apic_icr_low = ICR_LOWER_REGISTER(MappedVector);
    }
    stop = 0;
    return 0;
}
```

그림 9. IPI 생성 코드

[그림 9]와 같이 rtik_timer_startHWTimer()함수에서 IPI를 발생하기 위해 ICR을 설정한다. 하위 ICR을 설정하는 순간 인터럽트를 보내기 때문에 상위 ICR을 먼저 설정하고 하위 ICR를 설정해야 한다. 따라서 상위 ICR에 있는 Destination 필드를 먼저 설정하고 하위 ICR에 있는 필드 중 Level필드와 Trigger Mode필드를 설정한다. 코드에서 하위 ICR을 설정하기 위해 ICR_LOWER_REGISTER라는 매크로를 사용했는데 이는 Level필드와 Trigger Mode필드를 설정하기 위함이다.

```
VOID RTiKDpc(WDFINTERRUPT Interrupt, WDFOBJECT Device)
{
    ....
    KeQuerySystemTimePrecise(&current_qpc);
    if (isr_count == 0) {
        KeQuerySystemTimePrecise(&standard_qpc);
    }
    lapse.QuadPart = current_qpc.QuadPart -
standard_qpc.QuadPart;

    isr_count++;

    if (stop) {
    }
    else{
        if (*apic_id >> 24 == 2) {
            *apic_icr_high = 0x1 << 24;
            *apic_icr_low =
ICR_LOWER_REGISTER(MappedVector);
        }
        else{
            *apic_icr_high = 0x2 << 24;
            *apic_icr_low =
ICR_LOWER_REGISTER(MappedVector);
        }
        if(lapse.QuadPart > 10000){
            standard_qpc.QuadPart = current_qpc.QuadPart;
            rtik_systimer_defaultHandler();
        }

    }
    return;
}
```

그림 10. IPI 처리 및 타이머 동작 코드

[그림 10]과 같이 IPI가 발생하고 DPC로 인터럽트 처리가 넘어가게 되면 RTiKDpc()함수에서 100ns의 intervals를 갖는 KeQuerySystemTimePrecise()함수를 사용하여 현재 시점을 변수에 저장한다. 그리고 저장되어 있는 직전의 Default Handler를 호출했던 시점과 현재 시점의 차이를 계산한다. 그리고 다시 ICR을 설정하여 IPI를 발생시킨다.

Default Handler를 호출했던 시점과 현재 시점의 차이가 10000(1ms)보다 크면 현재 시점을 Default Handler를 호출한 시점으로 저장하고 Default Handler인 rtik_systimer_defaultHandler()를 호출하여 원하는 주기 작업을 수행할 수 있도록 한다.

IV. 실험 환경 및 결과

실험을 위해 윈도우 10이 설치된 PC를 이용하여 테스트를 진행하고자 한다. 테스트를 위해 실험용 PC에 RTiK 디바이스 드라이버를 설치하고 타이머를 제어하는 프로그램과 타이머의 주기성을 확인하는 테스트 프로그램을 만들어 실행시켜 주기값을 측정한다.

1. 실험 환경

표 2. 실험 환경의 PC 사양

항목	사양
CPU	Intel 8700K 3.70 GHz
RAM	16GB
OS	Windows 10 Enterprise 1809

윈도우10에 실시간 성능을 제공하는 타이머를 검증하기 위해 [표 2]과 같은 실험 환경을 구성하였다. 윈도우10에 디바이스 드라이버를 설치하고 타이머 제어 프로그램을 실행하여 타이머를 제어하였다. 이후 타이머 확인 프로그램을 실행시켜 동작 주기를 입력하고 해당 주기를 측정한 값을 파일에 저장하고 프로그램 종료후 이를 확인하였다.

2. 테스트 프로그램

2.1 프로그램 구현 환경

표 3. 구현 환경의 PC 사양

항목	사양
CPU	Intel 8700K 3.70 GHz
RAM	16GB
OS	Windows 10 Enterprise 1809
IDE	Visual Studio 2017

윈도우10에 실시간 성능 검증을 위한 테스트 프로그램을 구현하기 위해 [표 3]과 같은 구현 환경을 구성하였다.

2.2 전체적인 동작 구조

테스트 프로그램의 전체적인 구조는 [그림 11]과 같다.

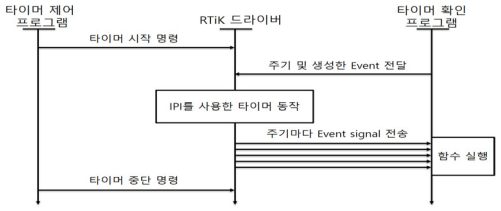

그림 11. 전체적인 동작 구조

타이머 제어 프로그램에서 드라이버에 타이머의 시작/중단 명령을 전송하고 타이머 확인 프로그램은 디바이스 드라이버에 주기 및 생성한 이벤트를 전달하고 해당 주기마다 디바이스 드라이버로부터 이벤트를 수신받는다. 타이머 제어 프로그램과 타이머 확인 프로그램은 다음과 같이 구현했다.

2.3 타이머 제어 프로그램

```
ch = getchar();

switch(ch){
case 'e':
case 'E':
      printf(" STart Timer -- )\n");
      rtik_coreStartTimer();

      break;

case 'd':
case 'D':
      printf("Stop Timer --)\n");
      rtik_coreStopTimer();

      break;
```

그림 12. 사용자 입력에 따른 타이머 시작/중단 코드

[그림 12]의 코드는 사용자로부터 값을 입력받아 타이머를 시작하거나 중단할 수 있도록 한다. e, E를 입력하면 타이머 시작 명령을 전송하는 rtik_coreStartTimer()함수가 호출된다. d, D를 입력하면 타이머 중단 명령을 전송하는 rtik_coreStopTimer()함수가 호출된다.

```
int rtik_coreStartTimer(void)
{
      RTiK_Pass_data* passData = getPassData();
      passData->vector = GlobalVector;
      DWORD  bytesReturned;
      if (!DeviceIoControl(mDriverHandle,
            RTiK_IOCTL_ENABLE,
            passData,
            sizeof(RTiK_Pass_data),
            NULL,
            0,
            &bytesReturned,
            NULL)) {
                  printf(" start Timer Failed ... \n");
                  return -1;
      }
      return 0;
}
```

그림 13. 타이머 시작 명령 전송 코드

[그림 13]에서의 rtik_coreStartTimer()함수에서 디바이스 드라이버에 RTiK_IOCTL_ENABLE를 전송하여 타이머를 동작시키도록 한다. 이는 [그림 7]의 코드에서 타이머 시작 명령으로 나타난다.

```
int rtik_coreStopTimer(void)
{
      DWORD  bytesReturned;
      if (!DeviceIoControl(mDriverHandle,
            RTiK_IOCTL_DISABLE,
            NULL,
            NULL,
            NULL,
            0,
            &bytesReturned,
            NULL)) {
                  printf(" start Timer Failed ... \n");
                  return -1;
      }
      return 0;
}
```

그림 14. 타이머 중단 명령 전송 코드

[그림 14]에서의 rtik_coreStopTimer()함수에서 디바이스 드라이버에 RTiK_IOCTL_DISABLE를 전송하여 타이머를 중단시키도록 한다.

2.4 타이머 확인 프로그램

```
int readParam(void){
    int periodic = 0;
    printf("insert periodic (ms) : ");
    scanf_s("%d", &periodic);
    printf("insert task loop count(infinity loop : -1) : ");
    scanf_s("%d", &loop_count);
    printf("insert vector : ");
    scanf_s("%x", &GlobalVector);
    return periodic;
}
```

그림 15. 주기, 카운트 등 사용자 입력 코드

[그림 15]와 같이 사용자로부터 주기값과 카운트값을 입력받아 쓰레드를 생성한다.

```
int rtik_coreAddTask(void)
{
    RTiK_Pass_data* passData = getPassData();
    DWORD bytesReturned;

    if (!DeviceIoControl(mDriverHandle,
        RTiK_IOCTL_PROCESS_INSERT,
        passData,
        sizeof(RTiK_Pass_data),
        NULL,
        0,
        &bytesReturned,
        NULL)) {
            printf(" 드라이버 오픈 실패 \n");
            return -1;
    }
    return 0;
}
```

그림 16. 주기 및 생성한 이벤트 전달 코드

[그림 16]과 같이 디바이스 드라이버에 입력받은 주기와 생성한 이벤트를 passData에 넣어 전달한다.

```
while (threadFlag) {
    WaitStatus = WaitForSingleObject(waitEventHandle,
INFINITE);
    if (WaitStatus != WAIT_OBJECT_0){
        printf("Driver 에서 event를 signal 시키는데 실패했다.\n");
    }
    else {
        rtik_coreAckMsg();
        count++;
        QueryPerformanceCounter(&clockval);
        timearray[k % 100000].QuadPart = clockval.QuadPart;
        ...
        if (k >= loop_count){
            QueryPerformanceFrequency(&freq);
            threadFlag = NO;
            break;
        }
        k++;
    }
}
k = 0;
...
for (int j = 1; j<100000; j++){
    fprintf(f, "%.10f\n", (double)((timearray[j % 100000].QuadPart
- timearray[j % 100000 - 1].QuadPart))/ (freq.QuadPart));

}
            fclose(f);
```

그림 17. 주기마다 수행하는 코드

[그림 17]의 WaitForSingleObject() 함수를 통해 정해진 주기마다 디바이스 드라이버에서 보내는 이벤트 시그널 수신을 기다리게 된다. 이벤트 시그널을 수신하게 되면 rtik_coreAckMsg()함수를 통해 ACK 메시지를 전송하여 이벤트 시그널을 수신했다는 것을 디바이스 드라이버에 알린다. 그리고 이벤트 시그널을 수신한 시간으로 QueryPerformanceCounter()함수를 호출하여 Perfomance counter의 값을 측정하여 저장한다[23].

카운트 값이 [그림 15]에서 입력받은 카운트값보다 크거나 같은 경우 QueryPerformanceFrequency()함수를 호출하여 현재 Performance counter의 주파수를 획득한다[23]. 그리고 획득한 주파수를 이용하여 이벤트 시그널을 수신한 시간의 차이를 주파수값으로 나눠 1초 단위로 계산한 주기를 파일에 저장하고 쓰레드를 종료한다.

3. 실험 결과

본 논문에서는 주기를 1ms, 5ms, 10ms로 설정해 10,000회 반복을 수행하여 설정한 주기에 어느 정도의 오차가 있는지 측정하였다. 오차는 실제 측정된 주기의 최대값과 최소값을 바탕으로 계산하였다.

[그림 18], [그림 19], [그림 20]은 주기 1ms, 5ms, 10ms에 대한 주기 측정 결과이다. 그래프의 x축은 측정 횟수를 y축은 주기 값을 나타낸다. 데이터의 오차가 적을수록 기준값에 가까이 나타나게 된다.

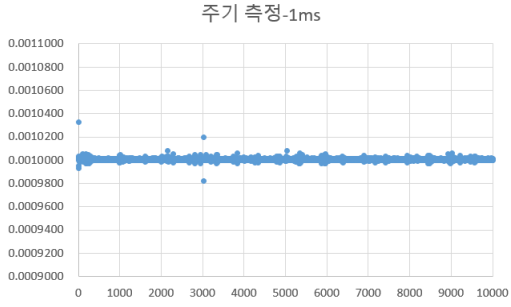

그림 18. 주기 측정(1ms)

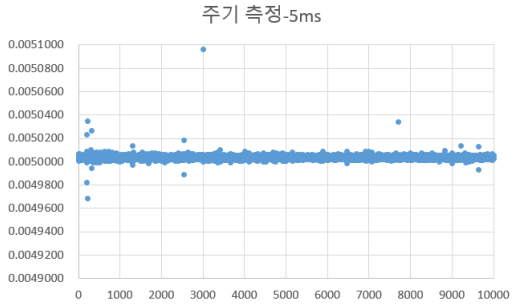

그림 19. 주기 측정(5ms)

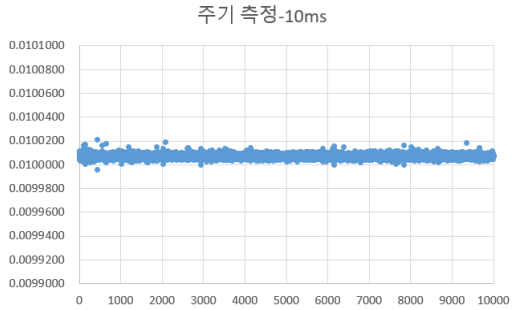

그림 20. 주기 측정(10ms)

상세 결과는 다음 [표 4]와 같다.

표 2. 주기 측정의 상세 결과

주기	1ms	5ms	10ms
최대값	1.0329000 ms	5.0963000 ms	10.0211000 ms
최소값	0.9822000 ms	4.9687000 ms	9.9960000 ms
평균	1.0007028 ms	5.0035826 ms	10.0071669 ms

상세 결과를 살펴보면 1ms 주기에서 0.9822ms~1.0329ms 사이에 측정값이 존재한다는 것을 확인할 수 있었고 오차 범위는 3.3%이다. 5ms 주기에서는 4.9687ms~5.0963ms 사이에 측정값이 존재한다는 것을 확인할 수 있으며 오차범위은 1.9%이다. 마지막으로 10ms 주기의 경우 9.9960ms~10.0211ms 사이에 측정값이 존재한다는 것을 확인할 수 있었고 오차범위는 0.2%이다.

1ms, 5ms, 10ms의 결과를 살펴보면 주기가 클수록 오차범위는 작아지는 것을 확인할 수 있다. 오차범위가 가장 큰 1ms주기에서 3.3% 오차 이내로 측정된 것을 보아 모든 주기 값을 크게 벗어나지 않고 비교적 안정적으로 동작하는 것을 확인할 수 있다.

V. 결론 및 향후 연구

윈도우 10에서 실시간 성능을 제공하기 위해 고가의 서드파티를 사용하게 된다. 이러한 서드파티를 대체하여 윈도우에 실시간 성능을 제공할 수 있는 방법에 대한 연구가 필요하다. 기존에 설계 및 구현되었던 RTiK은 윈도우 7에 디바이스 형태로 설치되어 Local APIC Timer를 사용하여 실시간 성능을 제공할 수 있었다. 하지만 윈도우 10에서는 KPP로 인해 Local APIC Timer를 사용하는 것이 불가능해졌다.

이에 본 논문에서는 윈도우 10에 실시간 성능을 제공하기 위한 방법으로 Local APIC IPI를 이용하여 타이머를 구현하였다. 해당 타이머는 IPI가 발생하여 인터럽트 서비스 루틴이 완료되고 인터럽트 서비스 루틴에서 등록한 DPC가 수행되고, DPC내에서 다시 IPI를 발생시키며 해당 시점의 시간을 매번 검사하여 바로 직전의 함수를 호출한 이후로 1ms가 지났는지 확인하고 1ms가 지났으면 함수를 호출하는 방식으로 동작한다.

이렇게 구현된 타이머를 이용하여 주기 작업이 정확하게 이루어지는지 실험해보았다.

실험의 결과는 1ms 주기에서 0.9822ms~1.0329ms 사이에 측정값이 존재한다는 것을 확인할 수 있었고 5ms 주기에서는 4.9687ms~5.0963ms 사이에 측정값이 존재한다는 것을 확인할 수 있으며 10ms 주기의 경우 9.9960ms~10.0211ms 사이에 측정값이 존재한다는 것을 확인할 수 있었다. 이를 보았을 때 모든 주기 값을 크게 벗어나지 않고 안정적으로 동작하는 것을 확인할 수 있다.

기존 연구에서 윈도우 7에서의 Local APIC Timer를 사용하여 타이머의 실시간 성능을 측정해본 결과 1ms주기에서 0.9716ms~1.0159ms 사이로 측정값이 존재하고 10ms 주기에서 9.9939ms~10.0105ms 사이로 측정값이 존재하는 것을 보면[12] 본 연구에서 나온 결과와 유사하다는 것을 확인할 수 있다.

해당 논문에서 제안한 윈도우10에서 IPI를 이용한 타이머가 기존 연구인 윈도우7에서의 Local APIC

Timer 성능과 유사하게 나타나는 것을 보면 제안한 방식이 우수하다는 것을 확인할 수 있다.

향후 연구로는 IPI의 호출구조로 인한 오차를 줄여 더욱더 안정적인 타이머 성능을 제공할 수 있도록 개선할 필요가 있다. 그리고 사용자들이 타이머를 편리하게 사용하기 위한 POSIX API와 같은 범용적인 API가 필요하다.

참 고 문 헌

[1] 이승율, 이상길, 이철훈, “ARM 프로세서 기반의 리눅스을 위한 실시간 확장 커널,” 한국콘텐츠학회논문지, 제17권, 제10호, pp.587-597, 2017.

[2] 고재환, 최병욱, “실시간 임베디드 리눅스 및 상용 RTOS의 실시간 메커니즘 성능 분석,” 한국조명·전기설비학회 학술대회논문집, pp.310-311, 2012(5).

[3] 신익희, 남경호, 이성엽, 우덕균, 김선태, 김형신, “임베디드 소프트웨어를 위한 FreeRTOS의 실시간 성능 분석,” 한국정보과학회 학술발표논문집, pp.1612-1614, 2016(6).

[4] 이진욱, 김종진, 조한무, 이철훈, “휴대용 점검장비에서 윈도우즈의 지연처리호출(DPC)을 이용한 실시간 이식커널(RTiK)의 설계 및 구현,” 한국콘텐츠학회 종합학술대회 논문집, pp.5-9, 2010(5).

[5] https://www.intervalzero.com/, 2020.07.03.

[6] https://www.tenasys.com/intime-for-windows/, 2020.07.03.

[7] 송창인, 이승훈, 이철훈, “멀티프로세서 윈도우 XP 상에서 실시간성 지원,” 한국컴퓨터정보학회 학술발표논문집, 제20권, 제1호, pp.21-24, 2012.

[8] 김주만, 송창인, 이철훈, “RTiK-Linux 리눅스용 실시간 이식 커널의 설계,” 한국콘텐츠학회논문지, 제11권, 제9호, pp.45-53, 2011.

[9] 이진욱, 조문행, 김종진, 조한무, 박영수, 이철훈, “윈도우 기반의 점검장비에 실시간성을 지원하는 실시간 이식 커널의 설계 및 구현,” 한국콘텐츠학회논문지, 제10권, 제10호, pp.36-44, 2010.

[10] 주민규, 이진욱, 장철수, 김성훈, 이철훈, “윈도우 유저 레벨 로봇 컴포넌트에 실시간성 지원 방법,” 한국콘텐츠학회논문지, 제11권, 제7호, pp.51-59, 2011.

[11] https://www.microsoft.com/ko-kr/microsoft-365/windows/end-of-windows-xp-support, 2020.07.08.

[12] 송창인, 이승훈, 주민규, 이철훈, “멀티프로세서 윈도우즈 상에서 실시간성 지원,” 한국콘텐츠학회논문지, 제12권, 제6호, pp.68-77, 2012.

[13] https://support.microsoft.com/ko-kr/help/4057281/ windows-7-support-ended-on-january-14-2020, 2020.07.08.

[14] https://docs.microsoft.com/en-us/windows-hardwar e/drivers/kernel/driver-x64-restrictions, 2020.07.08.

[15] 김효중, 허용관, 권병기, “윈도우 운영체제 기반의 실시간 점검장비 소프트웨어 설계 및 성능검증,” 한국콘텐츠학회논문지, 제17권, 제10호, pp.1-8, 2017.

[16] 박지윤, 조아라, 김효중, 최정현, 허용관, 조한무, 이철훈, “태블릿 PC 환경의 실시간 처리 기능 지원,” 한국콘텐츠학회논문지, 제13권, 제11호, pp.541-550, 2013.

[17] 모상만, 윤석한, “프로세서간 인터럽트의 전송 재시도 제어,” 한국정보과학회 학술발표논문집, 제23권, 제1A호, pp.347-350, 1996.

[18] 한승훈, *64비트 멀티코어 OS 원리와 구조*, 한빛미디어, 2011.

[19] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, Part 1, 2016.

[20] http://ext2fsd.sourceforge.net/documents/irql.htm, 2020.07.10.

[21] 김희철, “경량 임베디드 디바이스 환경에서 소프트웨어 타이머의 정확성 향상을 위한 오버헤드 보정기법,” 한국산업정보학회논문지, 제24권, 제4호, pp.9-19, 2019.

[22] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kequerysystemtimeprecise, 2020.07.08.

[23] https://zadd.tistory.com/56, 2020.07.10.

저 자 소 개

이 정 국(Jeong-Guk Lee) 정회원

- 2011년 2월 : 인하대학교 정보통신공학과(공학사)
- 2012년 1월 ~ 2015년 12월 : LIG넥스원 연구원
- 2016년 2월 ~ 2018년 4월 : LIG넥스원 선임연구원
- 2019년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

〈관심분야〉 : 임베디드 시스템, 실시간 시스템

이 상 길(Sang-Gil Lee) 정회원

- 2014년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2016년 2월 : 충남대학교 컴퓨터공학과(공학석사)
- 2018년 2월 : 충남대학교 컴퓨터공학과 박사과정 수료

〈관심분야〉 : 실시간 운영체제, 임베디드 시스템

이 철 훈(Cheol-Hoon Lee) 정회원

- 1983년 2월 : 서울대학교 전자공학과(공학사)
- 1988년 2월 : 한국과학기술원 전기 및 전자공학과(공학석사)
- 1992년 2월 : 한국과학기술원 전기 및 전자공학과(공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터 사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터 사업부 선임연구원
- 1994년 2월 ~ 1995년 2월 : Univ. of Michigan 객원연구원
- 1995년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙연구원

〈관심분야〉 : 실시간 시스템, 운영체제, 고장허용 컴퓨팅, 로봇 미들웨어