Journal of the Korea Industrial Information Systems Research Vol. 24 No.4, Aug. 2019 : 9-19
http://dx.doi.org./10.9723/jksiis.2019.24.4.009 ISSN:1229-3741

경량 임베디드 디바이스 환경에서 소프트웨어 타이머의 정확성 향상을 위한 오버헤드 보정기법+

(Overhead Compensation Technique to Enhance the Accuracy of a Software Timer for Light-weight Embedded Devices)

김 희 철[1)*]
(Hiecheol Kim)

요 약 경량 임베디드 디바이스가 저전력 네트워킹뿐만 아니라 고정밀 센서 데이터 획득과 같은 영역에서 널리 활용되면서 소프트웨어 타이머에 대한 높은 시간정확성이 요구된다. 이 논문은 경량 MCU(Micro controller unit)를 장착한 임베디드 디바이스 환경에서 소프트웨어 타이머의 정확성 문제를 다룬다. 먼저, 소프트웨어 타이머의 전형적 구현 모델을 구현할 때 오차를 발생시키는 주요 오버헤드의 유형을 면밀히 분석한 후에 실제 환경에서 오버헤드를 측정한다. 이 오버헤드를 타이머 설정주기에 반영하는 오버헤드 보정 기법을 통해 소프트웨어 타이머의 정확성을 향상시킬 수 있다는 점을 검증한다.

핵심주제어 : 소프트웨어 타이머, 타이머 정확성, 오버헤드 보정, IEEE 802.15.4,

Abstract As light-weight embedded devices become widely used in the area of low-power networking and high-precision sensor data acquisition, support for time-critical applications becomes essential for the light-weight embedded devices. This paper addresses the accuracy issue of a software timer for small or tiny embedded devices equiped with light-weight MCUs(Micro controller units). We first explore the characteristics of overhead in a typical implementation of a software timer, and then measure the overhead through a realistic implementation. Using the measurement result, we propose an overhead compensation technique which reduces the overhead from the hardware timer-tick.

Key Words : Software timer, Timer accuracy, Overhead compensation, IEEE 802.15.4

1. 서 론

IoT(Internet of things)와 WSN(Wireless sensor network)의 급격한 확산에 따라 경량 임베디드 디바이스의 컴퓨팅과 네트워킹 요구가 지속적으로 증가하고 있다. 일반적으로 경량 임베디드 디바이스는 10 Kbytes 내외의 클럭, 10 Kbytes 내외의 데이터 메모리, 100 Kbytes 정도의 프로그램 메모리를 가진 경량 MCU(Micro controller

* Corresponding Author : hckim@daegu.ac.kr
+ 이 논문은 2015학년도 대구대학교 학술연구비 지원에 의한 논문임.
Manuscript received February 27, 2019 / revised July 10, 2019 / accepted July 22, 2019
1) 대구대학교 정보통신공학부, 제1저자

unit)를 기반으로 제작되며, 배터리 기반의 저전력 환경에서 작동하는 사례가 많다[1-3]. 이런 자원제약적 상황에서도 경량 임베디드 디바이스의 내부 컴퓨팅 모델은 단순한 단일 스레드 모델에서 전통적인 멀티프로그래밍 또는 멀티스레드 컴퓨팅 모델로 발전하고 있다. 실제로 최근의 IoT와 WSN 영역에서는 멀티스레드 지원과 무선경량네트워킹 스택 탑재 등의 요구로 경량 임베디드 운영체제를 도입해 그 상위에서 필요한 태스크와 네트워크 스택의 구성요소를 작동시키고 있다[4].

경량 임베디드 디바이스는 주로 주기적 또는 비주기적 센서 데이터 획득(Aquisition)과 이웃 노드 간 데이터 교환 등과 같은 작업 위주로 활용된다. 이에 따라, 경량 임베디드 디바이스는 자원제약적 환경에서 운용되지만, 타이머의 요구는 일반적인 범용 컴퓨팅 환경보다 훨씬 크다. 대부분의 경량 MCU는 소수의 하드웨어 타이머를 내장하고 있어, 애플리케이션이나 네트워킹 스택의 구성요소별로 전용 하드웨어 타이머를 제공하는 것이 어렵다. 따라서 한 개의 하드웨어 타이머를 사용해 다수의 가상 타이머를 구현하는 기술인 소프트웨어 타이머를 통해 부족한 하드웨어 타이머 요구를 보충하는 경우가 많다[5].

경량 임베디드 디바이스 환경에서 소프트웨어 타이머의 정확성을 보장하는 것은 매우 중요하다. 센서마다 액세스 시간 등의 차이로 다양한 센서를 장착한 디바이스의 경우에 데이터 획득의 정확성을 높이기 위해서는 정밀한 타이머가 필요하다[6]. 한편, 저전력 무선 통신을 위해서는 인접 노드간 데이터 전송의 동기화를 비롯한 CSMA/CA(Carrier sense multiple access/collision avoidance) 백오프(Backoff) 시간 지연, ACK(Acknowledge) 전송 확인 대기 등과 같이 다양한 해상도의 정밀한 타이머가 요구된다. 실제로 소프트웨어 타이머의 부정확성을 고려해 인한 비콘 수신 시간을 보정하는 기법 등을 비롯해 저전력 무선 통신 스택의 구현에서 위와 같은 타이머와 시간동기화 관련 이슈에 대해 많은 연구가 이루어지고 있다[7-11].

이 논문에서는 경량 임베디드 디바이스 환경에서 소프트웨어 타이머의 정확성에 관한 이슈를 다룬다. 단일 스레드 환경의 프로그래밍 환경에서 소프트웨어 타이머에 구현은 비교적 단순하며, 소프트웨어 타이머에 대한 요구가 작아 이벤트 처리를 위한 오버헤드(Overhead)가 작을 수밖에 없다. 하지만 멀티프로그래밍 환경에서는 소프트웨어 타이머에 요구가 많아 다수의 가상 타이머들이 동시에 작동하는 상황이 발생할 수 있어 해당 이벤트를 내부에 저장하고 관리하는 데에 상대적으로 큰 오버헤드가 발생한다. 아울러 소프트웨어 타이머의 이벤트 처리와 관련한 스케줄링 오버헤드도 소프트웨어 타이머의 시간 오차 발생을 초래한다. 이 논문에서는 이러한 오버헤드를 분석해 정량화한 후, 이 오버헤드를 소프트웨어 타이머 내부에 있는 하위 물리적 타이머의 오버플로 인터럽트 주기에 반영해서 소프트웨어 타이머의 정확도를 향상시키는 실험 연구를 수행한다.

기존 유사 연구로 소프트웨어 타이머의 부정확성으로 인해 발생되는 비콘 동기화 성능저하 현상을 줄이기 위해 비컨수신대기 시간을 앞당기는 보정 기법이 있다[12]. 본 연구는 소프트웨어 타이머의 부정확성의 요인으로 작용하는 내부의 오버헤드를 분석해서 이를 상쇄할 수 있도록 내부 물리적 타이머 주기를 보정하는 새로운 소프트웨어 타이머를 구조를 제시한다. 따라서 비컨수신 보정기법은 소프트웨어 타이머는 그대로 놔둔 채 그 부정확성에 대응해 비컨 동기화의 강건성을 확보하기 위한 네트워크 프로토콜 차원의 기법이라면, 본 연구는 근본적으로 소프트웨어 타이머 자체의 부정확성을 해결하기 위한 운영체제 차원의 연구라는 차이가 있다.

본 논문은 다음과 같이 구성된다. 2절에서는 본 연구 주제와 관련한 배경지식으로 소프트웨어 타이머의 구현 모델을 소개한다. 3절에서 소프트웨어 타이머의 동작 분석과 오차 발생량을 평가하며, 그 내용을 바탕으로 오버헤드 보정 기법을 제시한다. 4절에서는 오버헤드 보정 기능을 갖춘 소프트웨어 타이머가 경량 무선 네트워크의 성능에 미치는 영향을 설명한다. 마지막으로 5절에서 본 연구의 결과에 대한 고찰로 결론을 맺는다.

2. 경량 환경의 타이머 구현

소프트웨어 타이머의 구현은 체계적인 이론보다는 주로 구현 차원에서 다뤄져서 일반적인 구조나 관련 기술적 용어조차도 제대로 정립되어 있지 않은 상황이다. 이 소절에서는 앞으로 전개할 소프트웨어 타이머와 관련한 논의를 위한 배경 지식으로 소프트웨어 타이머의 구현 내용을 간략히 소개한다.

Fig. 1에서 볼 수 있듯이 이 논문에서 구현한 소프트웨어 타이머의 구조는 크게 3 계층으로 구성된다. 먼저, 최상위 계층은 커널(Kernel) 계층이다. 이 계층은 상위 애플리케이션이나 네트워크 스택에서 사용할 수 있는 시간관련 API(Application programming interface) 서비스로 구성된다. 시간 관련 API는 기본적으로 지정된 시간만큼 대기(Wait)하다가 특정 이벤트(Event)를 수행하는 기능 등으로 구성된다.

한편, 커널 아래 계층은 트리거(Trigger) 계층으로 상위에서 요청한 이벤트를 관리하는 역할을 한다. 일반적으로 특정 이벤트는 지정된 지연시간(Delay Interval)과 함께 요청되는데, 이 지연시간이 경과하면 바로 해당 이벤트가 처리되어야 한다. 트리거 계층은 상위에서 요청된 이벤트를 저장·관리하며, 이벤트에 대해 하위 하드웨어 타이머를 작동시켜 타이머 종료 시에 발생한 인터럽트 서비스를 통해 해당 이벤트가 지정된 시간에 처리될 수 있도록 한다. 여기서 이벤트에 대해 하드웨어 타이머를 작동시킨 후에 해당 이벤트 처리를 위한 타이머 인터럽트가 발생하는 것을 "이벤트 트리거"라고 하며 그 시점을 '트리거 시간'이라고 표현한다.

마지막으로 최하위 계층은 물리 계층으로 하드웨어 타이머로 구성된다. 이 계층에서는 상위 계층에 물리적 타이머 서비스를 제공하는 한 개의 일반적인 타이머가 존재한다. 상위 계층의 요청에 따라 타이머가 지정된 경과시간이 설정되고, 해당 시간이 경과되면, 타이머 인터럽트를 발생해서 상위 이벤트를 처리한다.

커널 계층에서는 타이머 관련 API를 호출 후에 해당 타이머 입출력 요청에 따른 대기상태(Waiting state)와 실행상태(Running stat) 간의 스케줄링 오버헤드가 발생한다. 트리거 계층에서

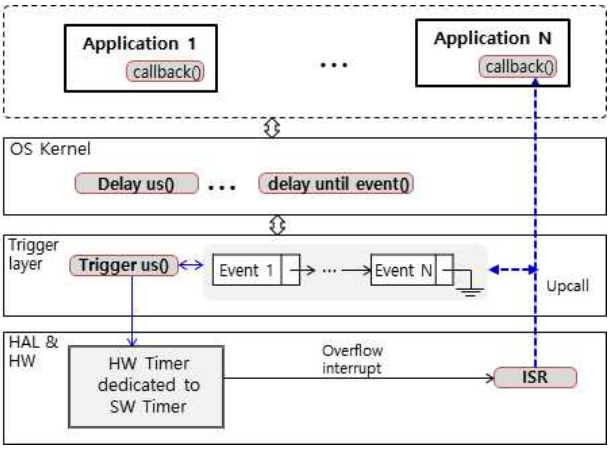


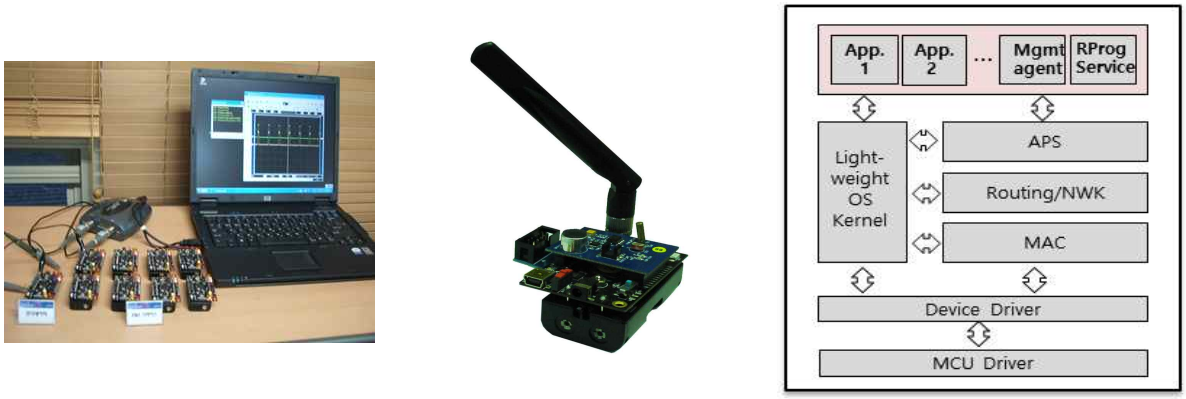
Fig. 1 Software Timer Architecture

제기되는 이슈는 이벤트 리스트를 위한 저장구조와 이벤트 트리거를 위한 타이머 관리이다. 멀티프로그래밍 환경에서는 다수의 애플리케이션과 네트워크 스택의 구성 요소에 대한 태스크가 동시에 구동할 수 있다. 따라서 다수의 이벤트가 상위 계층에서 트리거 계층으로 비동기적으로 요청될 수 있다. 이런 상황에서는 이벤트들을 저장하기 위한 효율적인 저장 구조와 처리 방식이 요구된다. 본 구현에서 이벤트 리스트를 위한 저장구조로는 연결리스트를 사용하며, 이벤트는 트리거 시간 기준으로 올림차 순으로 정렬해 연결리스트에 저장한다. 따라서 가장 빨리 처리될 이벤트가 맨 앞에 놓이며, 가장 늦게 처리될 이벤트가 맨 뒤에 놓이게 된다.

한편, 이벤트 트리거를 위한 타이머 관리도 중요한 이슈로 제기된다. 앞에서 이벤트 리스트는 트리거 시간 차례로 저장된다는 점을 살펴봤다. 이벤트 트리거 대상 이벤트를 찾는 방식은 크게 두 가지인데, 하나는 폴링(Pooling) 방식이며 다른 하나는 합핑(Hopping) 방식이다.

2.1 폴링 방식

지정된 주기로 이벤트 리스트의 첫 번째 항목에 대한 트리거 시간이 도래했는지 여부를 검사하는 방식이다. 이를 위해서는 절대 시간이 필요한데, 보통 지정된 시간 주기의 타이머를 상시 동작시켜 타이머 오버플로우 인터럽트가 발생할 때마다 절대



(a) Experimental setting (b) SN-100 sensor node (c) Embedded node software

Fig. 2 Experimental Environment

시간 용도로 사용하는 변수의 값을 증가시키는 방식을 사용한다. 타이머 인터럽트가 발생할 때마다, 즉 각 시간마다, 이벤트 리스트의 첫 번째 항목의 트리거 시간이 도래했는지 여부를 검사한다. 이 방식에서 이벤트 리스트 내의 각 이벤트의 트리거 시간은 절대 시간으로 저장된다.

2.2 합핑 방식

이벤트 리스트 내의 이벤트에 대해 차례로 타이머를 설정해 이벤트를 처리하는 방식이다. 즉, 이벤트 리스트의 첫 번째 항목에 대해 지연시간만큼 타이머를 설정한 후 인터럽트가 발생하면 해당 이벤트를 처리하며, 그 다음 이벤트에 대해서도 앞의 방법을 반복한다. 이 경우에는 폴링 방식과는 달리 이벤트 리스트에 각 이벤트의 트리거 시간이 절대 시간이 아니라 이전 이벤트 시점 대비 상대시간으로 저장된다. 즉 이전 이벤트 시점이 200이고 현 이벤트 시점이 300이라면 저장되는 트리거 시간은 100이다. 이는 이전 시점보다 100만큼 뒤에 이벤트가 발생한다는 것을 나타낸다.

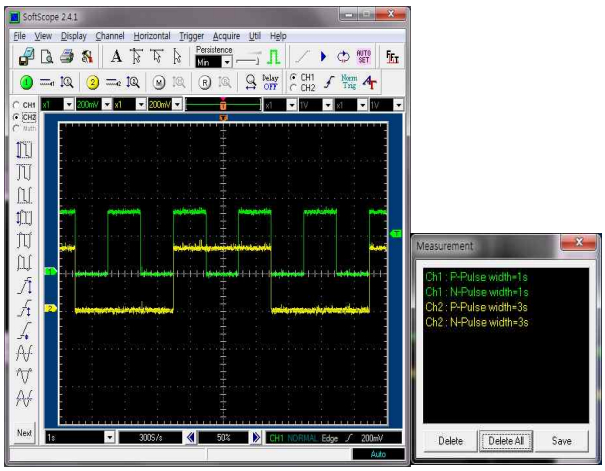


Fig. 3 Software Scope Test Example

주기적 폴링 방식과 합핑 방식은 각자 고유한 장점과 단점을 가진다. 폴링 방식은 구현이 단순하지만, 주기적인 타이머 인터럽트로 CPU 컴퓨팅 자원이 낭비되며, 아울러 인터럽트 처리에 소요되는 오버헤드의 누적 현상이 소프트웨어 타이머 오차로 이어지는 단점이 있다. 한편, 합핑 방식은 이벤트 당 1회의 타이머 인터럽트가 발생하므로 폴링 방식에서 존재하는 다수 인터럽트 누적현상은 피할 수 있으나, 다수의 이벤트가 동시에 존재할 때 이벤트 리스트가 커지며 그에 따른 오버헤드도 오차발생의 요인으로 작용한다. 이러한 단점에도 불구하고 경량 임베디드 디바이스를 위한 소프트웨어 타이머의 구현에는 합핑 방식이 상대적으로 훨씬 오차가 적은 것으로 나타나고 있어 본 연구의 실험 대상 소프트웨어 타이머도 합핑 방식을 채택하고 있다.

3. 경량 환경의 타이머 보정 기법

Fig. 2에서 볼 수 있듯이 실험환경은 SN-100 센서 노드와 내장 소프트웨어, 소프트웨어 오실로스코프로

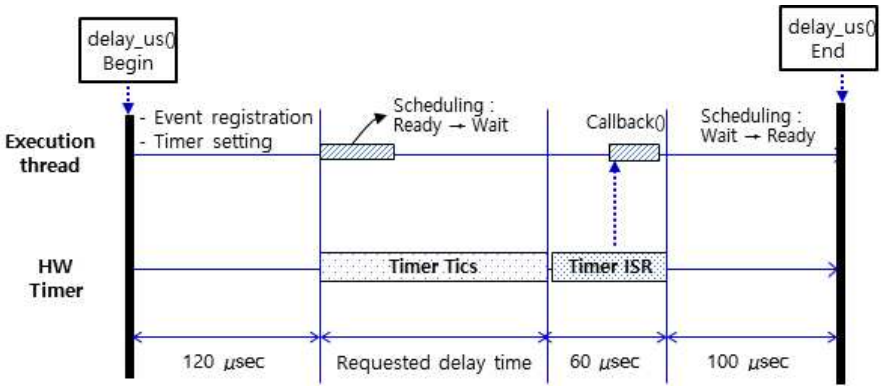


Fig. 4 Execution of delay_us() and Measured Overhead

구성되어 있다. SN-100 센서노드는 ATmega128과 CC2420을 장착하고 있으며, 이 센서노드에는 본 연구실에서 개발한 자체 경량 멀티스레드 OS 커널과 802.15.4 MAC을 포함한 ZWeaver 스택이 탑재되어 있다. ZWeaver는 ZigBee 사양을 완벽하게 구현해 ZigBee 얼라이언스 국제공인인증(인증번호 ZIG12021ZCP2682 3 -24)을 취득한 버전이다. 경량 임베디드 OS 커널은 소프트웨어 타이머 서비스를 제공하고 있으며, MAC, NWK, APS 등 ZWeaver의 모든 구성요소는 이 소프트웨어 타이머 서비스를 사용하고 있다.

소프트웨어 타이머의 정확도는 이벤트와 경과시간을 입력으로 받는 시간 관련 API 함수가 실제 경과시간이 지난 후에 해당 이벤트를 수행하는지를 측정한다. 경량 환경은 일반 범용 컴퓨터 환경에서와 같은 정확한 글로벌 타이머가 존재하지 않아서 시간측정을 위한 환경이 필수적이다. 특히 본 실험에서는 매우 높은 정확성이 요구되므로 외부 측정기기를 활용할 수 있는 환경을 구축했다. 일단 소스코드의 측정구간에서 MCU의 PORTB의 6번과 7번 핀에 신호(High 또는 Low)를 출력하는 코드를 추가했다. 이 두 핀은 연장선을 통해 소프트웨어 오실로스코프 채널 1과 채널 2에 각각 연결하고 오실로스코프의 p-pulse 폭과 n-pulse 폭을 확인할 수 있도록 구성하였다. p-pulse와 n-pulse의 폭을 시간으로 환산하면 구간의 시간간격을 얻을 수 있다. Fig. 3에서는 채널 1에 채널 2에 각각 1 us과 3us를 측정하는 작업을 예시로 보여준다.

3.1 오버헤드 분석

이 실험에서는 다양한 소프트웨어 타이머 관련 API 함수를 조사했는데 여기서는 지면을 고려해 대표적인 함수 하나를 집중적으로 상세히 설명한다. 여기서 다룰 함수는 delay_us() 함수로서 기본적으로 함수의 입력 파라미터는 μsec 단위의 지연시간과, 콜백(Callback) 함수 포인터를 갖는다. Fig 4는 delay_us() 함수의 주요 세부 실행 과정과 세부 과정별 실행 시간을 보여준다. 먼저 세부 실행 과정을 간략히 살펴보면 다음과 같다.

1) 트리거 설정 단계

애플리케이션에서 지정된 시간대기 후에 특정 동작을 취할 필요가 있을 때, 대기시간량과 해당 동작에 대한 콜백(Callback) 함수 주소를 입력 파라미터로 해서 delay_us() 함수를 호출한다. 이 함수의 내부에서는 상호배제(Mutual exclusion)를 설정한 후 바로 trigger() 함수를 호출한다. trigger() 함수는 요청된 이벤트를 이벤트 리스트에 등록한 후 이벤트 리스트의 맨 앞 항목의 대기시간에 맞춰 타이머를 작동시킨다. 참고로, 위 그림은 이벤트 리스트에 비어있어 요청된 이벤트에 대한 타이머 설정 및 구동이 바로 진행되어 트리거가 발생하는 시나리오를 나타낸다.

2) 타이머 카운팅 단계

타이머를 작동시키면 타이머는 카운팅 단계에 들어선다. 한편, trigger() 함수는 타이머를 작동시킨 후에 바로 끝나고, 실행 문맥은 delay_us() 함수로 복귀한다.. 이어진 delay_us() 함수의 문

맥에서는 스케줄러 함수를 작동시켜 현 애플리케이션 스레드를 실행상태(Running state)에서 대기모드(Wait state)로 전환한다.

3) 이벤트 처리 단계

한편 하드웨어 타이머의 시간만기(Timer expiration)가 도래하면, 오버플로우(Overflow) 인터럽트가 발생한다. 인터럽트 서비스 루틴(Interrupt service routine)은 기본적으로 이벤트 리스트에서 첫 번째 항목을 제거하고, 이 항목에 기재된 콜백함수를 업콜(Upcall)을 통해 실행해서 해당 이벤트를 처리한다.

4) 애플리케이션 문맥 진입 단계

이벤트 처리 후에는 바로 스케줄러를 동작시켜 현재 대기상태에 있는 현 애플리케이션 스레드를 준비상태로 전환한다. 준비 큐에서 저장되어 있던 스레드가 스케줄러에 의해 실행상태로 변경이 되면, 대기모드 전환시 스케줄러에 의해 중단된 delay_us() 함수가 계속 실행되지만 바로 종료하며, 곧이어 delay_us함수를 호출한 애플리케이션 문맥으로 돌아온다.

위 4개의 구간 중에서 타이머 카운팅 단계를 제외하면 나머지는 오버헤드로서 소프트웨어 타이머의 정확도를 떨어뜨리는 요소로 작용한다. 첫 번째 실험에서는 각 구간별 실행시간을 측정함으로써 소프트웨어 타이머의 기본 오차를 파악했다. 이 실험에서는 500 μsec(microseconds)의 실행지연 기능을 위해 delay_us() 함수를 사용했다. Fig. 4에서 보는 바와 같이 각 단계의 실행시간 측정치를 볼 수 있으며, 아래에 간략히 그 내용을 요약했다.

- 위 그림에서 delay_us() 함수 시점부터 타이머 작동 시작 시점까지의 구간인 트리거 설정단계는 약 120 μsec 정도의 시간이 걸리는 것으로 나타났다. 그 뒤를 따르는 타이머 카운팅 단계의 소요시간은 단순히 타이머 카운팅 시간이므로 당연히 delay_us() 함수에 입력 파라미터에 해당하는 시간지연 값과 동일하다. 참고로 이 단계에서 발생하는 애플리케이션의 상태 변경, 즉 실행모드에서 대기모드로의 변경 시간은 타이머 카운팅 시간과 중첩되므로 따로 오버헤드로 작용하지 않는다.
- 타이머 카운팅 단계를 마치면 바로 인터럽트가 발생해서 콜백함수를 처리하는 단계인 이벤트 처리 단계가 진행되는데, 이 단계는 기본적으로 약 60 μsec의 시간이 소요되는 것으로 나타났다. 여기서 콜백 함수의 실행 시간은 애플리케이션마다 다를 수 있으며 그 실행시간이 delay_us()의 일반 공통 오차에 영향을 미치지 않도록 하기 위해 로직 코드 없는 빈 내용의 콜백 함수를 사용했다.
- 마지막으로 콜백함수를 수행한 후 애플리케이션 문맥으로 돌아오는 애플리케이션 문맥 진입 단계는 100 μsec 정도의 시간이 걸리는 것으로 나타났다. 물론 애플리케이션 진입 문맥은 멀티프로그래밍 정도(Degree of multiprogramming)에 영향을 받는다. 여기서는 기본 공통 오차 산출에 주안점을 두었으므로 그러한 멀티프로그래밍 수준이 높은 혼잡 상황은 배제했다.

3.2 오버헤드 보정 기법 및 적용 결과

이어진 실험에서는 다양한 지연시간 값에 대해 delay_us() 함수의 오차를 오리지널 소프트웨어 타이머와 오베헤드 보정 소프트웨어 타이머 각각에 대해 측정했다. 이 실험에서는 delay_us() 함수의 호출 시점부터 delay_us()가 종료되어 응용 애플리케이션 문맥으로 돌아올 때까지 전체 구간을 경과시간을 측정했다. 지연요청값은 320 μsec 부터 5,000 μsec 내에서 20 μsec, 100 μsec, 1000 μsec, 3000 μsec 간격으로 모두 13개를 선택했다. Table 1은 각 지연요청값에 대해 측정한 실제 지연값(delay)을 보여준다.

위 실험을 통해 관찰한 바는 오버헤드를 보정하지 않은 오리지널 소프트웨어 타이머는 요청값의 증가에 따라 오차에 미세한 증가가 나타나지만, 전체적으로 보면 오차가 270 μsec ~ 300 μsec 범위로 비교적 균등하다는 점을 알 수 있다. 다시 말하면, 오차가 시간지연 요청값의 크기에 의존하지 않고 고정적인 크기를 갖는다. 이는 타이머 주기에 실제 요청값 대신 오차를 고려한 값을 사용함으로써 오차를 보정할 수 있다는 점을 시사한다. 오차보정은 대기시간 요청 값에서 오차 범위의 중간값인 280 μsec를 차감해서 타이머 클럭을 설정하는 방식으로 진행했다.

Table 1 Error Values of delay_us() between the Original SW Timer and the Compensation Version

Delay request (μsec)	Before Overhead Compensation		After Overhead Compensation	
	Measured value (μsec)	Error (μsec)	Measured value (μsec)	Error (μsec)
320	600	280	320	0
340	620	280	340	0
360	640	280	360 ~ 380	0' ~ 20
380	660	280	380 ~ 400	0 ~ 20
400	680 ~ 700	280 ~ 300	400	0
500	780 ~ 800	280 ~ 300	500 ~ 520	0 ~ 20
600	900	300	620	20
700	1000	300	720	20
800	1100	300	820	20
900	1200	300	920	20
1000	1300	300	1000 ~ 1020	0 ~ 20
2000	2300	300	2020	20
5000	5280 ~ 5300	280 ~ 300	5020	20

3.1절의 실험에서 오버헤드는 트리거 설정 단계, 이벤트 처리 단계, 애플리케이션 문맥 진입 단계에서 발생하며, 각각 100 μsec, 60 μsec, 100 μsec 정도가 된다는 점을 보았다. 여기서 트리거 설정단계와 이벤트 처리 단계는 trigger() 함수와 주로 관련이 있고, 애플리케이션 문맥 진입 단계는 delay() 함수와 관련이 있다. 이런 점을 고려해서 대기시간 차감은 두 곳에서 나눠 시행했다. 첫 번째는 trigger() 함수의 입력 파라미터로 제공되는 지연시간값에서 트리거 설정 오버헤드(100 μsec)와 이벤트 처리의 오버헤드(60 μsec)를 합한 180 μsec를 차감했다. 두 번째는 delay_us() 함수의 입력 파라미터로 제공되는 지연시간값에서 애플리케이션 문맥 진입 오버헤드, 즉 100μsec를 차감했다. 참고로 전체 오버헤드를 두 곳으로 나눠서 보정하는 이유는 delay_us() 외에도 커널 계층의 다른 시간 관련 API가 하위 트리거 계층의 trigger_us() 함수를 사용하기 때문이다. Table 1에서 보듯이 오버헤드 보정 후 오차는 거의 사라졌다는 것을 알 수 있다.

4. IEEE 802.15.4 MAC 적용 실험

본 절에서는 앞 절에서 제시한 소프트웨어 타이머의 시간보정이 소프트웨어 타이머를 사용하는 소프트웨어 구성요소의 성능에 미치는 영향을 평가한다. 다양한 실험 시나리오 가능하지만, 본 실험에서는 소프트웨어 타이머의 성능에 매우 민감한 무선네트워크의 기저 계층인 IEEE 802.15.4 MAC 비콘 모드에 대해 평가 대상으로 선택했다. 기존 소프트웨어 타이머 버전과 오버헤드 보정 버전 각각 탑재한 버전에 대해 IEEE 802.15.4 MAC 비콘 크기를 변화를 비교함으로써 타이머의 시간보정의 효과를 점검했다.

IEEE 802.15.4는 프레임 전송 동작과 관련하여 비컨 모드와 논비컨 모드를 둘 다 지원한다[13-14]. 비컨 모드는 비컨 메시지를 사용해서 메시지 송수신이 상호 시간 동기화 환경에서 동작할 수 있도록 한다. 비컨 모드에서는 비컨과 비컨 사이규격은 슈퍼프레임으로 명시된다. Fig. 1에서 볼 수 있듯이 슈퍼프레임은 일종의 듀티 사이클(Duty cycle)이며, 활성화(Active) 구간과 비활성화(In-Active) 구간으로 구성된다. 세부적으로 살펴보면, 활성화 구간은 전체 16개의 슬롯으로 나뉜다. 비컨 전송의 간격은 BO(Beacon Order)으로 설정되며, 슈퍼프레임 내의 활성화 구간의 크기는 SD(Superframe duration) 값으로 설정된다. BO의 값은 0부터 14까지 가능하며, 개별 값에 대한 비컨 간격은 이론적 수치는 Table 2의 두 번째 열에서 볼 수 있다.

일반적으로 IEEE 802.15.4 MAC에서 비컨 전

송 주기에 따른 실제 비콘 전송은 타이머를 사용해 구현된다. 본 실험에서는 두개의 IEEE 802.15.4 MAC 버전을 마련했다. 첫 번째는 OS의 일반 소프트웨어 타이머를 사용한 기존 버전이며, 두 번째는 오버헤드 보정 소프트웨어 타이머를 사용한 버전이다.

실험에는 코디네이터 노드 하나를 사용하며, 단순히 소프트웨어 타이머 오버헤드 영향을 최대한 정확하게 측정하기 위해 코디네이터 노드는 여타 메시지를 전송하지 않고 단순히 비컨 전송 기능만을 수행하도록 했다. 비컨 주기와 슈퍼프레임의 주기는 IEEE 802.15.4의 비컨 발생 프리머티브(Primitive)인 MLME_start_req를 사용했다. MLME_start_req의 입력 인수로 제공된 BO, SO의 값에 따라 비컨주기와 슈퍼프레임의 활성/비활성 구간의 크기가 결정된다. 설정가능한 모든 BO 값에 대해 비컨 주기를 측정하였다. SO 값은 BO 값과 동일한 값으로 설정해서 슈퍼프레임의 크기가 비콘 주기와 동일하게 하여 비활성화 구간이 존재하지 않도록 하였다.

오리지널 소프트웨어 타이머 기반 버전과 오버헤드 보정 소프트웨어 타이머 기반 버전의 비콘 인간격의 측정결과를 Table 1에서 나타냈다. 오리지널 소프트웨어 타이머 기반 버전의 경우 BO가 0에서 4까지는 3000 μsec를 약간 웃도는 정도의 오차를 보이다가 BO의 범위가 5부터 14까지는 대략 2배 정도 증가하는 것을 볼 수 있다. 이런 상황은 아래와 같이 두 가지로 해석될 수 있다.

첫 번째는 오리지널 소프트웨어 타이머의 오차에 작용한 오버헤드와 관련한 사항이다. 현재 가동하는 경량 OS의 스케줄러는 라운드로빈 방식의 스케줄링을 채택하고 있으며 디폴트 타임퀀텀은 250 msec로 설정되어 있다. 스케줄러도 소프트타이머를 사용하고 있어 비콘 간격이 커지면 스케줄링을 위한 인터럽트들에 대한 오류가 누적된다. BO가 14인 경우를 사례를 들어보자. 이 경우 비콘 간격의 이론치가 251.65824 sec이므로 비콘 간격 내에 1,006회 가량의 스케줄러용 소프트웨어 타이머 인터럽트가 처리된다. 소프트웨어 타이머 오버헤드 280 μsec를 감안하면 전체 오차는 281,680 μsec로 실험을 통해 측정한 값 264,760 μsec과 유사한 것을 알 수 있다. 전제적으로 보면, 이런 계산값과 측정치가 맥락적으로 일치하는 현상이 BO가 5에서 14까지는 유지되고 있다.

Table 2 Measurements of Beacon Interval Deviation

BO	Theoretic value (sec)	Error (μsec)		(B/A)*100(%)
		Original SW Timer (A)	Compensated SW Timer (B)	
14	251.65824	264760	5200	2.0
13	125.82912	134880	4980	3.7
12	62.91456	39440	4820	12.2
11	31.45728	36720	4400	12.0
10	15.72864	19360	4200	21.7
9	7.864	12000	3640	30.3
8	3.93216	7840	3700	47.2
7	1.96608	4920	3520	71.5
6	0.98304	3960	3400	85.9
5	0.49152	3780	3320	95.4
4	0.24576	3240	3040	93.8
3	0.12288	3120	3100	99.4
2	0.06144	3360	3080	86.5
1	0.03072	3280	3060	93.3
0	0.01536	3340	2920	80.2

두 번째는 비콘 전송과 관련한 작업부하이다. 전술한 바와 같이, 오리지널 소프트웨어 타이머 기반의 비콘 간격이 오차는 BO가 0에서 4까지는 3000 μsec를 웃도는 정도의 오차를 보이다가 점차로 증가하는 것으로 나타났다. 사실 이런 BO값에서는 비콘 간격이 15.36 msec에서 245.76 msec까지로 스케줄러의 인터럽트 주기보다 작으므로 스케줄링을 위한 소프트웨어 타이머 오차 누적의 영향을 받지 않는다. 따라서 단순히 비콘 전송과 관련한 작업부하가 비콘 간격의 오차를 구성한다. 비컨 전송관련 부하는 비컨의 전송에 소요되는 오버헤드로서 일반적으로 수십에서 수백 심볼이 걸리는 것으로 나타나고 있다. 비컨의 전송 오버헤드는 소프트웨어 타이머 오버헤드부터 비컨 생성, 비컨 전송 및 ACK 수신까지 다양한 요소로 작용하는데, 예를 들어 데이터 전송과 관련해 CSMA/CA 백오프(Backoff) 단위는 20 심볼로 1 심볼이 16 μsec이므로 약 320 usec가 된다.

한편, 소프트웨어 타이머 오버헤드를 보정한 버전은 기본적으로 오버헤드로 인한 오차 자체가 거

의 제거된 상태이므로 BO값 전반에 거쳐 누적효과가 거의 나타나지 않지 않는다는 점을 볼 수 있다. 물론, 누적 효과와 자신의 비컨 발생을 위한 오버헤드 자체가 제거됐지만, 비컨 전송과 관련한 순수 오버헤드 그대로 존재하므로 BO가 0에서 4까지의 오차는 약간 작아진 것을 알 수 있다.

제안하는 소프트웨어 타이머 보정기법이 실환경에서도 유효할 지를 검증하기 위해서는 외부 인터럽트의 영향도 평가할 필요가 있다. 데이터 패킷이 노드에 수신되면, PHY 계층의 RADIO 칩은 FIFOP 인터럽트를 발생시킨다. 한편, 수신을 완료한 수신노드는 송신노드에 ACK 전송하며, 이에 따른 FIFOP 인터럽트가 발생된다. 이러한 FIFOP 인터럽트는 외부 인터럽트로서 내부 인터럽트인 타이머 인터럽트보다 높은 우선순위를 갖는다. 이 실험에서는 패킷 송수신을 통해 외부 인터럽트를 발생시키고 이런 외부 인터럽트가 소프트웨어 타이머 보정기법에 미치는 영향을 평가한다. 이 실험에서는 패킷 송수신의 횟수를 최대화하여 외부 인터럽트의 발생량을 최대화하고 기존 소프트웨어 타이머 버전과 제안하는 보정기법이 적용된 버전 각각에 대해 특정 크기의 패킷에 대한 최대전송율과 최대수신율을 측정해 비교한다.

먼저, 최대전송율의 실험 내용을 살펴본다. 최대 전송률은 한 개의 송신노드가 단위시간에 전송할 수 있는 최대 패킷 개수에 의해 결정되며, 그 값은 네트워킹 스택의 패킷 전송 경로의 오버헤드에 의해 결정되므로 이웃 노드의 개수와는 무관하다. 이에 따라 실험환경은 송신노드 한 개와 수신노드 한 개로 구성했다. 송신노드 상에는 동작하는 한 개의 응용프로그램이 50 바이트 크기의 데이터를 전송지연 간격 없이 수신노드로 10,000회 전송하는 데 걸리는 시간을 측정했으며, 이 실험을 10회 반복해서 평균 전송량을 계산했다. 전송량을 늘리기 위해 네트워크의 응용 계층인 APS(Application service) 계층은 NO_ACK 모드로 설정했다.

한편, 최대수신율의 실험 내용은 다음과 같다.

일반적으로 네트워킹 스택에서 패킷 수신 오버헤드는 패킷 전송 오버헤드보다 작다. 주된 이유는 패킷 수신의 경우에는 상위 계층으로의 패킷 흐름만 존재하지만, 패킷 전송의 경우에는 패킷 하위 계층으로의 패킷 흐름뿐만 아니라 상위 계층으로의 전송확인 신호(Indication) 흐름이 존재하며, 아울러 전송과정에서는 CSMA 처리에도 상당한 지연이 추가되기 때문이다. 이에 따라 노드의 최대수신량을 측정하기 위해서는 수신노드에 패킷을 전송하는 노드의 개수는 다수이어야 한다. 실험에서는 한 개의 코디네이터와 5개의 엔드노드로 구성했는데 코디네이터를 중심으로 엔드노드들을 스타형 토폴로지 형태로 배치하였다. 5개의 엔드노드는 패킷을 수신노드인 중앙의 코디네이터로 전송하는 역할을 한다. 전송노드의 패킷 크기, 전송방식, 측정방법 및 설정 내용은 이전 최대전송 환경의 실험과 동일하다.

Table 3은 위 두 실험의 결과를 보여준다. 보정 이전 소프트웨어 타이머 버전과 보정기법 적용 버전 각각에 대한 10,000개의 패킷의 전송 또는 수신 소요시간이 나타나있으며, 이 소요시간을 바탕으로 계산한 APP(Application layer), MAC(Medium access layer), PHY(Physical layer) 계층의 전송률도 맨 오른쪽 열에서 볼 수 있다. 한편, 세 번째 열에는 계산된 전송률의 이해를 돕기 위해 각 계층별로 부착되는 헤더의 크기가 반영된 Payload 크기를 나타냈다.

이 두 실험의 결과를 보면, 제안하는 보정기법을 적용한 SW타이머 기반의 환경이 최대전송률과 최대수신율 둘 다에 대해 원래 버전보다 우위를 보이고 있음을 볼 수 있다. 구체적으로 보면, 제안 기법을 적용한 소프트웨어 타이머 버전이 전송률의 경우에는 10.6%, 수신율은 11.1%로 우수한 것으로 나타났다. 일반적으로 외부 인터럽트 인한 소프트웨어 타이머 인터럽트 유실이나 소프트웨어 타이머 내부의 이벤트 관리 오버헤드 등으로 유발되는 타이머 자체의 부정확성은 비컨 동기화 저해 등으로 패킷 송수신율을 떨어뜨리는 요인으로 작용한다. 따라서 이 실험을 통해 관찰한 송수신율의 향상은 제안한 보정기법이 외부 인터럽트가 발생하는 실환경에서도 잘 작동되어 상대적으로 높은 소프트웨어 타이머 정확성을 보장한다는 점을 시사한다.

본 절을 마치기 전에 한 가지 지적할 사항이 있다. 소프트웨어 타이머의 오류 보정은 소프트웨어 타이머 구현에 있어서는 반드시 필요한 기

Table 3 Measurement of Packet Transmission and Reception Rate

	SW Timer version	Configuration parameters				Total time (ms)	Transmission rate(bps) (Based on Payload length)		
		Payload			Transmissions				
		APP	MAC	PHY			APP	MAC	PHY
1:1 Maximal Transmission	Original SW Timer	50	66	77	10,000	297,359	13,452	17,756	20,716
	Compensated SW Timer	50	66	77	1,0000	265,708	15,054	19,871	23,183
	Transmission Enhancement Ratio					10.6%			
N:1 Maximal Reception	Original SW Timer	50	66	77	10,000	151,827	26,346	34,776	40,572
	Compensated SW Timer	50	66	77	1,0000	135,028	29,623	39,103	45,620
	Reception Enhancement Ratio					11.1%			

술이지만, 소프트웨어 타이머의 오류 발생과 관련한 모든 문제를 해결해 주진 못한다는 점이다. 이에 따라, 본 연구에서 주안점을 둔 특정 고정값을 보정한 정적 보정 기법보다는 상황에 따라 적절한 보정값을 사용하는 동적 보정 기법도 고려해 볼 수 있다.

5. 결 론

경량 MCU를 장착한 임베디드 디바이스 상에서는 하드웨어 타이머의 수적 제약으로 애플리케이션이나 네트워킹 스택 등의 구현에 필요한 타이머 요구를 내장 운영체제의 소프트웨어 타이머에 의존하는 경향이 높다. 이러한 경우에 타이머의 오차가 시분할 동기적 통신의 신뢰성을 상당히 훼손할 수 있다. 이러한 타이머 오차는 소프트웨어 타이머 관리, 타이머 이벤트 처리를 위한 맥락 전환 등 다양한 요인에 의해 발생한다. 이 논문에서는 소프트웨어 타이머 오차 발생 요인을 분석한 후 실험을 통해 소프트웨어 타이머 관리 오버헤드와 이벤트 처리를 위한 맥락전환 등의 오버헤드는 타이머 요청시간과는 무관하게 고정적인 크기를 가진다는 점을 보였다. 이를 활용한 한 소프트웨어 타이머 오차보정 기법은 소프트웨어 타이머의 정확도를 개선할 수 있다는 점도 실험을 통해 확인하였다. 응용 실험에서 통해 이러한 정확도 개선은 소프트웨어 타이머를 기반한 경량 네트워킹 스택 성능도 향상시킬 수 있다는 점을 확인했다.

References

[1] M Sethi, P. and Sarangi, S., “Internet of Things: Architectures, Protocols, and Applications,” Journal of Electrical and Computer Engineering, Article ID 9324035, 2017.

[2] Mounica1, A. and Subbareddy, G., "Zigbee transmitter for IOT Wireless Devices", International Journal of VLSI Design & Communication Systems (VLSICS), Vol. 8, No. 5, pp. 1-13, 2017.

[3] Kim, H., Hong, W., Yoo, J. and Yoo, S., “Experimental Research Testbeds for Large-Scale WSNs: A Survey from the Architectural Perspective,” International Journal of Distributed Sensor Networks, Vol. 11, No. 3, Article No. 2, 2015.

[4] Dong, W., Chen, C., Liu, X., Liu, Y., Bu, J. and Zheng, K., "SenSpire OS: A Predictable, Flexible, and Efficient OS for Wireless Sensor Networks," IEEE Transactions on Computers, Vol. 60, No. 12, pp. 1788-1801, 2011.

[5] Aron, M. and Druschel, P., "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing", ACM Tractions on Computer Systems, Vol. 18, No. 3, pp. 197-228, 2000.
[6] Eswaran, A., Rowe, A. and Rajkumar, R., "Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks," Proceedings of IEEE Real-Time Systems Symposium, pp. 1-10, 2005.
[7] Al-Suhail, G., Jero, M. and Nikolakopoulos, G., "A Practical Survey on Wireless Sensor Network Platforms," Journal of Communications Technology, Electronics and Computer Science, Issue 13, pp. 23-29, 2017.
[8] Kim, H. and Yoo, S., "Implementation and Analysis of IEEE 802.15.4 Compliant Software based on a Vertically Decomposed Task Model," Journal of the Korea Industrial Information Systems Research, Vol. 19, No 1, pp. 53-60, 2014.
[9] Kim, T. and Ahn, K. "Mitigating Hidden Nodes Collision and Performance Enhancement in IEEE 802.15.4 Wireless Sensor Networks," Journal of Korea Information Processing Society, Vol. 4, No. 7, pp. 235-238, 2015.
[10] Stanislowski, D., Vilajosana, X., Wang, Q., Watteyne, T. and Pister., K., "Adaptive Synchronization in IEEE802.15.4e networks," IEEE Transactions on Industrial Informations, Vol. 10. No. 1, pp. 795-802, 2014.
[11] Gonzalez, S., Camp, T. and Jaffres-Runser, K., "The Sticking Heartbeat Aperture Resynchronization Protocol," 26th International Conference on Computer Communication and Networks, pp. 1-8, 2017.
[12] Kim, H., "Improving the Reliability of Beacon Synchronization of IEEE 802.15.4 MAC Protocol using a Reception Time Compensation Scheme," Journal of the Korea Industrial Information Systems Research, Vol. 23, No 3, pp. 1-11, 2018.
[13] IEEE Std 802.15.4-2011: Part 15.4: Medium Access control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), IEEE Computer Society, 2011.
[14] Khoufi, I., Minet, P. and Rmili, B., "Beacon Advertising in an IEEE 802.15.4e TSCH Network for Space Launch Vehicles," 7th European Conference for Aeronautics and Aerospace Science (EUCASS), 2017.

김 희 철 (Hiecheol Kim)
• 정회원
• 연세대학교 전자공학과 학사
• 미국 남가주대(USC) 컴퓨터공학과 석사
• 미국 남가주대(USC) 컴퓨터공학과 박사
• 대구대학교 정보통신대학 정보통신공학부 교수
• 관심분야 : 임베디드 OS, 무선센서네트워크, 머신러닝, 딥러닝