임베디드 소프트웨어의 테스팅과 모니터링을 위한 RIOS 기반 어플리케이션 구조 설계

이성희°, 김덕엽, 윤보람, 이우진*
경북대학교 컴퓨터학부, *소프트웨어기술연구소
°e-mail:lee3229910@gmail.com

Architecture Design of RIOS-based Application for Testing and Monitoring Embedded Software

Sunghee Lee°, Deok Yeop Kim, Bo Ram Yun, Woo Jin Lee*
School of CSE & *SWRC, Kyungpook National University

요 약

임베디드 소프트웨어의 개발은 실제 어플리케이션이 수행되는 대상 시스템이 아닌 호스트 시스템에서 개발되기 때문에 개발 중 테스팅을 수행하기 어렵다. 또한 대상 시스템에서 어플리케이션이 실행될 때 결함 또는 오류가 발견되면 이를 재현하기 어렵다. 이러한 문제를 해결하기 위한 기존의 연구로는 RTOS 시뮬레이터를 사용하거나 모니터링 시스템을 추가하여 임베디드 소프트웨어의 동작을 확인한다. 하지만 RTOS 시뮬레이터는 기능 테스트만 가능하고 실직적인 시간 추정이 불가능하다. 또한 임베디드 소프트웨어에 모니터링 시스템을 추가하게 되면 어플리케이션의 동작에 영향을 주기 때문에 실시간 시스템의 제약 조건을 확인하기 어렵다. 따라서 본 논문에서는 임베디드 소프트웨어의 RIOS 기반 어플리케이션 구조를 제안하여 호스트 시스템에서 대상 시스템의 테스팅과 모니터링이 가능함을 보인다.

1. 서론

최근 IT산업의 발전으로 사물인터넷, 자동차, 드론 등의 임베디드 소프트웨어가 사용되는 범위가 확장되고 있다. 이러한 임베디드 소프트웨어의 개발은 많은 경우에 호스트 환경인 PC에서 개발된다. 하지만 실제 소프트웨어가 탑재되고 동작하는 대상 시스템은 PC가 아닌 경우가 대부분이다[1]. 이러한 두 시스템 환경의 차이 때문에 많은 개발자들이 임베디드 소프트웨어의 테스팅에 많은 어려움을 겪는다.

먼저, 임베디드 소프트웨어의 테스팅이 어려운 이유는 대상 시스템이 존재할 때는 로그기반의 테스팅[2]을 수행할 수 있지만 로깅 모듈이 존재하지 않으면 테스팅을 수행하기 어렵다. 반면에 대상 시스템이 존재하지 않을 때는 주로 RTOS 시뮬레이션으로 테스팅이 수행된다. 하지만 RTOS에서 제공하는 시뮬레이터는 대부분 기능적인 부분만 호스트에서 동작이 가능하기 때문에 대상 시스템에서 실제 실행될 때의 시간 등의 제약 조건을 확인하기 어렵다[3].

그리고 임베디드 소프트웨어의 모니터링은 모니터링의 특성상 어플리케이션이 수행될 때 특정 시점 또는 특정 지점에 개발자가 확인하고 싶은 동작 또는 변수 값을 확인해야하기 때문에 어플리케이션 코드가 수행되는 중간에 모니터링 코드가 수행된 후 다시 어플리케이션 코드를 수행해야한다. 이러한 일련의 과정을 반복하는 동안에 해당 시스템과 상호작용하는 외부 시스템과 환경의 상태를 변화할 수 있기 때문에 모니터링 시스템으로 확인할 수 있는 것에 한계가 있다.

따라서 본 논문에서는 이러한 문제를 해결하기 위해 스케쥴링 중심의 RIOS 기반의 어플리케이션 구조를 사용한다. 테스팅 태스크와 모니터링 태스크를 생성하고 테스팅 파라미터와 모니터링 파라미터를 등록한 뒤 RIOS 스케쥴러를 수정하여 순수 어플리케이션 태스크들이 먼저 수행되게 한다. 그 이후에 테스팅 태스크와 모니터링 태스크를 수행되게 하여 호스트 시스템에서 대상 시스템의 테스팅과 모니터링을 할 수 있다.

2. 관련연구

2.1 RTOS 환경에서의 임베디드 소프트웨어 테스팅

임베디드 소프트웨어의 테스팅은 주로 RTOS 시뮬레이터를 사용하거나 대상 시스템에 적합한 형태의 테스트 프레임워크 또는 도구를 개발하여 수행한다[2-7]. RTOS 시뮬레이터를 사용한 테스트는 호스트에서 개발 중에 어플리케이션의 테스트가 가능하지만 타겟에서 실행될 때 실제 수행되는 시간측정은 불가능하다는 단점이 있다[3].

대상 시스템에 적합한 형태의 테스트 프레임워크를 개발하여 테스트를 진행하는 경우는 호스트에서 테스트에 필요한 정보를 전달하여 대상 시스템의 동작을 확인한다. 하지만 대상 시스템에 테스트를 진행하기 위해서 필요한 로깅 코드를 삽입하거나[2] 테스트 드라이버 등의 테스트

수행에 필요한 코드를 삽입하여[4-6] 테스트를 수행한다. 하지만 이러한 테스트 방법은 추가로 삽입된 코드로 인해 어플리케이션의 동작에 영향을 줄 수 있다. 따라서 시간과 자원의 제약이 있는 RTOS 환경에서는 테스트 결과가 정확하지 않을 수 있다.

2.2 임베디드 소프트웨어의 모니터링

임베디드 소프트웨어를 개발하고 테스트할 때 어려운 점 중 하나는 외부로부터의 입력이 정확하게 들어가는지 혹은 소프트웨어 동작 중에 주요한 값들이 정확한지 확인하는 것이다. 이러한 것들을 확인하기 위해서는 임베디드 시스템 내부 또는 외부에 모니터링 시스템이 존재해야 한다. 하지만 이러한 모니터링 시스템은 그 자체로 임베디드 기기의 자원을 사용하기 때문에 일반적으로 임베디드 시스템에 없는 경우가 많다.

임베디드 소프트웨어의 모니터링을 수행하기 위해서는 내부적으로 모니터링 기능을 제공하지 않는다면 추가적으로 모니터링 시스템을 개발하여야 한다. 하지만 이러한 모니터링 시스템이 디버거를 사용하거나 매 함수 호출마다 핸들러를 사용하는 등의 방법으로 동작하기 때문에 실제 어플리케이션의 동작에 영향을 줄 수 있다[8-9].

2.3 RIOS(Riverside/Irvine Operating System)

RIOS[10]는 UC리버사이드와 UC어바인 대학교에서 개발된 간단한 형태의 스케쥴러이다. RIOS는 C언어 기반의 멀티태스킹을 제공하며 그림 1과 같은 스택 구조를 가진다[11]. RIOS는 타이머 인터럽트를 사용하여서 인터럽트 서비스 루틴을 호출하는데 이 인터럽트 서비스 루틴에서 스케쥴링을 수행한다.

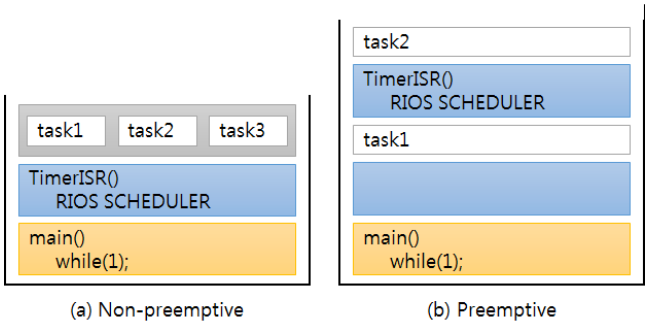


그림 1. RIOS의 스케쥴링 스택 구조

3. RIOS 기반의 임베디드 소프트웨어 구조 설계

본 논문에서는 비선점형 RIOS를 대상으로 임베디드 소프트웨어 구조를 제안하고 있다. 어플리케이션의 동작에 주는 영향을 최소화하면서 테스팅과 모니터링이 가능한 어플리케이션 구조는 그림 2와 같다. 테스팅 태스크는 어플리케이션의 테스팅에 필요한 값을 조작하고 모니터링 태스크는 테스팅과 모니터링에 필요한 값을 전달한다.

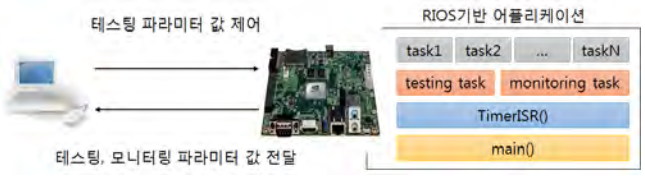


그림 2. 테스팅과 모니터링을 위한 RIOS 기반의 어플리케이션 구조

그림 3은 테스팅과 모니터링을 수행하는 어플리케이션 수행도이며 흰색으로 나타낸 부분이 순수 어플리케이션 동작에 해당하는 수행과정이고 회색으로 나타낸 부분이 테스팅과 모니터링을 위해 필요한 수행과정이다. 먼저 메인함수가 수행되면 어플리케이션에 필요한 것들을 초기화하는 과정이 수행된다. 그리고 테스팅과 모니터링 태스크를 생성하고 호스트로부터 모니터링과 테스팅에 필요한 변수를 전달받아 어플리케이션 태스크들이 수행되기 전에 미리 등록한다. 이 과정이 끝나고 어플리케이션 태스크를 생성한다. 그리고 RIOS 스케쥴러에 따라 스케쥴링된 태스크들이 수행된다. 모든 태스크들이 수행이 완료한 뒤 테스팅 태스크와 모니터링 태스크를 수행함으로써 순수 어플리케이션 태스크들에 영향을 주지 않으면서 테스팅 태스크와 모니터링 태스크를 수행할 수 있다.

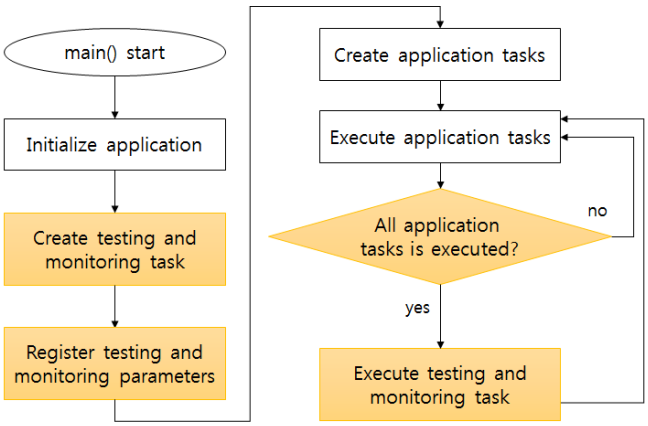


그림 3. 테스팅과 모니터링을 위한 어플리케이션 수행도

어플리케이션의 태스크들의 동작에 영향을 최소화하면서 테스팅과 모니터링하기 위해서는 RIOS 스케쥴러가 각 태스크들의 주기별 동작이 완료한 후 태스크들이 동작하지 않을 때 테스팅 태스크와 모니터링 태스크를 수행하면 된다. 이러한 스케쥴링을 하기 위해서 테스팅 태스크와 모니터링 태스크를 그림 4와 같이 정의한다. 테스팅 태스크와 모니터링 태스크가 일정한 주기마다 반복적으로 수행하게 되면 어플리케이션 동작에 영향을 줄 수 있기 때문에 태스크들의 주기를 음수로 정의하여 다른 태스크들이 동작하지 않는 시점에 태스크가 수행될 수 있도록 정의하여 스케쥴러가 태스크를 구분하여 처리할 수 있다.

테스팅 태스크와 모니터링 태스크를 포함한 모든 태스크를 스케쥴링하기 위한 스케쥴러의 핵심 코드는 그림 5

와 같다. 타이머 인터럽트를 최소한으로 하기 위해서 테스팅 태스크와 모니터링 태스크를 제외한 모든 태스크들의 주기의 최대공약수인 tasksPeriodGCD를 주기로 설정한다. 테스팅 태스크와 모니터링 태스크를 제외한 모든 태스크는 타이머 인터럽트 주기만큼 자신이 수행될 차례인지 확인하고 수행을 하게 된다. 이 과정에서 어떤 태스크도 수행되지 않는 시점이 오면 그 때 테스팅 태스크와 모니터링 태스크를 수행할 수 있어 다른 태스크들의 동작에 영향을 최소화한다.

```
#define TESTING_PERIOD    -1
#define MONITORING_PERIOD -2

typedef struct task {
    unsigned long period;         // task cycle
    unsigned long elapsedTime; // last execution time
    void (*TickFct)(void)      ;     // task function
} task;

task tasks[TASK_MAX];
taskCount = 0;

tasks[taskCount].period = TESTING_PERIOD;
tasks[taskCount].elapsedTime = 0;
tasks[taskCount].TickFct = &testingFunction;
taskCount++;

tasks[taskCount].period = MONITORING_PERIOD;
tasks[taskCount].elapsedTime = 0;
tasks[taskCount].TickFct = &monitoringFunction;
taskCount++;
```

그림 4. 테스팅 태스크와 모니터링 주기 설정 코드

```
TimerSet(tasksPeriodGCD);
TimerOn();

while(1) {
    taskExecutionCount = 0;
    for (i=2; i<TASK_MAX; i++) {
        if (tasks[i].elapsedTime >= tasks[i].period) { // ready
            tasks[i].TickFct();// task execution
            tasks[i].elapsedTime = 0;
            taskExecutionCount++;
        }
        tasks[i].elapsedTime += tasksPeriodGCD;
    }

    if(taskExecutionCount == 0)
    {// testing & monitoring task execution
        for(i=0; i<2; i++)
            task[i].TickFct();
    }

    TimerFlag = 0;
    while (!TimerFlag) {
        Sleep();
    }
}
```

그림 5. RIOS 기반 스케줄러의 핵심 코드

테스팅과 모니터링에 필요한 제어 변수와 파라미터 등록 함수는 그림 6와 같이 정의하여 사용할 수 있다. 제어 변수는 정수 정보를 관리할 intBuf와 실수 정보를 관리할 floatBuf를 선언하여 사용한다. 또한 임베디드 소프트웨어는 자원에 제약이 많기 때문에 최소한의 자원으로 테스팅과 모니터링을 수행하여야 한다. 따라서 테스팅과 모니터링에 필요한 정보를 관리하는 변수를 공용으로 사용한다.

```
#define INT_PARAMETER_MAX     10
#define FLOAT_PARAMETER_MAX 10

uint8_t intBufIndex = 0, floatBufIndex = 0;
int32_t *intBuf[INT_PARAMETER_MAX];
float *floatBuf[FLOAT_PARAMETER_MAX];
```

그림 6. 테스팅과 모니터링을 위한 제어 변수 선언

테스팅과 모니터링에 필요한 파라미터 등록에 관한 함수는 그림 7과 같이 정의하고 사용한다. 테스팅과 모니터링에 사용할 파라미터를 포인터로 참조함으로써 테스팅 테스크와 모니터링 테스크에서도 접근하여 값을 변경하거나 호스트로 전달할 수 있다. 이 때, 유의해야할 점은 파라미터를 관리하는 버퍼의 인덱스는 파라미터를 등록할 때마다 1씩 증가하는데 파라미터를 타입별로 관리하기 때문에 인덱스도 타입별로 증가한다.

```
int8_t registerIntParam(int32_t *param)
{
    if(intBufIndex >= INT_PARAMETER_MAX)
        return -1;

    intBuf[intBufIndex] = param;
    return intBufIndex++;
}

int8_t registerFloatParam(float *param)
{
    if(floatBufIndex >= FLOAT_PARAMETER_MAX)
        return -1;

    floatBuf[floatBufIndex] = param;
    return floatBufIndex++;
}

registerIntParam(&sensor1);
registerFloatParam(&sensor2);
```

그림 7. 테스팅과 모니터링 파라미터 등록 코드

테스팅과 모니터링할 때 호스트와 임베디드 시스템간의 통신 프로토콜은 표 1과 같이 정의한다. 이 프로토콜에서 타입은 정수형을 뜻하는 “I”와 실수형을 뜻하는 “F” 2가지만 존재하며 인덱스는 임베디드 시스템에서 등록된 버퍼에서의 인덱스이다.

표 1. 어플리케이션 수행 중 사용하는 통신 프로토콜

구분		통신 프로토콜(12byte)				
바이트 수(byte)		2	4	1	1	4
테스트용	요청	“TI”	[시각]	[타입]	[인덱스]	[값]
	응답	“TO”				
모니터용	응답	“MP”	[시각]	[타입]	[인덱스]	[값]

파라미터 등록 시 응답으로 받았던 인덱스들을 관리하여 테스트용 파라미터는 값을 제어할 수 있고 모니터용 파라미터는 값을 응답만 받을 수 있다. 모든 메시지 크기를 통일하기 위해서 총 12바이트로 구성하고 메시지 식별자인 최상위 2바이트는 아스키코드로 전송되고 이를 제외한 나머지 10바이트는 모두 바이너리로 전송된다. 테스팅과 모니터링을 할 때, 응답 메시지의 시각은 절대적인 시각이 아니라 어플리케이션에서 타이머가 수행된 횟수를

의미하며 이를 계산하여 호스트에서는 수행시간을 측정할 수 있다.

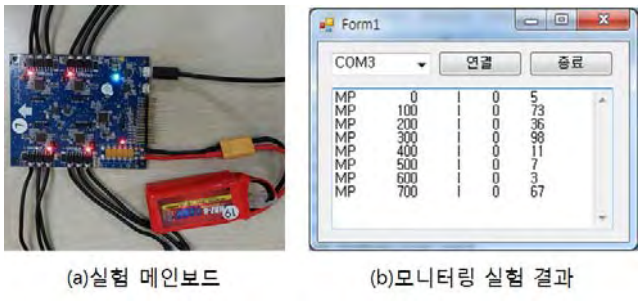
본 장에서 제안하는 RIOS 기반의 임베디드 소프트웨어 구조를 사용하면 별도의 디버거와 모니터링 시스템 없이 호스트에서 주기적으로 대상 시스템을 테스트하고 모니터링할 수 있다.

4. 적용 사례

본 논문에서 제안한 RIOS 기반의 구조를 적용한 샘플 어플리케이션을 개발하여 ARM Cortex-M3 프로세서를 사용하는 드론 메인보드인 블루비[12]에 탑재시켜 실험하였다. 실험에 사용된 프로그램 태스크는 총 4개로 표 2와 같이 구성하였다. 테스팅 태스크는 3번 태스크에 사용할 n값을 호스트로부터 입력받아 n값을 변경하고 모니터링 태스크는 센서 값을 호스트로 출력한다. 그리고 2, 3번 태스크는 자신의 태스크 주기마다 자신의 작업을 수행한다. 실험결과로 통신 프로토콜에 맞게 전송된 메시지를 확인할 수 있으며 가상화된 센서 값을 모니터할 수 있음을 그림 8에서 확인할 수 있다.

표 2. 어플리케이션의 태스크 구성과 설명

태스크 번호	태스크 주기	태스크 작업 내용
0	-1	테스팅 태스크
1	-1	모니터링 태스크
2	100	센서 값을 가상화한 랜덤 함수 수행 태스크
3	200	1부터 n가지 합을 구하는 태스크



(a)실험 메인보드 (b)모니터링 실험 결과

그림 8. 실험 메인보드와 모니터링 실험 결과

5. 결론

본 논문에서는 대상 시스템을 테스트하거나 모니터링할 때 디버거나 추가적인 모니터링 시스템의 개발 없이 순수 어플리케이션 태스크의 동작에 영향을 주지 않는 RIOS 기반의 어플리케이션 구조를 제안한다. ARM Cortex-M3 프로세서를 사용하는 메인보드에 적용하여 가상화된 센서 값을 모니터할 수 있음을 확인하였다. 향후 연구로는 본 연구를 확장하여 임베디드 소프트웨어의 실시간 테스팅 프레임워크를 개발하고자 한다.

※ 본 논문은 2014년도 정부(교육부)의 재원으로 한국과학재단의 지원을 받아 수행된 기초연구사업(No. NRF-2014R1A1A2058733)으로 수행되었음

참고문헌

[1] 이신영, 노혜성, 이성희, 이우진, “임베디드 소프트웨어의 모듈별 코드 적합성 분석 도구 개발”, 한국정보과학회 동계학술발표회 논문집, pp. 1582-1584, 2015년 12월

[2] 류호동, 정수용, 이우진, 김황수, “임베디드 소프트웨어의 단위 테스팅을 위한 로그 기반 테스팅 프레임워크 개발”, 한국정보처리학회, 한국정보처리학회논문지/소프트웨어 및 데이터 공학, Vol. 4, No. 9, pp. 419-424, 2015년 9월

[3] 김방현, 류성준, 김종현, 남영광, 이광용, “실행시간 추정 가능한 RTOS 시뮬레이터의 구현”, 한국시뮬레이션학회, 춘계학술대회논문집, pp. 125-129, 2002년 5월

[4] Louis-Marie Givel, Matthias Brun, Camille Constant, "Use of Runtime Enforcement for the Test of Real-time Systems," 2015 IEEE HPCC, CSS and ICESS, Aug. 2015

[5] 배승천, 김명진, 하대연, “DO-178B 레벨 A인증을 위한 NEOS(Nano Engine Operating System) RTOS(Real Time Operating System) 기반의 테스트 자동화 프레임웍 구현”, 대한전기학회, 정보 및 제어 심포지엄 논문집, pp. 216-217, 2014년 4월

[6] 김상일, 이남용, 류성열, “실시간 이동형 내장 소프트웨어 시험 도구의 구조 설계”, 한국정보과학회, 정보과학회논문지: 소프트웨어 및 응용, Vol. 33, No. 4, 2006년 4월

[7] I.V. Druzhinin, A.A. Mynzhasova, E.A. Sinelnikov, "Design and implementation of a hardware-software module for testing real-time systems," 2011 Proceedings of the 34th International Convention MIPRO, May. 2011

[8] 한인규, 임성수, “가상화 환경에서 임베디드 시스템을 위한 모니터링 프레임워크와 디버깅 시스템”, 한국정보과학회, 정보과학회 컴퓨팅의 실제 논문지, Vol. 21, No. 12, pp. 792-797, 2015년 12월

[9] 백장운, 이정욱, 김재영, “OSEK/VDX 운영체제 기반의 차량용 소프트웨어 개발을 위한 런타임 모니터링 시스템”, 한국자동차공학회, 학술대회 및 전시회, pp. 2109-2110, 2010년 11월

[10] RIOS, http://www.riosscheduler.org/

[11] Bailey Miller, Frank Vahid, Tony Givargis, "RIOS: A Lightweight Task Scheduler for Embedded Systems ", ACM SIGBED, Proceedings of Workshop on Embedded and Cyber-Physical Systems Education, No. 9, Oct. 2012

[12] 블루비(BLUEBEE), http://rgblab.kr/