Journal of The Korea Society of Computer and Information
Vol. 21 No. 4, pp. 1-8, April 2016
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2016.21.4.001

An Implementation of Single Stack Multi-threading for Small Embedded Systems

Yong-Seok Kim*

Abstract

In small embedded systems including IoT devices, memory size is very small and it is important to reduce memory amount for execution of application programs. For multi-threaded applications, stack may consume a large amount of memory because each thread has its own stack of sufficiently large size for worst case. This paper presents an implementation of single stack multi-threading, called SSThread (Single Stack Thread), by sharing a stack for all threads to reduce stack memory size. By using SSThread, multi-threaded applications can be programmed based on normal C language environment and there is no requirement of transporting multi-threading operating systems. It consists of several library functions and various C macro definitions. Even though some functional restrictions in comparison to operating systems supporting complete multi-thread functionalities, it is very useful for small embedded systems with tiny memory size and it is simple to setup programming environment for multi-thread applications.

▸ Keyword : Multi-threading, Stack sharing, C preprocessor, IoT

I. Introduction

사물 인터넷 (IoT: Internet of Things)을 위한 시스템들은 소형 내장형 시스템의 대표적인 예인데, 주위에서 흔히 사용하던 각종 사물들에 계산 능력과 통신 기능을 부여해서 서로 정보를 교환하도록 하는 것이 그 핵심이다[1]. 이러한 시스템을 개발하는 데 있어서 8비트 MCU (Micro-Controller Unit)에 블루투스나 와이파이 기능이 부가된 모듈이 수천원 정도의 가격에 시중에서 구매할 수 있을 정도로 하드웨어 가격은 급격하게 하락하고 있지만 응용프로그램 개발을 위한 비용은 그리 줄지 않고 있다. 특히 인터넷 통신과 멀티태스크 기능을 적용하기 위해서는 프로그램 개발 환경을 특별하게 구성해야 하고 이를 바탕으로 프로그램을 개발하는 과정도 그리 쉽지는 않다.

소형 MCU 기반의 하드웨어를 개발하는 환경에서는 기본적으로 C 언어를 적용하고 적절한 인터럽트 처리 기능을 부가할 수 있도록 하는 정도가 일반적이다. 이 정도는 하드웨어 개발 과정에서 기본으로 활용하는 환경인데 조금 복잡한 응용 프로그램을 개발하기 위해서는 여러 개의 태스크들을 적용하여 개념적으로 병행하여 실행하도록 하는 것이 좋다. 복잡한 프로그램을 태스크 별로 단순화 시킬 수 있고, 특히 태스크들을 다른 응용 분야들에 그대로 가져다 활용할 수 있는 가능성도 높아진다.

IoT 기능의 기기들을 포함하여 소형 내장형 시스템들은 실행해야 할 작업이 일정한 주기마다 반복적으로 이루어지는 경우가 많다[2,3]. 일정한 시간 간격으로 센서들로부터 데이터를 수집하고, 이들을 적절히 가공하여 다른 기기들로 전달한다. 시스템에 따라서는 일정한 시간 간격으로 액추에이터를 통하여

• First Author: Yong-Seok Kim, Corresponding Author: Yong-Seok Kim
*Yong-Seok Kim (yskim@kangwon.ac.kr), Department of Computer and Communications Engineering, Kangwon National University
• Received: 2016. 01. 04, Revised: 2016. 02. 15, Accepted: 2016. 04. 05.
• This study is supported by 2015 Research Grant from Kangwon National University (No. 520150467)

적절한 동작을 하며 여기에는 센서들로부터 수집한 데이터를 활용하기도 하고 외부 기기들로부터 전달받은 데이터를 활용하기도 한다. 따라서 이러한 목적의 응용 프로그램 개발에 있어서 태스크들이 각각 일정한 주기마다 지정된 작업을 실행하도록 하는 것이 중요하다.

저렴하고 소형인 하드웨어들에서 널리 사용되는 8비트 MCU인 AVR 칩들의 경우에는 프로그램 저장을 위한 수십 킬로바이트 정도의 플래시 메모리와 데이터 저장을 위한 몇킬로바이트 정도의 RAM을 가지고 있다[4]. 따라서 이러한 하드웨어에 적용할 수 있는 응용 프로그램을 개발하는 데 있어서 메모리 소요량을 줄이는 것 또한 매우 중요하다. 일반적인 멀티태스킹 운영체제에서는 태스크 마다 독립적인 스택을 할당해서 사용하므로 메모리 소요량이 늘어나는 문제가 있다. 따라서 태스크들 간에 스택을 공유해서 사용함으로써 메모리 소요량을 줄이는 노력들도 많이 이루어지고 있다[5,6,11].

본 논문은 하드웨어 개발과정에서 사용하는 단순한 프로그램 개발 환경에서 간편하게 멀티스레드 기반의 프로그램 개발 환경으로 전환할 수 있도록 하는 SSThread (Single Stack Thread)에 대하여 설명한다. C 컴파일러 이외의 별도의 프로그래밍 툴이 필요하지 않고 작은 크기의 C 언어 파일 하나와 헤더 파일 하나만 추가하면 된다. 스레드들을 각자의 주기마다 실행되도록 간편하게 표현할 수 있는 기능이 제공되며, 스레드들 간에 하나의 스택을 공유함으로써 멀티스레드 기능을 위해 추가로 소요되는 메모리 용량도 극히 작다. 대신에 일반적인 멀티태스킹 운영체제와는 다르게 가상적으로 멀티스레딩을 구현하고 있어서 응용 프로그램 개발에 있어서 약간의 제약점은 존재한다. 소형 모듈에서 메모리 사용량을 줄이면서도 제한적이나마 간편하게 멀티스레드 개념을 적용할 수 있도록 하는 것이 본 논문의 목적이다.

II. Related Works

순환적 실행개체 (cyclic executive) 방식은 오래전부터 간단한 시스템들에서 많이 사용해 온 것으로서, 주기적인 태스크들을 사전에 분석하여 주기들의 최소공배수인 초주기 동안의 실행 스케줄을 테이블에 기록해 놓고 시간에 따라 해당 태스크의 함수를 호출한다[7]. 이 방식의 장점은 단순하고, 오버헤드가 작으며, 실행 시간이 예측 가능하다는 것이다. 프로세스나 스레드 개념 자체가 필요하지 않으므로 멀티태스킹 운영체제가 필요하지 않고, 태스크 디스패치 절차가 없이 해당 함수 호출로 처리하므로 매우 효율적이다. 또한 태스크별로 별도의 스택을 할당할 필요가 없으므로 메모리 사용량을 절약할 수 있다. 그러나 태스크 함수 중간에 일정 시간을 대기하는 상황은 포함할 수가 없고, 실행 중간에 다른 태스크를 생성할 수도 없다. 태스크들 간의 주기가 서로 맞지 않으면 태스크 스케줄링 테이블이 커질 수 있고, 전체 태스크 집합이 일부 변경되면 전체적인 사전 스케줄링 작업을 다시 해야 한다.

멀티태스킹 운영체제를 사용하면 순환적 실행개체 방식에 비해서 응용 프로그램 개발에 있어서 자유도가 훨씬 높아진다. 그러나 각 태스크들이 독자적인 스택을 가져야 하므로 제한된 메모리를 갖는 소형 시스템에는 적절하지 못하다. 스택 자원 정책 (SRP: Stack Resource Policy)은 태스크들 간에 하나의 스택을 공유할 수 있도록 함으로써 메모리 사용량을 대폭 절약할 수 있음을 보여주었다[8]. 하나의 스택을 공유하여 사용할 수 있도록 하기 위해서는 실행중인 태스크가 더 높은 우선순위의 태스크에 의해 선점되지 않는 한 절대로 대기상태로 전환되지 않고 실행을 완료할 수 있도록 한다. 이를 위해서 실행에 필요한 자원들을 모두 확보할 수 있는 상태에서만 실행을 시작하도록 함으로써 실행 중에 자원을 추가로 할당받기 위해서 대기하는 상황이 발생하지 않도록 한다. 그러나 태스크는 실행 중에 일정시간 대기하는 것이 허용되지 않으므로 프로그램 작성에 제약이 된다.

TinyOS는 아주 작은 규모의 시스템을 위한 것으로서 싼 가격과 저전력 등의 요구조건을 만족하면서 센서 네트워크에서 센서 데이터의 수집, 처리, 통신, 및 액추에이터 구동 등의 동작을 병행하여 실행하는 것을 목적으로 개발되었다[9]. 응용 프로그램을 작성하는데 있어서 소프트웨어 부품 모듈들을 조합하여 구현하도록 유연성을 부여하기 위한 방안으로서 이벤트 구동형의 병행 실행 모델을 적용하였다. 실행중인 태스크는 순환적 실행개체 방식과 유사하게 중간에 대기상태를 허용하지 않으며, 인터럽트에 의하지 않고는 선점되지 않는다. 따라서 태스크 별로 별도의 스택이 필요하지 않고 전체적으로 하나의 스택만을 사용함으로써 메모리 소요량을 대폭 줄일 수 있다. 그러나 응용 프로그램 작성을 위해서는 C 언어의 변형인 NesC라는 새로운 언어로 프로그램을 작성해야 하므로 별도의 프로그램 개발 환경을 구축해야 하는 불편함이 따른다.

TOSThread는 TinyOS에 일반적인 멀티스레드 개념을 추가한 것으로서[10] 스레드별로 스택을 할당해야 하는 문제가 다시 대두되었는데, UnstackedC는 메모리 소요량을 줄여야 한다는 관점에서 이것을 스레드별 스택이 필요하지 않도록 변형한 것이다[5]. TOSThread 기반으로 개발된 프로그램은 별도의 변환 프로그램으로 자동 변환하여 TinyOS 기반의 시스템에 적용하도록 하였다. 이 자동 변환 프로그램은 NesC의 전처리기 역할을 하는 것이다. 하나의 스택을 공유하기 위해서는 실행중인 스레드가 다른 스레드에 의해 선점되어서는 안되며, 스레드가 실행중에 대기해야 하는 경우에는 모든 지역변수들을 별도의 자료구조로 관리하는 번거로움이 있어서 실행과정의 오버헤드가 비교적 클 수 있다.

ProtoThread는 이벤트 구동형 운영체제인 Contiki에 추가하여 사용할 수 있는 것으로서 메모리가 제한된 시스템에서 특정 이벤트에 의해 구동되는 태스크를 가상적으로 여러 개의 스레드로 구성할 수 있도록 하였다[11]. 여기서 가상적이라 함은 실제로는 멀티스레드로 실행되지 않지만 프로그래밍 관점에서는 마치 멀티스레드인 것처럼 보이게 해주는 것이다. 각 스레드는

PT_BEGIN 함수 호출과 PT_END 함수 호출로 감싸도록 한다. 중간에 특정 조건이 만족될 때 까지 대기할 필요가 있는 부분에는 PT_WAIT_UNTIL 함수에 조건을 표현하도록 하였다. 이렇게 작성된 프로그램은 별도의 스레드 패키지의 활용이 필요하지 않고 일반적인 C 컴파일러로 컴파일하는 것으로 완성된다. 즉, ProtoThread에서 추가한 함수들은 전부 C 전 처리기의 매크로로 선언되어 있어서 이들을 포함하는 헤드 파일의 include 표현만 추가하면 된다.

조건이 만족될 때까지 대기하는 것을 실제로는 해당 스레드 함수에서 그냥 복귀하도록 하고 다음 실행 기회가 될 때에 이 스레드 함수가 다시 호출되고 바로 대기 부분으로 건너뛰도록 처리하였다. 실제로는 스레드 함수가 반복적으로 호출되는 것으로 처리하므로 겉으로 보기에 스레드처럼 보일 뿐이다. 대기 조건에 해당하면 바로 스레드 함수에서 복귀하므로 스레드가 실행 중에 다른 스레드로 선점되는 일이 없으며 따라서 스레드 별로 별도의 스택을 할당할 필요도 없어지므로 스택을 위한 메모리 용량을 절약할 수 있게 된다. PT_WAIT_UNTIL는 기본적으로 비지웨이팅 방식에 바탕을 두고 있어서 CPU의 실행속도를 늦추거나 일시적으로 멈추어서 전력 소모량을 줄이도록 하는 것이 매우 어렵다.

MCU를 기반으로 하는 아주 작은 규모의 하드웨어 모듈을 개발하는 과정에 있어서, 시험용 프로그램 개발은 일반적으로 C 언어를 사용하여 응용 프로그램을 스레드 개념이 없이 작성하고 인터럽트 처리 루틴들을 적절히 추가하여 이루어진다. 본 논문에서는 이러한 환경에서 최소한의 노력만으로 멀티스레드 개념을 활용한 응용 프로그램을 개발할 수 있도록 전환하는 방안을 제시한다. 이러한 목적을 달성하기 위해서 별도의 멀티태스킹 운영체제를 도입하지도 않고 프로그램 개발 툴을 변경하지 않으면서도 멀티스레드 개념을 적용할 수 있어야하고, 또한 메모리의 사용량을 최소화하기 위해서 스레드들 간에 스택을 공유하여 하나의 스택만으로 실행되도록 하는 방안을 제시하였다. 본 논문에서 제시하는 SSThread는 ProtoThread에서와 같이 C 언어의 전처리기 방식을 적용하였다. 이 방식의 가장 큰 장점은 C 언어 기반의 일반적인 응용 프로그램 개발 환경을 전혀 변경하지 않고 그대로 사용할 수 있다는 것이다.

III. Implementation of SSThread

1. Basic Operation Principle

SSThread는 기본적인 접근 방향으로서 ProtoThread의 방법을 원용하였는데 C 언어의 전처리기에 적용하는 매크로 함수들을 정의하여 사용하는 것이다. 그림 1은 SSThread를 적용하여 간단하게 작성한 프로그램의 예이다. 스레드 2개가 생성되어 실행되며 각각 자신의 LED 번호를 받아서 토글 동작 후에 다음 스레드로 실행을 양보하도록 작성되어 있다.

```
#include "ssthread.h"
SST_THREAD Blink(int led)
{
   SST_BEGIN();
   while (1) {
      ledToggle(led);
      SST_YIELD();
   }
   SST_END();
}
int main()
{
   SST_INIT();
   SST_CREATE(Blink, 1);
   SST_CREATE(Blink, 2);
   SST_START();
}
```

Fig. 1. An Example of SSThread Application

먼저 ssthread.h 파일을 포함하는데 여기에 SSThread를 위한 매크로들이 선언되어 있다. 모든 스레드는 시작부분에 SST_BEGIN 함수를, 끝부분에 SST_END 함수를 호출해야 한다. 전체 응용 프로그램은 main 함수에서 시작하는데 항상 시작 부분에 SST_INIT 함수를, 끝부분에는 SST_START 함수를 호출해야 한다. 이 예제는 SST_INIT 함수로 SSThread를 위한 자료구조들을 초기화하고, SST_CREATE 함수를 통하여 2개의 스레드들을 생성하면서 스레드 실행 함수인 Blink의 인수 led로 전달할 값을 1과 2로 지정하였다. SST_START 함수는 스레드들이 실행을 시작하도록 한다.

스레드 함수 Blink에서 SST_YIELD 함수 호출 부분은 실행 순서를 다른 스레드에게 양보하는 것인데, 실제로는 일단 Blink에서 복귀하도록 한 다음에 다시 Blink가 호출되면 SST_YIELD 바로 다음 위치로 건너 띄게 처리함으로써 while 문을 계속 실행하는 것처럼 보이는 것이다. 이러한 동작을 하도록 ssthread.h에서 정의한 매크로 중 일부는 그림 2와 같다.

```
#define SST_BEGIN() \
  if ((SST_MYSELF->cont) != NULL) \
  goto *(SST_MYSELF->cont)

#define SST_YIELD() { \
  __label__ r; SST_MYSELF->cont = &&r; \
  return SST_STAT_READY; \
  r: SST_MYSELF->cont = NULL; }

#define SST_END()  return SST_STAT_END
```

Fig. 2. A Part of SSThread Macro Definitions

SST_MYSELF는 지금 실행되고 있는 스레드의 제어블록을 가리키며, 스레드 함수가 호출되었을 때 바로 건너뛸 위치를 cont 필드에 기록해 둔다. 스레드 생성 초기에는 cont 필드가 NULL로 설정되므로 SST_BEGIN에서 건너뛰지 않는다. SST_YIELD 부

분에서 cont 필드를 레이블 r의 위치로 기록만 해놓고 일단 SST_STAT_READY 값으로 복귀한다. 이 복귀 값은 실행가능 상태로서 다시 호출되어야 한다는 표시이다. 다음에 다시 스레드 함수가 호출되어 SST_BEGIN 부분을 실행하게 되면 레이블 r의 위치로 건너 띄게 되는 것이다. 스레드 함수의 끝에까지 오게 되면 SST_END를 호출하는데 여기서는 SST_STAT_END 값으로 복귀한다. 이 값은 스레드의 실행이 종료되었으므로 이 스레드 함수는 더 이상 호출하지 말 것을 표시한다.

SSThread는 2개의 파일 ssthread.h와 ssthread.c로 구성되는데, SST_INIT과 SST_START 함수는 ssthread.c 파일에 포함되어 있다. SST_START 함수는 main 함수에서 스레드들의 실행을 시작하기 위해서 호출하는 것인데, 상태가 실행가능 상태인 스레드들의 함수를 반복적으로 호출하는 것이 주목적이다. 주요 알고리즘은 그림 3과 같이 구성된다. 생성된 스레드들에 대한 정보는 스레드 제어블록 형태로 SSThreadTab 배열에 등록되어 있으며, 스레드들은 우선순위가 없이 돌아가면서 선택된다. 선택된 스레드의 함수를 호출하면서 스레드가 생성될 때 지정된 인수를 전달한다. 스레드 함수에서 반환되는 값은 스레드의 상태 값으로서 stat 필드에 기록하고 다음번에 선택할 때 활용된다.

```
index = 0;
while (1) {
   if (index >= MAX_THREAD) index = 0;
   thread = &SSThreadTab[index++];
   if (thread->stat == SST_STAT_READY) {
      SST_MYSELF = thread;
      thread->stat =
         (*(thread->func))(thread->arg);
  }
}
```

Fig. 3. Simple Thread Scheduling of SSThread

2. Sharing a Single Stack

C 언어 컴파일러들은 대부분 (gcc를 포함하여) 함수를 호출하는 과정에서 스택 상단에 복귀할 주소와, 전달할 인수들을 쌓고, 해당 함수로 이동한 다음, 해당 함수에서는 지역변수들을 위한 크기만큼 스택 상단에 영역을 할당한다. 함수에서 복귀하면 스택은 이전의 상태로 복구되도록 한다. 함수를 실행하는 중간에 다른 스레드로 문맥교환이 이루어지면 스택에 저장된 정보들은 그대로 유지되어야 하므로 스레드별로 별도의 스택 영역을 사용해야하는 것이다.

SSThread에서 모든 스레드의 함수는 중간에 대기상태로 가지 않고 일단 복귀한다. 다음에 실행 차례가 오면 다시 스레드 함수가 호출된다. 따라서 스레드 별로 스택에 보관해야할 정보가 아예 없는 것이다. 이렇게 스레드별로 별도의 스택을 할당할 필요가 없으므로 응용프로그램 전체에 대하여 하나의 스택만 있으면 된다. 이 스택은 인터럽트가 발생할 경우에 그 처리 루틴에서도 그대로 사용될 수 있다. 따라서 스택 공간을 위한 메모리 용량을 대폭 줄일 수 있게 된다.

대신에 스레드 함수의 작성에는 몇 가지 제약이 있다. 첫째로 스레드는 한번 호출되면 그 실행시간이 일정한 한도 내로 제한되어야 하고 바로 복귀하여야 한다. 따라서 시간이 오래 걸리는 반복문 등에서는 중간에 반드시 SST_YIELD나 SST_SLEEP_MSEC, SST_PERIOD_WAIT 등의 함수 호출이 있어야 한다. 이러한 구조는 순환적 실행개체[7]와 동일한 원리이다. 즉, 실행 가능 상태의 스레드들은 그 함수가 돌아가면서 호출되어야 하므로 한 스레드에서 오랫동안 실행하게 되면 다른 스레드들에게 실행 기회가 오지 않을 것이다.

두 번째 제약은 지역변수를 사용할 수 없다는 점이다. 지역 변수는 스택의 영역에 할당되므로 일단 스레드 함수에서 복귀하는 순간 그 스택 공간은 다른 목적으로 사용된다. 따라서 다음에 스레드 함수가 다시 호출되는 시점에는 이전의 값이 유지되지 않는다. 만약 값을 계속 유지해야 하는 지역 변수가 필요하면 static을 선언하여 스택이 아닌 고유의 메모리 공간이 확보되도록 하면 된다. 그런데 동일한 스레드 함수를 여러 스레드들이 공유하는 경우에는 이 static 변수도 공유되므로 주의해야 하며, 이 경우를 대비하여 스레드별로 별도의 값을 저장할 수 있는 영역으로 SST_PRIVATE 배열을 제공한다. 이것은 스레드 제어블록의 일부에 기록되도록 한다.

세 번째로는 스레드의 기본 실행함수 (시작함수) 이외에는 SSThread 관련 함수들을 호출할 수가 없다. 기본 실행함수에서 호출하는 함수들은 대기상태로 들어가거나 하지 않고 바로 복귀할 수 있는 것들만 가능하다. 이러한 제약은 스택에 스레드 별로 지역변수를 포함한 상태정보들을 저장하지 않기 위해서 모든 스레드는 일단 기본 실행함수에서 복귀해야하기 때문이다. 소형 시스템들에서는 응용 프로그램이 비교적 단순하기 때문에 이러한 제약은 실제로는 별로 문제가 되지 않을 것이다.

3. Primitives for Periodic Threads

소형 내장형 시스템들은 일정한 주기마다 반복적으로 실행하는 경우가 많은데 일정한 시간 간격으로 센서들로부터 데이터를 수집하고, 액추에이터를 통하여 적정한 동작을 하며, 다른 기기들과 통신을 한다[2,3]. 따라서 이러한 목적의 응용 프로그램 개발에 있어서 스레드들이 각각 일정한 주기마다 지정된 작업을 실행하도록 하는 것이 중요한데, SSThread는 스레드 별로 주기를 설정하고 다음 주기까지 대기했다가 실행을 계속하는 기능을 제공한다. 그림 4는 매 100msec 마다 적절한 작업을 하도록 작성한 스레드의 예이다.

```
SST_PERIOD_SET(100);
while (1) {
    SST_PERIOD_WAIT();
    ActionForEachPeriod();
}
```

Fig. 4. Periodic Thread of 100msec Period

SST_PERIOD_SET 함수는 현재 스레드의 주기를 밀리초 단

위로 설정하도록 한다. 주기적인 실행 작업은 반복문으로 표현하고 SST_PERIOD_WAIT 함수는 다음 주기가 될 때까지 대기하다가 깨어나는 기능을 한다. 이러한 목적을 달성하기 위해서는 스레드 제어블록에 주기와 현재 주기 시간 필드를 추가하고, SST_PERIOD_SET 함수에서 이러한 정보를 설정하도록 한다. SST_PERIOD_WAIT 함수는 현재주기의 시간에 주기 길이만큼을 더한 시간을 다음 주기시간으로 설정하고 그 시간이 될 때까지 대기상태 (SST_STAT_WAIT)로 전환한다. SST_START 함수에서는 대기상태인 스레드들에 대해서는 설정한 시간과 현재시간을 비교해서 대기시간에 도달했으면 그 스레드의 함수를 호출하도록 한다.

스레드 프로그램을 작성하는데 있어서 일정한 시간이 지난 후에 깨어나도록 할 필요가 흔히 존재한다. 이러한 목적으로 제공되는 것이 SST_SLEEP_MSEC 함수이다. 이 함수도 대기시간을 스레드 제어블록에 기록하고 대기상태로 전환되도록 하고, SST_START 함수에서 시간을 검사하여 처리하도록 한다.

그림 5는 SSThread의 매크로 정의를 보여준다. 여기서 사용하는 함수 SST_GET_TIME 은 반환 값은 초단위이고 밀리초 단위의 값은 인수를 통해서 받는다.

```
#define SST_SLEEP_MSEC(t) { \
 __label__ r; int msec; \
 SST_MYSELF->cont = &&r; \
 SST_MYSELF->wsec = SST_GET_TIME(&msec); \
 msec += t; \
 SST_MYSELF->wsec += msec / 1000; \
 SST_MYSELF->wmsec = msec % 1000; \
 return SST_STAT_WAIT; \
 r: SST_MYSELF->cont = NULL; }

#define SST_PERIOD_SET(t) \
 SST_MYSELF->period = t, \
 SST_MYSELF->psec = \
   SST_GET_TIME(&SST_MYSELF->pmsec)

#define SST_PERIOD_WAIT() { \
 __label__ r; SST_MYSELF->cont = &&r; \
 SST_MYSELF->pmsec += SST_MYSELF->period;\
 SST_MYSELF->psec += \
   SST_MYSELF->pmsec / 1000; \
 SST_MYSELF->pmsec %= 1000; \
 SST_MYSELF->wsec = SST_MYSELF->psec; \
 SST_MYSELF->wmsec = SST_MYSELF->pmsec; \
 return SST_STAT_WAIT; \
 r: SST_MYSELF->cont = NULL; }
```

Fig. 5. Macro Definitions of SSThread

대부분의 운영체제나 스레드 패키지에서는 일정시간 대기하는 기능은 제공하지만 일정한 주기마다 실행하도록 하는 기능은 거의 없다. 일정시간 대기하는 기능만을 사용한다면 한 주기는 실제 실행시간과 대기 시간을 더한 것이 되는데 주기마다 실행시간이 상황에 따라서 다를 수도 있으므로 정확하게 주기를 맞추는 것이 매우 어려울 것이다. SSThread의 주기적 실행 기능은 이러한 목적의 프로그램을 작성하는데 있어서 매우 편리할 것이다.

4. Primitives for Thread Control

SSThtread에서 멀티스레드 기능을 위해 지원하는 함수들은 표 1과 같다. 스레드들은 main 함수에서 SST_START를 호출하기 전에 생성될 수도 있고 스레드가 실행 중에 다른 스레드를 생성하는 것도 가능하다. SST_JOIN은 임의의 다른 스레드를 지정하여 그것이 끝날 때까지 대기하도록 한다. 이에 비해서 ProtoThread에서는 실행 중에 스스로 생성한 스레드에 한해서 무조건 대기하도록 제한되어 있다[11]. SST_SLEEP과 SST_WAKEUP은 PtotoThread에서는 지원되지 않는 기능들이다. 특히 SST_SLEEP은 인터럽트 처리 루틴에서 호출할 수 있는 ISR_WAKEUP과 연계하여 사용하면 특정 인터럽트가 발생한 시점에 지정된 스레드가 깨어나서 실행을 재개할 수 있도록 할 수 있다.

Table 1. Primitives of SSThread

함수	기능
ssThread *SST_CREATE (int (*func)(int), int arg)	인수 func에 지정된 함수를 실행하는 스레드를 생성함; 인수 arg는 실행함수에 전달할 인수
void SST_JOIN(ssThread* th)	인수로 지정된 스레드가 종료될 때까지 대기하였다가 실행을 재개함
void SST_YIELD(void)	다른 스레드에게 실행 순서를 양보함
void SST_SLEEP(void)	다른 스레드가 깨울 때까지 대기함
void SST_WAKEUP(ssThread *th)	인수로 지정된 스레드가 깨어나서 실행을 재개하도록 함
void ISR_WAKEUP(ssThread *th)	인터럽트 처리 루틴에서 호출하며, 인수로 지정된 스레드가 깨어나서 실행을 재개하도록 함
void SST_SLEEP_MSEC(int msec)	인수로 지정된 밀리초 단위의 시간동안 대기했다가 실행을 재개함
void SST_PERIOD_SET(int msec)	인수로 지정된 밀리초 단위의 시간을 주기로 설정함
void SST_PERIOD_WAIT(void)	다음 주기에 해당하는 시점까지 대기했다가 실행을 재개함

ProtoThread에서는 임의의 조건을 지정하여 그것이 만족될 때까지 대기하는 기능인 PT_WAIT_UNTIL 함수를 제공한다. 이것은 조건을 자유롭게 지정할 수 있는 장점은 있지만 비지웨이팅에 기반을 두고 있으므로 끊임없이 이 조건을 반복적으로 검사해야 한다. 소형 내장형 시스템들은 일반적으로 전력소모량을 줄이는 것도 중요한 이슈중 하나이다. 특히 휴대형이거나 장기간 충전없이 사용해야 하는 환경에서는 이것이 매우 중요한 항목이다. SSTThread에서는 비지웨이팅을 사용하지 않고

단순히 일정한 시간동안 대기했다가 깨어나도록 하는 SST_SLEEP_MSEC 함수만 제공한다. 일정한 시간을 대기한 후에 조건을 다시 검사하는 과정을 반복하도록 하고, 대기할 시간은 필요에 맞게 적절히 지정하면 된다. 따라서 모든 스레드가 대기상태이면 SST_START 부분에서 적절히 CPU를 쉬도록 처리하는 것이 가능해진다.

IV. Performance Comparison

1. Programming Environment

표 2는 프로그램 개발환경 관점에서 SSThread를 ProtoThread와 UnstackedC를 비교한 것이다. ProtoThread는 이벤트 구동형 Contiki 운영체제를 보완하는 목적으로 개발된 것인데 반해서 SSThread는 다른 운영체제가 없이 그냥 main 함수로 시작하는 응용 프로그램에 멀티스레드 개념을 적용하는 것을 목적으로 한다. 별도의 기반 운영체제가 필요하지 않으므로 멀티스레드 개념이 없이 단순하게 응용 프로그램을 개발하던 환경을 아무런 변경도 없이 그대로 사용할 수 있다. 기반 운영체제가 필요하면 그 운영체제를 이식하는 데에 많은 수고와 비용을 감수해야 할 것이다. 특히 UnstackedC의 경우에는 TinyOS에서 필요로 하는 NesC 컴파일러와 자체 개발한 자동변환 프로그램을 별도로 설치해야만 사용이 가능하다.

Table 2. Comparison of Programming Environment

비교 항목	SSThread	Proto-Thread	UnstackedC
기반 운영체제	필요 없음	Contiki 기반	TinyOS 기반
C 컴파일러 이외의 개발 툴	필요 없음	필요 없음	NesC 컴파일러 및 자동변환 프로그램
인터럽트 처리 루틴과의 연계 기능	담당 스레드 깨우기	없음	없음

인터럽트 처리 루틴은 내장형 시스템에서 매우 중요하게 사용되는데 이것을 멀티스레드 기반의 응용 프로그램 개발에 있어서 쉽게 포함할 수 있어야 한다. 하드웨어를 개발하는 과정에서 필요한 인터럽트 처리 루틴들이 이미 작성되어 있는 경우에 SSThread에서는 그대로 사용하면 된다. 특정 인터럽트 처리 과정이 많은 시간을 요하는 경우에는 다른 인터럽트들에 대한 인터럽트 지연시간이 길어질 우려가 있다. 이러한 경우에는 인터럽트 처리 루틴에서는 최소한의 작업만 하고 나머지는 차후에 다른 방법으로 실행하도록 하는 것이 필요해진다. SSThread에서는 인터럽트 처리 루틴과 스레드 실행을 연계할 수 있도록 인터럽트 처리루틴에서 ISR_WAKEUP 함수 호출을 통하여 대기 중인 특정 스레드를 깨워서 나머지 작업을 실행하도록 할 수 있다.

2. Primitives

지원하는 기능들에 대하여 ProtoThread[11]와 비교해 본다. UnstackedC는 TOSThread의 기능들을 지원하지만 훨씬 복잡한 개발환경을 필요로 하므로 단순한 개발환경을 목표로 한다는 관점에서 비교대상에서 제외한다. ProtoThread는 기본적으로 PT_YIELD, PT_WAIT_UNTIL 및 PT_SPAWN의 세 가지의 함수를 활용한다. SSThread에서는 스레드가 실행 중에 다른 스레드들을 자유롭게 생성할 수 있고 SST_JOIN을 이용하여 임의의 다른 스레드를 지정하여 대기할 수 있도록 자유로운데 반해서, ProtoThread에서는 실행 중에 PT_SPAWN을 이용하여 스스로 생성한 스레드에 한해서 무조건 대기하도록 제한되어 있다. SST_SLEEP과 SST_WAKEUP은 PtotoThread에서는 지원되지 않는 기능이다. 특히 SST_SLEEP 기능은 인터럽트 처리루틴에서 호출할 수 있는 ISR_WAKEUP 기능과 연계하여 사용하면 특정 인터럽트가 발생한 시점에 지정된 스레드가 실행을 재개할 수 있도록 할 수 있다.

스레드가 특정 조건이 만족될 때까지 기다려야 하는 경우에, ProtoThread에서는 PT_WAIT_UNTIL을 활용하여 필요한 조건을 자유롭게 지정할 수 있는데 반해서 SSThread에서는 SST_SLEEP_MSEC를 활용하여 일정 시간 동안 대기하는 기능만 제공하고 응용 프로그램 개발자가 적절히 알고리즘으로 해결하도록 한다. 즉, 반복문에 조건을 표현하고 이것이 만족되지 않으면 SST_SLEEP_MSEC로 적절한 시간동안 대기한 후에 다시 반복문의 조건을 검사하도록 한다. 대기 시간은 목적에 맞게 적절히 (가능하면 길게) 지정하면 된다. PT_WAIT_UNTIL은 조건을 자유롭게 지정할 수 있는 장점이 있지만 비지웨이팅에 기반을 두고 있으므로 끊임없이 이 조건을 반복적으로 검사한다. 이것은 소형 내장형 시스템들에서 전력 소모량을 줄여야 하는 관점에서 치명적이다. SSThread에서는 비지웨이팅 대신에 일정 시간동안 대기하도록 함으로써 지나치게 빈번하게 조건을 검사하는 오버헤드를 줄일 수 있고 스레드들이 모두 대기중인 시점에는 CPU를 쉬게 함으로써 전력 소모량을 줄일수 있는 여지도 있다.

3. Memory Size

SSThread를 적용한 경우에 소요되는 메모리 크기는 표 3과 같다. avr-gcc 컴파일러를 이용하여 8비트 AVR 코드를 생성하였으며, 메모리 크기는 실행 파일을 avr-size 프로그램으로 측정하였다. 스레드를 3개 생성할 경우에 코드를 위해서 922바이트와 데이터 저장을 위해서 70바이트가 필요하므로 총 992바이트가 소요된다. 한 개의 스레드를 추가하면 코드가 76바이트 늘어나고 데이터가 20바이트 늘어난다. 여기서 데이터 20바이트는 추가된 스레드 제어블록을 위한 메모리이고, 코드 76바이트는 SST_BEGIN과 SST_END만 포함되고 실행 내용이 비

어있는 것을 스레드 함수를 적용한 경우의 메모리 크기이다. TinyOS에서 활용하는 대표적인 예제중의 하나인 Blink는 3개의 스레드들이 각기 담당 LED를 자기의 주기에 따라서 켜고 꺼고를 반복하는 것이다[12]. Blink 예제를 SSThread 에 적용한 결과 소요 메모리 용량은 총 1728 바이트 정도로 아주 작다.

Table 3. Memory Size of SSThread (bytes)

기 준	코드	데이터	합계
기본 (스레드 3개 기준)	922	70	992
스레드 1개 추가 당	76	20	96
Blink 예제 (스레드 3개)	1658	70	1728

UnstackedC나 ProtoThread와의 메모리 소량 비교는 동일한 예제를 통하여 간접적으로 할 수 있다. 참고문헌 [5]에 의하면 Blink 예제를 적용하여 측정한 결과는 표 4와 같다. TinyOS를 기반으로 UnstackedC를 적용한 경우와 Contiki를 기반으로 ProtoThread를 적용한 경우를 비교하고 있는데 SSThread가 이들에 비해서 대략 절반 정도의 메모리만 필요로 한다. 이러한 결과는 SSThread가 기반 운영체제를 사용하지 않는다는 점이 그 핵심 이유일 것이다.

Table 4. Memory Size for Blink Example (bytes)

종 류	코드	데이터	합계
UnstackedC/TinyOS	3760	152	3912
ProtoThread/Contiki	3186	66	3252
SSThread	1658	70	1728

IV. Conclusions

IoT를 적용하는 응용들을 포함하여 소형 내장형 시스템들은 다양한 분야에서 다양한 수준의 개발자들이 하드웨어 및 응용 소프트웨어 개발에 참여하고 있는데, 응용 프로그램을 C 언어로 멀티스레드 개념이 없이 단순하게 개발하는 환경에서 시작한다. 하드웨어는 메모리 용량이 극히 제한되는 경우가 많고, 배터리 사용량을 최대한 줄이는 것도 중요한 이슈이다.

본 논문에서 제시하는 SSThread는 이러한 분야에 적용하는 것을 목적으로 개발하였다. C 언어로 응용 프로그램을 멀티스레드 개념이 없이 단순하게 개발하는 환경에서 멀티스레드 기능을 쉽게 활용할 수 있도록 한다. 별도의 운영체제 이식 작업도 필요하지 않고, 별도의 프로그램 개발 툴도 필요하지 않다. SSThread가 제공하는 스레드 기능은 가상적으로 처리되므로 본격적인 멀티태스킹 운영체제가 제공하는 정밀한 스레드 제어는 할 수 없지만 소규모의 간단한 응용들에는 매우 유용하게 사용될 수 있을 것이다.

SSThread는 단순한 개발환경에서 간단하게 멀티스레드 기능을 적용할 수 있으면서도 메모리 용량이 극히 제한된 소규모 시스템들에 적용할 수 있도록 스레드들 간에 하나의 스택을 공유하도록 하였다. 응용 프로그램을 개발하는 데 있어서 유용한 추가 기능들로는 스레드마다 일정한 주기를 지정하여 실행할 수 있으며, 스레드들 간의 동기화를 위하여 대기와 깨우기를 적용할 수 있다. 특히 대기중인 스레드를 깨우는 기능을 인터럽트 처리 루틴에서 호출할 수 있도록 함으로써 긴 시간이 소요되는 인터럽트 처리 루틴은 특정 스레드가 담당하도록 할 수 있게 한다.

최근에 IoT가 핵심 이슈로 부각되고 있는데 이러한 목적에 SSThread가 유용하게 활용될 수 있을 것이라고 기대된다. IoT 분야에서 다양하게 활용되기 위해서는 블루투스나 와이파이 통신을 위한 프로토콜 스택을 개발하여 함께 제공하는 것이 필요하며, 그 외에도 다양한 응용 분야에 적용해 보고 현장에서 필요로 하는 기능들을 지속적으로 반영하여 개선해 나가는 노력도 필요하다.

REFERENCES

[1] K. Kim and J. Shin, “Early Stage of IoT Market, From Things rather than Internet”, LG Business Insight, LG Economic Research Institute, pp.2-13 March 2015.

[2] P. A. Laplante, Real-Time Systems Design and Analysis, 3rd ed., IEEE Press, 2004.

[3] H. J. Park and C. H. Lee, “An Efficient Real-Time Middleware Scheduling Algorithm for Periodic Real-Time Tasks,” Springer-Verlag, Artificial Intelligence and Simulation, pp 304-312, 2005.

[4] Atmel Microcontrollers, *http://www.atmel.com/products/microcontrollers/avr/.*

[5] W. P. McCartney and N. Sridhar, “Stackless Preemptive Multi-Threading for TinyOS,” Proceedings of the 2011 International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 1-8, 2011.

[6] T. Reusing, “Comparison of Operating Systems TinyOS and Contiki,” Network Architectures and Services, pp. 7-13, August 2012.

[7] I. C. Bertolotti and T. Hu, Embedded Software Development: The Open-Source Approach, CRC Press, 2015.

[8] T.P. Baker, "Stack-Based Scheduling of Realtime

Processes", The Real-Time Systems Journal, pp.67-100, March 1991.

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh and E. Brewer, "TinyOS: An Operating System for Sensor Networks," Ambient Intelligence, Springer, pp. 115-148, 2005.

[10] K. Klues, C.J.M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, and R. Govindan, "TOSThreads: thread-safe and non-invasive preemption in TinyOS," Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, pp. 127-140, 2009.

[11] A. Dunkels, O. Schmidt, T. Voigt and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," Proc. LCN'04, IEEE Computer Society, pp. 455-462, 2004.

[12] TinyOS Community Website, http://www.tinyos.net.

Author

Yong-Seok Kim received B.S. degree in Oceanography from Seoul National University, Korea, in 1984, and M.S. and Ph.D. degrees in Electric and Electronics Engineering from KAIST (Korea Advanced Institute of Science and Technology), Korea, in 1986 and 1989, respectively. Dr. Kim is a professor in Department of Computer and Communications Engineering at Kangwon National University, Kangwon-do, Korea, from 1995. He was a research staff of KETI (Korea Electronics Technology Institute) in 1994, and KITECH (Korea Institute of Industrial Technology) from 1990 to 1993. He is interested in system software for real-time and embedded systems, and internet of things.