타스크 프레임워크를 제공하는 메시징 기반 선점형 임베디드 실시간 운영체제의 개발

차정윤O 안순신
고려대학교 전자전기 공학과
{jychaO, sunshin}@dsys.korea.ac.kr

Development of messaging based pre-emptive embedded RTOS providing Task Framework

JungYoon ChaO , Sunshin An
Computer Network Lab. Dept. of Electronic Electrical Eng., Korea University

요 약

유비쿼터스 컴퓨팅 환경이 급격히 발전함에 따라 유비쿼터스 환경을 구성하는 임베디드 디바이스의 종류와 수가 빠른 속도로 늘어나고 있으며 이에 따라 이러한 디바이스를 구동하는 내장형 실시간 운영체제의 연구 개발도 많은 발전을 이루고 있다. 하지만 이렇게 많은 종류의 실시간 운영체제가 존재함에 따라 각 운영체제 별 응용 프로그램의 개발 방법이 상이하여 혼란을 야기시킬 수 있는 가능성을 내포한다. 따라서 본 논문에서는 메시징 기반의 운영체제를 이용하여 사용자 응용 프로그램을 구조화 할 수 있는 방법인 타스크 프레임워크를 제시하고 이러한 타스크 프레임워크를 도입함에 따라 얻을 수 있는 자원 효율 면에서의 이점을 보인다.

1. 서 론

최근 들어 유비쿼터스 컴퓨팅 환경으로의 이행이 급격히 가속화 됨에 따라[1] 이러한 유비쿼터스 컴퓨팅 환경을 구성하는 임베디드 디바이스 및 이를 구동하는 실시간 운영체제 (embedded real-time operating system) 에 관련된 연구가 활발히 진행되고 있으며[2] 그 결과로 다양한 응용분야를 지원하는 많은 종류의 실시간 운영체제가 출현하고 있다[3]. 이러한 실시간 운영체제들은 각기 고유한 특징을 지니고 있으며 많은 경우 적용 대상 디바이스에 특화(최적화)된 형태를 지닌다. 따라서 이러한 실시간 운영체제를 이용하는 응용 프로그램의 형태도 각 운영체제에 따라 종속적이다. 이러한 사실 때문에 동일한 결과를 얻기 위한 응용 프로그램의 모습이 어떤 운영체제를 선택하는가에 따라 구조적으로 큰 차이를 보이며 이는 다른 운영체제로의 포팅을 어렵게 만드는 요소가 됨은 물론이고 동일 운영체제 플랫폼 상에서 동작하는 각 타스크의 구현에 있어서도 코드의 중복을 가져올 수 있다. 따라서 본 논문에서는 메시징 기반의 실시간 운영체제를 구현하고 이 바탕 위에 정형화된 형태의 타스크를 구동할 수 있는 프레임워크를 제공함으로써 사용자로 하여금 구조화된 응용 프로그램의 구성을 가능하게 하는 방법을 제안하고 실제 이러한 방법을 적용하였을 경우 얻을 수 있는 자원 효율성의 증대에 관해 알아본다.

2. 메시징 기반의 실시간 운영체제 개발

메시징 기반의 실시간 운영체제란 타스크 관점에서의 입/출력이 모두 타스크간 통신에 의해 이루어 지는 운영체제를 의미한다. 타스크간 통신에는 메시지, 이벤트, 타이머 등을 예로 들 수 있는데 본 논문에서 제시하는 운영체제에서는 위의 세가지 모두를 구현하여 사용자가 응용프로그램을 구성할 때 기능상의 부족함이 없도록 한다. 아래의 그림 1 에서 운영체제의 구조 및 타스크 프레임워크의 위치를 파악할 수 있다.

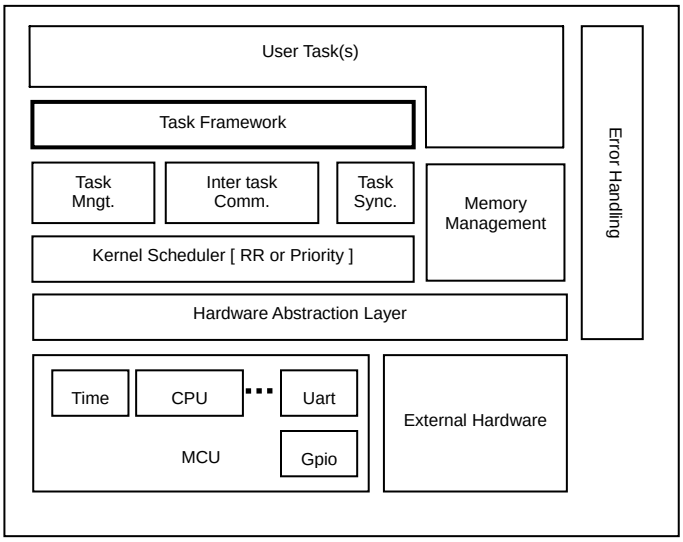

그림 1. 운영체제 구조

타스크 프레임워크의 아래에 위치하는 운영체제

커널의 구조는 일반적인 실시간 임베디드 운영체제의 구조와 크게 다르지 않으며 아래의 각 절에서 본 운영체제를 구성하는 주요 요소에 대해 설명한다.

2.1 커널 스케줄러

선점형 실시간 운영체제는 필수적으로 각 타스크의 CPU 점유를 관리하는 기능 (스케줄링 기능)을 가져야 한다[4]. 이를 구현하기 위해서는 먼저 상호 독립적으로 수행되는 각각의 타스크에 대해 task control block 이라는 타스크 별 고유 속성값을 저장하는 관리 구조를 할당해야 하며 또한 각 타스크는 각자의 수행공간을 별도로 가져야 하므로 n 개의 타스크에 대해 1:1 매칭되는 n 개의 stack 공간의 확보가 필수적이다[5].

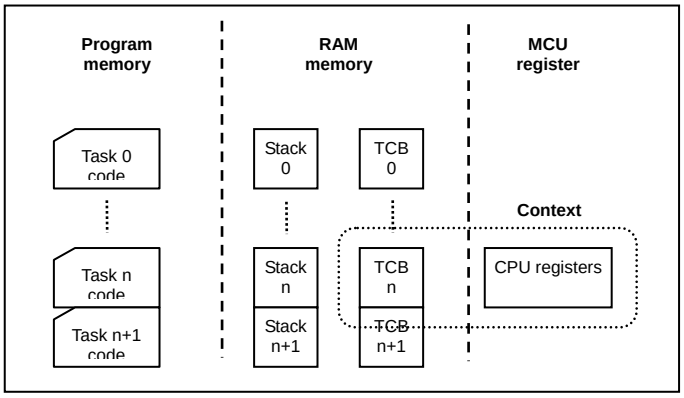

그림 2. 코드, 스택, TCB, 콘텍스트

여러 개의 타스크 중 현재 수행중인 단 하나의 타스크를 current task 라 지칭하며 이것은 우선순위나 time slice 와 같은 kernel 스케줄러의 정책에 의해 다른 타스크로 대체될 수 있는데 이 과정을 context switching 이라 하며 context 는 그림 2 와 같이 task control block (TCB) 및 스위칭 당시의 CPU register 의 상태로 구성된다.

2.2 타스크

타스크의 실행과 관련된 모든 정보는 TCB 와 해당 타스크의 스택에 기록되어 있다. (context switching 시) 타스크 스택에는 CPU register 의 값과 실행중인 타스크의 주소가 기록되며 이 외의 다른 모든 타스크 속성치는 TCB 에 기록된다. TCB 는 다음과 같은 구조체로 구성되어 있다.

표 1. TCB structure

```
typedef struct s_tcb
{
```

변수명	설명
p_curstack	현재 access 중인 타스크 스택의 주소.
p_stackbegin	스택의 시작번지
stacksize	스택의 크기. context switching 시 CPU register 가 모두 저장되므로 최소 64 byte 이상으로 설정해야 한다.
task_priority	타스크 우선순위. idle task 는 0으로 제일 낮으며 최대 255까지 설정 가능하다.
task_id	타스크 번호. TCB 배열에 대한 인덱스로 사용되므로 순서대로 1씩 증가하며 부여되어야 한다.
task_state	타스크의 상태.
sema_id	타스크가 기다리고 있는 세마포어 번호.
task_queue	타스크로 전달된 커널 오브젝트를 대기시키는 타스크 큐
p_next_tcb	다음 타스크
} t_tcb;	

2.3 타스크간 통신

타스크의 입/출력이 타스크간 통신을 통해 이루어지기 때문에 이를 구현하는 메시징 시스템이 필요하다. 따라서 본 운영체제에서는 이벤트, 메시지, 타이머 세 가지의 타스크간 통신 방법을 구현하며 이 모두는 커널 오브젝트라는 하나의 통합된 구조체에 의해 관리된다. 아래의 각 절에서 메시징 시스템의 각 구성 요소에 대해 설명한다.

커널 오브젝트 :

커널 오브젝트는 각 타스크의 큐로 전달될 수 있는 단일 포맷의 구조체로 내부적으로는 union 의 형태를 띠고 있다.

```
typedef struct s_kernel_object {
    union {
        t_message message;
        t_timer     timer;
        t_event     event;
    }u_object;
    u8 object_type;
    struct s_kernel_object *p_next_object;
}t_kernel_object;
```

그림 3. 커널 오브젝트 구조체

따라서 전달된 커널 오브젝트의 실제 타입을 결정하기 위해서는 object_type 을 참조해야 하며 커널 오브젝트 간의 연결을 위해 연결 리스트로 구성될 수 있도록 p_next_object 를 구조체의 마지막 멤버로 삽입하였다.

이러한 커널 오브젝트를 관리하기 위해 운영체제는 다음과 같은 API 를 제공한다.

표 2. 커널 오브젝트 관련 API 목록

커널 오브젝트 관련 API 목록	설명
__init_objects	초기화 과정. 빠른 occupy 와 release 를 위해 각각의 커널 오브젝트를 연결 리스트의 형태로 연결시킨 후 첫 번째 free 커널 오브젝트를 free object head 에 기록한다.
__occupy_object	free object list 로 부터 한 개의 커널 오브젝트를 가져온다.
__release_object	사용을 마친 커널 오브젝트를 free object list 에 반환한다.
__wait_for_object	타스크 큐에 커널 오브젝트가 있는 경우 선입선출 방식으로 꺼내어 처리한다. 처리를 마친 후 더 이상 처리할 커널 오브젝트가 없는 경우에는 타스크의 상태를 '대기상태' 로 변경한 후 새로운 커널 오브젝트가 전달될 때까지 대기하게 된다.
__post_object	수신 타스크로 한 개의 커널 오브젝트를 전달(post) 한다. 이 때 수신 타스크가 송신 타스크 보다 높은 우선순위를 가지는 경우 제어권은 바로 수신 타스크로 넘어가게 된다.

타스크 프레임워크를 사용하지 않는 응용 프로그램의 경우 위와 같은 커널 오브젝트 API 를 직접 이용하여 커널 오브젝트의 생성, 전달 및 타스크 큐 대기(모니터링) 등을 구현하여야 한다.

이벤트 :

앞부분에 설명된 바와 같이 타스크간 통신을 위해 세가지 방법을 제공하고 있는데 그 중 가장 간단한 것이 이벤트이다. 이벤트는 메시지와는 달리 커널 오브젝트 자체로 모든 정보를 표현할 수 있기 때문에 특정 사건의 발생 여부가 주 관심 대상인 경우 이용한다. 사건의 발생 여부와 더불어 간단한 형태의 매개변수를 덧붙일 수도 있다.

```
typedef struct s_ event {
    u8   sender;            // 이벤트를 발생시킨 타스크
    u8   receiver;          // 이벤트를 받는 타스크
    u8   event_id;          // 이벤트 식별자
    u8   param_u8;          // 1 byte 범용 파라미터
    u16 param_u16;          // 2 byte 범용 파라미터
    u32 param_u32;          // 4 byte 범용 파라미터
} t_event;
```

그림 4. 이벤트 구조체

메시지 :

메시지는 이벤트와는 달리 메시지와 함께 전달하는 내용이 의미를 가지는 경우에 이용한다. 따라서 메시지의 내용을 담을 수 있는 메시지 바디를 외부의 메모리 공간에서 할당하여 연결시켜 주어야 하며 메시지를 받는 수신 타스크 측에서는 적절한 시기에 메시지 바디를 위해 할당 되었던 메모리를 해제해야만 한다.

```
typedef struct s_message {
    u8 sender;              // message 를 발생시킨 task
    u8 receiver;            // message 를 받는 task
    u8 message_id;          // message 식별자
    u8 message_length;      // message body 의 길이
    u8 *p_message_body;     // message body 포인터
} t_message;
```

그림 5. 메시지 구조체

타이머 :

타이머는 특정한 시간에 특정한 코드를 수행시키기 위한 목적으로 사용한다. 이벤트나 메시지의 경우 타스크간에 직접적인 커널 오브젝트의 전달이 이루어 지지만 타이머의 경우 타이머 오브젝트 큐에 입력된 후 커널 스케줄러의 관리 하에 만료 시점이 계산되고 만료되는 경우 타스크 큐에 전달된다.

특정한 시간 간격으로 계속해서 타이머를 발생시키고 싶은 경우에는 '자동 반복' 기능을 이용하면 되는데 이 경우 스케줄러는 타이머 오브젝트를 복사하여 하나는 타스크 큐로 전달하고 나머지 하나는 또 다시 타이머 큐에 입력하여 다음 만료시점까지 보관하게 된다.

```
typedef struct s_timer {
    u8   receiver;          // 수신 타스크 식별자
    u8   timer_id;          // 타이머 식별자
    u32 repeat_period;      // 반복될 시간간격 (duration)
    u32 absolute_tick;      // 만료될 시점
} t_timer;
```

그림 6. 타이머 구조체

3. 타스크 프레임워크 구현

타스크 프레임워크는 본 운영체제의 특징적인 부분 중 하나로 응용프로그램의 구성을 더욱 쉽게 할 수 있도록 운영체제 차원에서 하나의 틀을 제공하는 것이다.

응용프로그램은 타스크의 형태로 구현되며 이러한 타스크는 내부적으로 커널과의 인터페이스를 통해 제어 구조를 형성한다. 이러한 제어 구조를 통해 타스크의 스케줄링이나 타스크간의 통신을 구현할 수 있으며

이것이 결국 응용 프로그램의 형태를 가지는 것이다.

응용프로그램의 목적과 사용자의 코딩 스타일에 따라 구현된 결과물은 천차만별일 수 있지만 어떠한 경우에도 일정수준의 제어구조를 갖추는 형태는 피할 수 없다. 따라서 하나의 시스템 상에서 여러 개의 응용프로그램, 즉 여러 개의 타스크를 구동하는 경우 각각의 타스크마다 개별적으로 일정 부분의 제어구조를 포함하게 되며 이러한 제어구조는 대개의 경우 공통적인 특성을 지닌다.

본 논문에서 제안하는 운영체제에서는 이러한 공통된 제어구조의 틀을 운영체제 차원에서 제공함으로써 사용자로 하여금 규격화되고 신뢰성 있는 응용 프로그램 작성을 할 수 있도록 돕고 결과적으로 개발 기간의 단축을 꾀할 수 있는 가능성을 제공하고 있는데 이것이 바로 타스크 프레임워크인 것이다.

타스크 프레임워크 로 통합할 응용 프로그램의 제어구조를 결정하기 위해 일반적인 응용프로그램이 공통적으로 가지는 제어구조에 대해 나열해 보면 다음과 같다.

- 타스크의 시작 시 초기화 루틴을 실행시키는 제어 구조
- 현재 타스크 큐가 비어 있는 경우 새로운 커널 오브젝트 (메시지, 타이머, 이벤트) 가 전달될 때까지 타스크를 대기 시키는 제어 구조.
- 대기중인 타스크에 메시지 오브젝트가 전달되었을 경우 해당 처리 루틴으로 분기하는 제어 구조.
- 대기중인 타스크에 타이머 오브젝트가 전달되었을 경우 해당 처리 루틴으로 분기하는 제어 구조.
- 대기중인 타스크에 이벤트 오브젝트가 전달되었을 경우 해당 처리 루틴으로 분기하는 제어 구조.

위와 같은 제어구조를 각각의 타스크에서 별도로 구현하는 것은 대단히 비효율 적이므로 본 운영체제에서는 다음과 같은 일반화된 제어구조 (즉, 타스크 프레임워크) 를 제공하고 각각의 타스크가 필요한 처리루틴만을 개별적으로 구현하여 등록하는 형식을 제시한다.

다음의 그림에서 타스크 프레임워크 의 구조를 확인할 수 있다. 실선으로 표시된 부분이 타스크 프레임워크 에서 제공하는 제어 구조이고 점선으로 표시된 부분은 사용자가 응용프로그램에 적합하게 내용을 구현하여 타스크 프레임워크 에 바인딩 시켜야 할 부분이다.

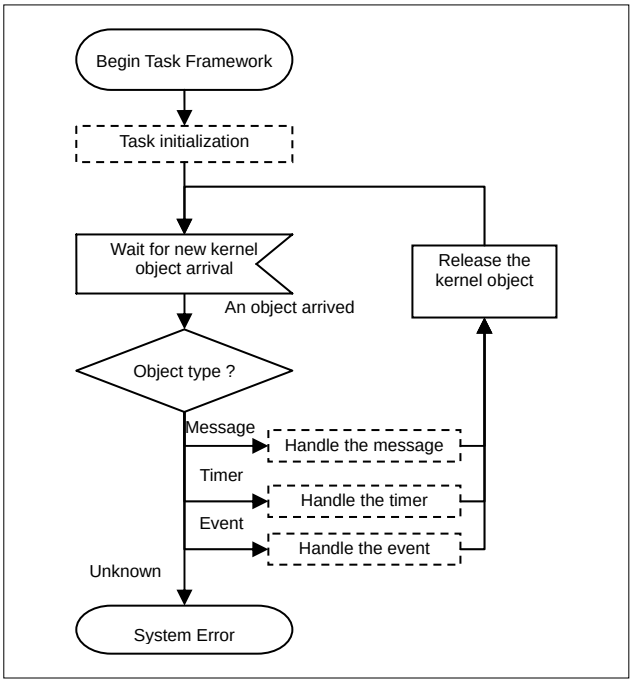

그림 7. 타스크 프레임워크 플로우

4. 타스크 프레임워크 효과 측정

타스크 프레임워크를 도입하는 경우에 대한 전체 코드 사이즈의 감소를 수식적으로 표현할 수 있다.
먼저 사용자 타스크 코드를 프레임워크로 통합할 수 있는 공통 부분과 각 타스크 별 고유 부분으로 분리하면 다음과 같은 관계가 성립한다.

$$size(T_n) = size(T_{common} + T_{n-specific})$$

따라서 N 개의 타스크에 대한 전체 코드 사이즈는 다음과 같다.

$$TotalSize_{no-framework} = \sum_{n=1}^{N} size(T_{common} + T_{n-specific})$$

$$= N \times size(T_{common}) + \sum_{n=1}^{N} size(T_{n-specific})$$

프레임워크 자체의 코드의 사이즈는 타스크 공통부분에 통합에 따른 약간의 오버헤드를 더한 크기를 가지게 되므로 다음과 같이 표현할 수 있다.

$$size(T_{framework}) = size(T_{common} + T_{overhead})$$

따라서 프레임워크를 사용한 경우 N 개의 타스크에 대한 코드 사이즈를 수식으로 표현하면 다음과 같다.

$$TotalSize_{framework} = size(T_{framework}) + \sum_{n=1}^{N} size(T_{n-specific})$$

$$= size(T_{common} + T_{overhead}) + \sum_{n=1}^{N} size(T_{n-specific})$$

$$= size(T_{common}) + size(T_{overhead}) + \sum_{n=1}^{N} size(T_{n-specific})$$

따라서 프레임워크 적용, 미 적용시의 전체 코드 사이즈를 나타내는 두 개의 수식을 비교해 볼 때 아래의 관계가 성립한다면 프레임워크 도입 시 전체 코드 사이즈가 감소되는 효과를 가져올 수 있다

$$(N \times size(T_{common})) \geq (size(T_{common}) + size(T_{overhead}))$$

위의 수식에서 볼 수 있듯 코드 사이즈 감소의 효과는 타스크 개수 N 이 높을수록, 프레임워크가 가지는 오버헤드가 낮을수록 좋다.

타스크 프레임워크를 이용하여 응용 프로그램을 구성한 경우에는 그렇지 않은 경우에 비해 코드 사이즈가 감소되므로 결과적으로 실행 파일 크기가 줄어들게 된다. 따라서 상대적으로 자원의 제약으로부터 자유롭지 못한 임베디드 디바이스의 경우 자원 효율적 측면에서 볼 때 타스크 프레임워크의 적용은 상당한 이점으로 작용할 수 있다.

아래의 그림 4 는 타스크의 개수가 1~6 개인 경우 코드 사이즈의 감소 효과를 실행 파일의 크기를 기준으로 측정한 것이다. 위의 수식에서 이미 밝힌 바와 같이 타스크의 개수와 절약되는 자원의 크기가 비례함을 알 수 있다.

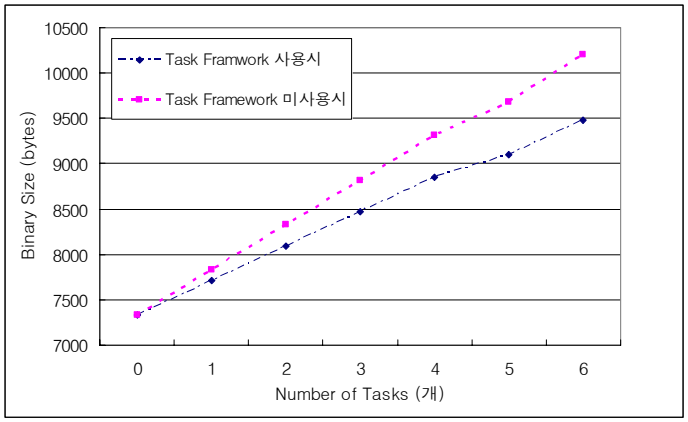

그림 8. 타스크 프레임워크 사용, 미 사용시 실행 파일의 크기 비교.

5. 결론

본 논문에서는 실시간 운영체제 상에서 사용자 어플리케이션의 최소단위, 즉 타스크의 실행환경을 제공해 주는 타스크 프레임워크의 구현 과정과 이의 도입에 따른 효과를 제시하고 결과를 측정하였다.

타스크 프레임워크란 메시징 기반의 실시간 운영체제를 기반으로 운영체제가 제공하는 커널 API 를 이용하여 사용자가 이용하기 편리한 구조화된 틀을 제공하는 것으로써 사용자는 스케줄링과 관련된 커널 API 에 대해 자세히 알 필요가 없고 단지 타스크로 전달되는 이벤트, 메시지, 타이머 각각에 대한 처리 루틴만을 따로 구현하고 이를 타스크 프레임워크에 바인딩 함으로써 어플리케이션이 목적하는 바를 이룰 수 있음을 보였다.

타스크 프레임워크를 도입함에 따라 사용자의 어플리케이션을 구조화 할 수 있을 뿐만 아니라 실질적으로 전체 코드 사이즈를 감소시키는 효과도 가져올 수 있다. 각각의 어플리케이션이 공통으로 가지고 있는 스케줄링 관련 루틴을 타스크 프레임워크로 통합하여 모든 타스크가 이를 공유하는 구조를 가짐으로써 시스템에 존재하는 타스크의 개수에 비례하여 바이너리 사이즈의 감소 효과는 커지게 된다. 일반적으로 선점형 실시간 운영체제를 이용하는 임베디드 시스템의 경우 시스템에 상주하는 타스크의 개수는 하나 이상인 경우가 많으므로 이러한 종류의 임베디드 시스템에서 타스크 프레임워크를 도입하는 것은 의미를 가진다 할 수 있다.

6. 참고문헌

[1] Dave Keefe, Phoebe Farag & Andy Zucker. Annotated Bibliography of Ubiquitous Computing Evaluations. Ubiquitous Computing Evaluation Consortium.

[2] T Kindberg, A Fox. System software for ubiquitous computing. Pervasive Computing, IEEE, 2002.

[3] http://www.faqs.org/faqs/realtime-computing/list/

[4] Balarin, F. Lavagno, L. Murthy, P. Sangiovanni-Vincentelli, A. Systems, C.D. Sangiovanni-, A. Scheduling for embedded real-time systems. IEEE Design & Test of Computers, 71-82, Jan-Mar 1998.

[5] A. Silberschatz, P.B. Galvin, G. Gagne, Operating systems principles, 7th edition, Wiley, 2006