技術論文

디바이스 데이터 입출력에 있어서 폴링 방식과 인터럽트 구동 방식의 데이터 처리 방법

구철회*

Method of data processing through polling and interrupt driven I/O on device data

Cheol-Hea Koo*

ABSTRACT

The methods that are used for receiving data from attached devices under real-time preemptive multi-task operating system (OS) by general processors can be categorized as polling and interrupt driven. The technical approach to these methods may be different due to the application specific scheduling policy of the OS and the programming architecture of the flight software. It is one of the most important requirements on the development of the flight software to process the data received from satellite subsystems or components with the exact timeliness and accuracy. This paper presents the analysis of the I/O method of device related scheduling mechanism and the reliable data I/O methods between processor and devices.

초 록

실시간 선점형 다중 태스크 운영체제를 기반으로 구동하는 프로세서와 연결된 디바이스로부터 데이터를 입수하는 방법은 크게 폴링(Polling)과 인터럽트 구동(Interrupt driven) 방식으로 구분할 수 있다. 이들 모두에 대한 기술적인 접근은 운영체제의 스케줄링 정책 및 소프트웨어 아키텍쳐에 따라 달라질 수 있다. 위성 컴퓨팅 환경에서 위성 서브 시스템 또는 컴포넌트로부터 입수되는 데이터의 처리시 시간 준수와 정확성을 보장하는 것은 비행 소프트웨어를 개발시마다 요구되는 중요한 요구사항 중의 하나이다. 본 논문에서는 디바이스의 입출력 방식과 스케줄링과의 관계에 대한 분석 및 이에 따른 프로세서와 디바이스간의 신뢰적인 데이터 입출력 방법을 제안한다.

Key Words : Polling(폴링), Interrupt driven(인터럽트 구동), I/O(입출력), Semaphore(세마포), Ada, Task(태스크), 병행성(Concurrency), Scheduling(스케줄링)

Ⅰ. 서 론

위성의 프로세서는 위성체(Spacecraft)의 상태를 제어 가능한 상태로 유지하기 위해 센서 및 자이로로부터 자세 데이터, 열센서로부터 온도 데이터, 배터리 센서로부터 전압, 전류, 압력 데이터 등을 주변 위성체 컴포넌트로부터 수집한다. 이렇게 수집되고 프로세서에 전달된 데이터들은 각각의 데이터를 처리하는 태스크들과 결합되어 위성체의 적절한 제어 동작이 가능하게 한다. 프로세서에 연결된 디바이스의 데이터를 프

† 2005년 3월 28일 접수 ~ 2005년 8월 2일 심사완료
* 정회원, 한국항공우주연구원 통신해양기상위성사업단 체계종합그룹
연락저자, Email: chkoo@kari.re.kr
대전시 유성구 어은동 45번지

로세서의 소프트웨어 로직이 데이터를 입수하기 위해, 프로세서는 위성체 컴포넌트(이하 디바이스)에게 데이터 전송 요청을 보내고 디바이스는 이에 대한 응답으로 전송 수락 신호를 전송하는 방식을 사용하거나 디바이스가 먼저 프로세서에 전송 요청을 보내고 프로세서가 전송 수락을 함으로써 데이터 전송을 할 수도 있다. 이와 같은 제어신호를 통해 디바이스의 데이터는 프로세서로 전송이 가능해 진다.

위성 시스템과 같은 실시간 처리를 필요로 하는 시스템에서는 데이터의 정확성과 더불어 데이터 처리 시간이 성능의 주요 파라미터가 된다. 일반적인 실시간 운영체제에서 프로세서는 제어 로직을 수행하는 태스크와 디바이스간의 데이터 입출력을 담당하는 태스크 등 동시에 다수의 태스크를 수행하고 있다. 그러므로 특히 선점형 다중 태스크(Preemptive multi-task) 운영체제에서는 데이터에 대한 상호 배제(Mutual Exclusion)를 통해 무결성(Integrity)을 보장하는 것이 매우 중요하다. 각각 다른 실행주기를 가지고 있는 태스크들이 각각 다른 우선순위에 의해서 수행 순서가 결정되기 때문에, 정확한 태스크들의 수행 순서를 예측하는 것은 어렵다. 이중 데이터 프로세싱은 가장 시간이 많이 걸리고 작업에 걸리는 시간을 예측하는 것이 매우 어렵다. 따라서 비효율적인 데이터 처리 로직은 실시간 시스템의 전체 성능을 떨어뜨리는 결과를 낳는다. 본 논문에서는 위성내에서 발생하는 데이터를 효과적이고 안정적으로 디바이스에서 프로세서로 데이터를 전송하는 방법에 대한 연구 내용을 기술할 것이다.

II. 폴링 방식

폴링은 데이터를 처리할 태스크가 디바이스로부터 가져올 새로운 데이터가 디바이스에 존재하는지 주기적으로 확인하는 방식이다. 이 방식은 디바이스에 데이터가 들어오는 시점과 태스크가 확인하는 시점이 다르고, 그 시간차에 의해서 태스크의 대기시간이 불명확해 지는 단점이 있으나 경성 실시간(Hard Real-Time) 시스템이 아닌 경우에는 사용하는데 있어서 무리가 없다. 데이터 폴링의 적용은 일반적으로 바이너리 세마포(Binary Semaphore)와 같은 실시간 운영체제 신호 유틸리티를 이용해서 이루어지며 기본 개념도는 그림 1과 같다.

2.1 동기 방식

폴링 방식에 의한 디바이스에서 프로세서로의 데이터 전송은 다양한 형태로 이루어질 수 있는데 크게 동기(Synchronous) 방식과 비동기(Asynchronous) 방식으로 구분이 가능하다. 만약 태스크가 각 주어진 실행 주기마다 데이터를 처리해야만 한다고 가정했을 때, 동기 방식의 데이터 전송에서 태스크는 필요한 데이터가 입수될 때까지 기다린 후 데이터가 입수된 후 나머지 동작을 완료하게 된다. 동기 방식은 데이터 입수 로직이 단순한 반면 태스크의 수행시간이 비예측적(Non-deterministic)이 되도록 만드는 경향이 있다. 그림 2는 동기 방식에서 데이터의 입수 시점과 관련하여 태스크의 수행 시간이 어떻게 영향을 받는지 보여준다.

그림 2는 우선순위 M을 가지고 있는 태스크가 디바이스로부터 입수되는 데이터를 처리하는 동작을 보여주는데 디바이스로부터 입수되는 데이터를 기다리기 위해서 불특정한 시간 w_1, w_2을 대기한다. 따라서 동기 방식에서 태스크 수행 시간은 태스크가 디바이스로부터 데이터를 받기 위해 대기하는 시간 w_1, w_2에 따라 달라지며 이 시간들은 예측할 수 없다. 하지만 태스크의 동작 시간이 일정하지 않아도 상관없거나 엄밀한 시간 해석에 의해서 실시간성이 검증된 시스템의 경우 동기 방식은 설계가 단순하며 제어 로직을 단순하고 직관적으로 만드는데 도움을 준다. 가급적 최신의 데이터를 이용하는 것이 이익이므로, 태

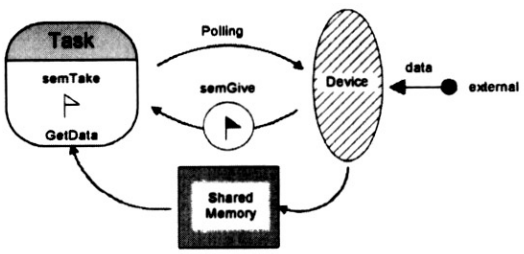

그림 1. 폴링 방식 동작 개념도

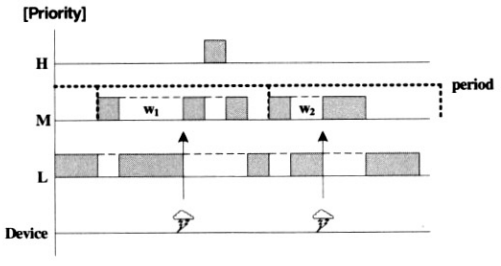

그림 2. 동기방식에서의 태스크 동작도

스크의 수행시간이 원래 설정되어 있는 주기(또는 데드라인)보다 짧다는 것이 보장되면 큰 문제는 없다[1].

동기 방식에서 태스크는 디바이스로부터 세마포를 얻은 후에 비로소 데이터를 가져오는 개념이기 때문에 태스크와 디바이스 사이의 공유 메모리는 단일 버퍼로 충분하다.

2.2 비동기 방식

비동기 방식은 디바이스로부터 연속적인 데이터를 수신하거나 디바이스의 최신 데이터를 수신할 필요가 있을 때에 사용하는 데이터 접속 방식이다. 태스크는 디바이스를 폴링할 때 디바이스가 가지고 있는 데이터에 대한 정보를 가져온다. 이 정보에 따라서 태스크는 데이터를 처리할 것인지, 아니면 이 과정을 건너뛸 것인지 판단한다.

연속적인 데이터를 처리해야 하므로 태스크와 디바이스 사이의 공유 메모리는 다중 버퍼(대부분의 경우 큐(Queue)로 구성됨)로 구성된다.

Ada95 언어에서 이러한 개념은 protected 형으로 훌륭히 구현된다[7, 9]. 리스트 1은 Ada95 언어의 비동기 방식 데이터 접속 Pseudo 코드를 나타낸다.

```
protected Device is
   data : QeueuBuffer;
   index_of_data : Information;
   function NewDataCheck return Bool;
   function GetNewData;
   procedure DeviceISR;
   pragma Attach_Handler(DeviceISR,
          DeviceINT);
end protected;

protected body Device is
function body NewDataCheck return
              Information is
begin
   return index_of_data;
end NewDataCheck;

function body GetNewData is
begin
   get data from shared memory
end GetNewData;

procedure body DeviceISR is
begin
   write data to shared memory
   update index_of_data
end DeviceISR;

end device;

-- polling device in main task
if Device.NewDataCheck > 0 then
   Device.GetNewData;
else
   do something else;
end if;
```

리스트 1. 동기방식에서의 태스크 동작도

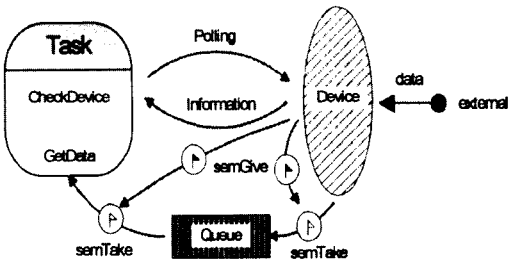

그림 3. 비동기방식의 동작 개념도

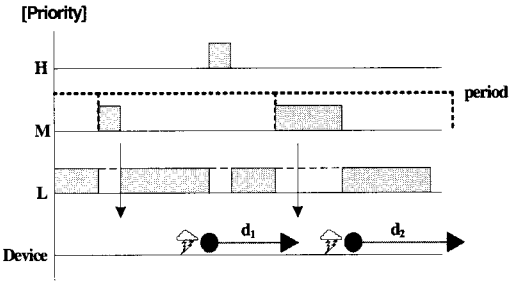

그림 4. 비동기방식에서의 태스크 동작도

Ada95 언어에서 protected 형의 내부 멤버는 상호 배제(Mutual Exclusion)가 되기 때문에 문제가 없지만 일반적인 실시간 운영체제에서는 디바이스가 데이터를 쓰는 시점과 태스크가 데이터를 읽는 시점이 중첩될 때 이들간에 상호 배제가 보장되도록 주의를 기울여야 한다. 비동기 방식의 동작 개념도는 그림 3과 같다(semGive와 semTake의 설명은 WindRiver VxWorks Prgrammer's Guide를 참고한다).

상호 배제를 보장하기 위해서 디바이스가 큐(공유 메모리)에 데이터를 쓸 때, 그리고 태스크가 큐로부터 데이터를 읽어드릴 때 세마포를 통해 그 권한을 얻도록 하여 서로 임계 구역(Critical Section)을 침범하지 않도록 설계한다.

비동기 방식의 태스크 동작도는 그림 4에 나타나 있다. 그림 2와 마찬가지로 우선순위 M인 태스크가 디바이스로부터의 데이터를 처리한다.

그림 2와 4를 비교하면 알수 있듯이 동기 방식에서는 태스크의 수행 지연이 비예측적이었지만, 그림 4와 같이 비동기 방식에서는 데이터가 생성되어 실제로 이용되는 걸리는 시간 d_1, d_2가 비예측적이다. 당연하지만 디바이스에서 데이터가 생성되는 즉시 태스크에서 그 데이터를 입력받아 처리하는 것이 이상적이다. 특히 최신의 데이터만을 디바이스로부터 받아 처리하는 태스크(예를들어, 온도 데이터, 센서 데이터, 각속도 데이터 등을 처리하는 태스크)들에서 디바이스의

데이터가 생성된 후 시간이 지나면 지날수록 그 데이터가 지닌 '계산적 값어치(Computational Value Weight)' 는 점점 작아진다. 만약 계산적 값어치가 극도로 작아진 데이터가 사용될 경우 태스크의 정상적인 동작에 심각한 지장을 초래할 수 있다. 이러한 데이터의 변질을 근본적으로 해결할 수 있는 방식은 인터럽트 구동에 따른 데이터 입수이다. 그러나 정확하고 효과적으로 구성된다면 동기방식 폴링 방식 또는 비동기 방식 폴링 방식은 실시간 시스템에서 데이터의 수신은 시스템의 실시간 특성을 만족시키며, 데이터 입출력 로직을 단순화하고 시스템의 이해가 보다 쉬워지는 장점이 있다.

III. 인터럽트 구동 방식

인터럽트 방식은 디바이스가 데이터를 가졌을 때 태스크에 인터럽트를 통해서 데이터를 전달하는 방식이다. 디바이스와 관련된 데이터 입출력에서 인터럽트 방식은 효율적이지만 시스템에서 인터럽트 신호선은 유한한 자원이며 상호 배제 로직을 복잡하게 하는 등 완전하고 안전한 성능을 보장하기 위해서는 여러 가지 고려가 요구된다. 하지만 폴링 방식을 사용하면 발생하는 단점들, 즉 과도하게 디바이스를 체크하면서 소모되는 프로세서 자원의 낭비, 또는 느슨하게 디바이스를 체크하면서 발생하는 중요한 데이터의 손실은 심각한 시스템의 성능상의 문제를 야기하기도 하므로 인터럽트 구동방식은 적절한 대안이 될 수 있다.

프로세서에 의한 인터럽트 처리는 최상위 우선순위를 가지고 있다. 일반적으로 인터럽트 처리는 모든 소프트웨어 수행에 대해서 가장 높은 우선순위를 가지고 수행되므로 태스크 수행 중간에 인터럽트를 지연시킬 방법이 없다. 물론 태스크 내에서 인터럽트를 금지하는 것은 가능하나 이것은 인터럽트 지연을 일으켜서 전혀 다른 시간 영역의 문제점을 야기할 수 있으므로 매우 특수한 경우를 제외하고는 사용해서는 안된다.

또한 인터럽트 우선순위가 있어서 실행중인 인터럽트도 우선순위가 높은 다른 인터럽트에 의해서 선점당할 수 있는 경우도 발생하기 때문에 인터럽트 로직을 수정하는 것은 바람직하지 못하다[9]. 모든 문제는 운영체제 내에서 소프트웨어적으로 해결하는 접근이 맞는다고 볼수 있다.

3.1 단순 공유 메모리를 이용하는 인터럽트 구동 방식

인터럽트 구동 방식은 그림 5와 같이 디바이스가 데이터가 준비되면 프로세서에 인터럽트를

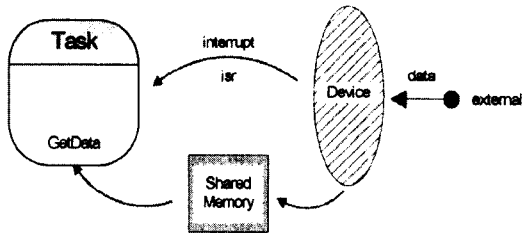

그림 5. 인터럽트 구동 기본 개념도

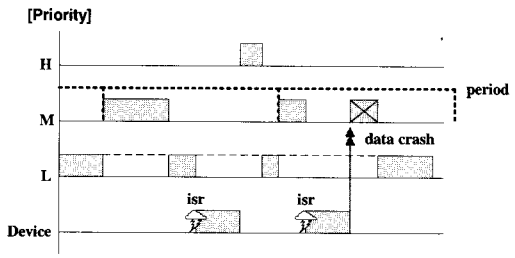

그림 6. 일반 인터럽트 구동 태스크 동작도

요청하여 디바이스가 직접 공유 메모리에 데이터를 기록한다. 이후 태스크는 공유 메모리에 업데이트된 데이터를 가져와 처리를 하는 방식이다.

그러나 이 개념은 실제 응용에 적용하기에는 몇가지 문제점을 내포하고 있다. 그중 하나는 태스크가 데이터를 사용하는 중에 디바이스가 공유 메모리의 데이터를 업데이트할 수 있기 때문에 데이터 무결성이 손상되는 경우이다. 그림 6은 이러한 문제점을 보여준다.

원칙적으로 데이터의 무결성이 보장되기 위해서는 모든 데이터의 읽기/쓰기 동작이 순차적으로 이루어져야 한다. 데이터 읽고 쓰기의 중첩은 공유 자원의 단위 동작 규칙(Atomic Action Rule)에 위배되기 때문이다[2,8]. 아무리 많은 양의 데이터라도 한번에 처리가 되어야 한다면 하나의 정수 연산과 같은 개념으로 처리가 되어야 한다[10]. 이 방식에서 데이터 무결성을 체크하기 위해서 CRC(Cyclic Redundancy Check)와 같은 체크섬(Checksum)이 평가되어야 하기 때문에 런타임(Run-time) 오버헤드(Overhead)가 중대한 문제가 될 수도 있다.

3.2 더블 버퍼링을 사용하는 인터럽트 구동 방식

태스크의 수행중에 발생하는 인터럽트에 의한 문제를 해결하기 위해서 고려된 것이 더블 버퍼링(Double Buffering)이다. 이 방식은 많이 사용되며 또한 매우 잘 동작한다[5].

많이 알려져 있는 바와 같이 더블 버퍼링은

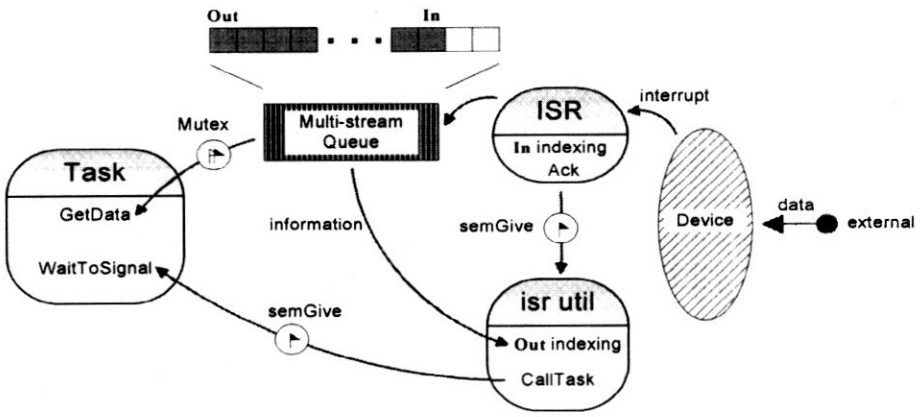

그림 7. 멀티 스트림 인터럽트 구동 개념도

데이터를 수신 또는 송신하는 버퍼를 이중화하여 한쪽이 처리되고 있는 동안에는 다른 한쪽의 버퍼를 사용함으로써 상호 배제를 가능하게 하는 방식이다. 그러나 더블 버퍼링은 표 1과 같이 한정된 조건하에서만 효과적인 사용이 가능하다.

표 1. 더블 버퍼링 사용 가능 환경

	조 건
버퍼 사이즈	더블 버퍼의 사이즈는 어느 한쪽이 처리되는 동안 다른 한쪽이 인터럽트에 의해서 '한번에' 처리가능 하도록 결정되어야 한다. 버퍼 사이즈가 과도하게 커지면 인터럽트 지연 시간이 문제가 될 수도 있다.
처리 주기	인터럽트 처리 주기는 소프트웨어 처리 주기와 원칙적으로 같아야 한다.
데이터 종속	데이터의 공급자와 수요자는 원칙적으로 1:1이어야 한다.

더블 버퍼링은 구현 개념이 간단하고 효과적으로 잘 작동하기 때문에 많이 사용되며 트리플 버퍼링(Tripple Buffering) 방식으로도 사용된다.

3.3 멀티 스트림 인터럽트 구동 방식

인터럽트 구동 방식에 있어서 가장 주의를 기울여야 하는 문제는 디바이스의 데이터 프로세싱과 태스크의 데이터 프로세싱사이의 상호 배제이다[5,9].

여러 응용적인 측면에서 사용가능하고 상호 배제를 보장하면서 인터럽트 구동 방식의 성능을 최대로 활용하기 위해서 멀티 스트림(Multi-stream) 인터럽트 구동 방식을 제안한다. 그림 7은 멀티 스트림 인터럽트 구동 방식의 구동 개념을 나타낸다.

인터럽트 구동 방식의 최대 문제점은 인터럽트가 소프트웨어적으로 스케줄링이 가능하지 않다는 것이다. 인터럽트의 출현 시점을 예측하는 것은 불가능하기 때문에 스케줄링이 가능하지 않다. 그리고 인터럽트 핸들러(Interrupt Service Routine, ISR) 동작은 지연되어서는 안되기 때문에 인터럽트와 현재 구동중인 태스크의 데이터는 직접적인 관련을 가지면 곤란하다[11]. 멀티 스트림 인터럽트 구동 방식은 상호 배제를 체크하는 'isr util' 를 가지고, 데이터 수신 버퍼를 큐(Queue) 방식으로 구성한다.

디바이스의 인터럽트에 의한 ISR은 멀티 스트림 큐 버퍼에 데이터를 저장한다. 멀티 스트림 큐 버퍼는 인덱스(In, Out)에 의해서 데이터 저장이 이루어지는데 Out 인덱스는 ISR에 의해서만 변경되는 것으로 새로운 데이터가 입수된 위치를 가리킨다. In 인덱스는 'isr util' 에 의해서만 변경되고 현재 태스크가 사용할, 또는 사용하는 데이터의 위치를 가리킨다. In, Out 인덱스의 변경이 각기 다른 블록에서 변경되므로 In, Out 인덱스는 상호 배제가 보장된다. 멀티 스트림 큐 버퍼는 링-큐(Ring Queue) 버퍼로 구성되는 것이 이상적이다.

ISR은 멀티 스트림 큐 버퍼에 데이터를 저장한 후 'isr util' 에 새로운 데이터가 도착했음을 알리는 신호를 바이너리 세마포를 통해 전송하고 ISR의 동작을 완료한다. 이상으로 ISR의 동작은 어떤 지연이 발생하는 소프트웨어 구성요소도 포함되어 있지 않다.

'isr util' 은 새로운 데이터가 도착했음을

표 2. 각 구동 방식의 비교

		장 점	단 점	비 고
폴링	동기 방식	·가장 최신의 데이터를 사용할 수 있음 ·구현하기 쉬움	·태스크의 수행시간이 비예측적임	·실시간 시스템에서 잘 사용되지 않음
	비동기 방식	·구현하기 가장 쉬움 ·태스크의 수행시간의 변화가 적음	·최신의 데이터를 사용하는 것은 보장하기 어려움	·다목적위성 1호를 비롯한 대부분의 위성에서 사용 ·정적 스케줄링에서 사용가능
인터럽트	공유 메모리를 사용	·구현하기 매우 쉬움	·데이터 무결성이 손상될 확률이 매우 높음 ·런타임 오버헤드가 많아 질 수 있는 방식	·실시간 시스템에서 잘 사용되지 않음
	더블 버퍼링을 사용	·디바이스와 태스크가 1:1로 구성되어 있을때 잘 작동함	·사용할 수 있는 조건과 환경이 다양하지 않음	·위성의 명령 및 텔레메트리 전송시 사용
	멀티 스트림을 사용	·디바이스와의 인터페이스가 소프트웨어 모듈처럼 단순화됨 ·인터럽트를 사용하면서도 폴링 방식의 동기 및 비동기 방식처럼 사용 가능	·데이터 처리 오버헤드가 다른 방식에 비해 높음(단, 오버헤드는 일정하게 유지됨)	·동적, 정적 스케줄링에서 모두 사용 가능

ISR의 신호로부터 인지하고, 이 데이터를 사용하는 태스크를 구동한다. 태스크가 이미 수행중이면 현재의 데이터는 태스크의 수행이 종료된 후 곧바로 재개되는 태스크의 수행에 의해서 사용되어진다. 뮤텍스(Mutex)를 사용하여 멀티 스트림 큐 버퍼에 접근하는 것은 멀티 스트림 큐 버퍼의 데이터를 여러 태스크에서 사용할 경우가 있을 경우 상호 배제를 보장하기 위한 것이다. 만약 한개의 태스크에서만 데이터를 사용한다면 뮤텍스의 사용은 필요치 않다.

그림 8에는 멀티 스트림 인터럽트 구동 방식을 사용했을 때의 태스크 동작을 보였다.

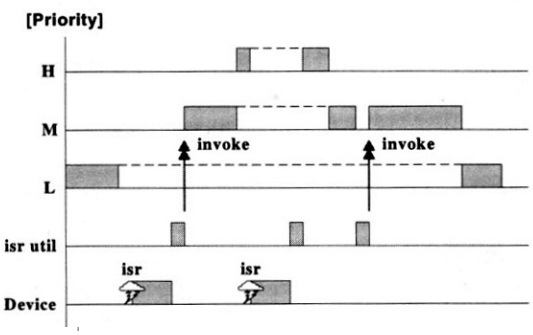

그림 8. 멀티 스트림 인터럽트 구동 태스크 동작도

그림 8과 같이 디바이스의 ISR에 의해 데이터가 최신 데이터로 업데이트되고, 더 나아가 디바이스의 ISR에 의해서 동기화되어 수행되는 태스크는 선점가능하므로 실시간 시스템의 성능에 지장을 주지 않는다. 무엇보다 그림 8에서 알수 있듯이 디바이스와 우선순위 M 태스크 사이에 'isr util' 이 존재하여 하드웨어와 소프트웨어 사이의 중간 레이어 역할을 하기 때문에 디바이스는 소프트웨어와 전혀 별개로 동작하는 것이 가능하다. 또한 디바이스의 ISR에 의해서 동기화되므로 데이터의 계산적 값어치가 가장 높을 때 태스크를 수행할 수 있다는 장점이 있다. 그리고 이 개념을 바탕으로 여러 시스템 구성에서 적용이 가능할 것이다.

Ⅳ. 결 론

본 논문에서는 위성용 컴퓨팅 환경에서 어떤 데이터 입출력 방식이 효율적인가 대해서 논술하였고, 폴링과 인터럽트 구동 방식의 적용방법 및 장단점에 대한 연구를 수행하였다.

폴링 방식은 데이터 처리의 접근 방법에서 단순성 및 용이성을 제공하는 반면 데이터의 계산적 값어치가 최상일 때 사용되지 못하는 단점이 있다. 이에 비해 인터럽트 구동 방식은 데이터를

계산적 값어치가 높은 상태에서 효율적으로 이용하는 이점은 있으나 실시간 시스템의 비예측성을 심화시키고 상호 배제와 우선순위 역전과 같은 실시간 시스템의 중요 설계 과제들을 해결하는 것이 어려운 문제점을 내포하고 있다.

폴링 방식은 어떠한 태스크 스케줄링 정책에서도 사용가능하지만, 인터럽트 구동 방식은 상호 배제를 제공하는 우선순위 기반의 선점형 다중 태스크 스케줄링 정책하에서만 사용가능하다는 제약이 있음은 주지하여야 할 사항이다[4,6].

위성 컴퓨팅 환경에서 폴링 방식은 매우 효과적으로 동작하고 있다. 그것은 입력되는 데이터의 계산적 값어치가 높지 않아도 위성 제어 로직이 위성을 제어하는데 크게 무리가 없기 때문이다.

폴링 방식은 위성 로컬 시스템과 같이 각 센서 데이터, 자이로 데이터 등이 꼭 연속적으로 보다는 일정한 주기를 가지고 입수되어도 충분한 실시간 데이터 처리 시스템에서 효과적이고 성공적으로 동작한다.

반면 인터럽트 구동 방식은 지상 관제국의 업링크 명령 데이터와 같이 순차적으로 일정하지 않은 크기의 데이터가 입력되는 경우에 적용하면 매우 효과적이다. 멀티 스트림 인터럽트 구동 방식은 효과적으로 실시간 특성을 만족시키면서 데이터 프로세싱의 성능도 우수한 것으로 판단된다.

본 논문에서 제시한 각 데이터 처리 방식의 장단점 비교 결과를 추후 실시간 데이터 프로세싱 시스템 개발시 활용할 수 있을 것이다.

후 기

본 논문은 과학기술부 "통신해양기상위성 1호 시스템 및 본체 개발사업" 의 일부임을 밝히며 연구지원에 감사를 드립니다.

참고문헌

1) Giorgio Buttazzo, "Real-Time Issues in Advanced Robotics Applications", Pisa, Italy.

2) Alan Burns, Andy Wellings, "Real-Time Systems and Programming Languages", 2nd Ed.", ADDISON-WESLEY, 1997, pp. 400~432.

3) A. Burns, A. J. Wellings, "Engineering a Hard Real-time System: From Theory to Practice", SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 25(7), pp. 705-726, 1995.

4) D. Kalinsky, "A Survey of Task Schedulers", Whitepaper, http://www.kalinsky-associates.com.

5) 구철회, 김중표, 최재동, "위성 원격측정명령 처리기의 성능검증모델 개발", 04년도 한국항공우주학회 추계학술대회 논문집, p. 233.

6) 구철회, "실시간 소프트웨어 개발기술 동향", 항공우주산업기술동향 2권1호, pp. 86 - 93, 2004.

7) Norman H. Cohen, "ADA AS A SECOND LANGUAGE, 2nd Ed.", The McGraw-Hill Companies, Inc, 1996, pp 972~979.

8) Ken Tindell, "Analysis of Hard Real-Time Communications", University of York, England.

9) M. Ben-Ari, "Principles of Concurrent and Distributed Programming", Prentice Hall International Series in Computer Science, 1990, pp 164~175.

10) Narain Gehani, Andrew D. McGettrick, "Concurrent Programming", International Computer Science Series, 1988, pp. 312~318.

11) David E. Simon, "An Embedded Software Primer", Addison Wesley, 1999, pp 199~207.