https://doi.org/10.7840/kics.2016.41.12.1884

드론을 위한 이식성과 확장성을 지원하는 ARINC 653

김 주 호◆, 조 현 철*, 진 현 욱°, 이 상 일**

Portable and Extensible ARINC 653 for Drones

Jooho Kim◆, Hyun-Chul Jo*, Hyun-Wook Jin°, Sangil Lee**

요 약

민간 드론의 활용범위가 취미, 영화촬영, 시설감시 등과 같이 다양해짐에 따라서 응용 분야의 요구사항에 맞게 소프트웨어를 안정적으로 재구성할 수 있는 기술에 대한 요구가 높아지고 있다. 항공전자 시스템의 소프트웨어 통합을 안정적으로 제공하기 위해서 ARINC 653 표준이 제안되어 현재 유인 항공기를 중심으로 적용되고 있다. 따라서 ARINC 653을 민간 드론에도 활용하는 것을 고려할 수 있다. 하지만 지금까지 ARINC 653을 구현하기 위한 다양한 연구가 진행되었으나, 다양한 플랫폼을 사용하고 응용 분야가 넓은 민간 드론에 적용되기 위해서는 추가로 고려되어야 하는 요구사항들이 존재한다. 본 논문에서는 이러한 사항들을 고려해서 이식성과 확장성이 높은 ARINC 653을 구현하고 그 성능을 분석한다. 이식성을 위해 OS 추상화 계층을 제공하여 운영체제에 대한 의존성을 낮추고 파티션 스케줄러 등의 기능을 확장할 수 있는 구조를 제공한다.

Key Words : ARINC 653, Drone, Integrated Modular Avionics, Partitioning, Software Platform

ABSTRACT

With the various usage of civil drones, such as hobby, filmmaking and surveillance, the need for technology that safely reconstructs software for target application domains has been increasingly rising. In order to support a reliable software integration of avionic systems, the ARINC 653 standard has been proposed and adapted mainly on manned aircrafts. Therefore, applying ARINC 653 on civil drones could be desirable. Though, various researches on implementing ARINC 653 has been conducted, there are still additional requirements to apply ARINC 653 to civil drones that use various platforms and have a wide range of use. In this paper, taking account of these requirements, we implement a portable and extensible ARINC 653 and analyze its performance. We offer the portability with the OS abstraction layer that reduces dependency on a specific operating system, and provide the design that can extend internal functions, such as partition scheduler and process scheduler.

Ⅰ. 서 론

민간 드론의 활용범위가 취미, 영화촬영, 시설감시 등과 같이 다양해짐에 따라서 응용 분야의 요구사항에 맞게 소프트웨어를 안정적으로 재구성할 수 있는 기술에 대한 요구가 높아지고 있다. 항공전자 시스템의 소프트웨어 통합을 안정적으로 제공하기 위해서 ARINC 653 표준이 제안되어 현재 유인 항공기를 중심으로 적용되고 있다[1-3]. 따라서 ARINC 653을 민간 드론에도 활용하는 것을 고려할 수 있다. ARINC 653

※ 본 연구는 민군겸용기술사업(Dual Use Technology Program) 지원을 받아 수행 되었습니다 (UM13018RD1).
◆ First Author : Konkuk University, joohokim@konkuk.ac.kr, 학생회원
° Corresponding Author : Konkuk University, jinh@konkuk.ac.kr, 정회원
* Konkuk University, hcjo@konkuk.ac.kr
** Agency for Defense Development
논문번호 : KICS2016-08-190, Received August 10, 2016; Revised November 3, 2016; Accepted December 13, 2016

은 하나의 응용프로그램에 대하여 독립적인 CPU 및 메모리 자원 사용량을 보장함으로써 응용프로그램 사이의 시간적, 공간적 격리(파티셔닝)를 제공한다. 이러한 자원 파티셔닝은 소프트웨어 테스트 및 통합의 용이성을 제공하고 재사용성을 극대화할 수 있으며, 하나의 컴퓨팅 노드에서 다수의 응용프로그램을 운영할 수 있기 때문에 크기, 중량, 전력 (SWaP; Size, Weight and Power) 문제를 효율적으로 해결할 수 있다.

지금까지 ARINC 653을 구현하기 위한 다양한 연구가 진행되었으나[4-7], 다양한 플랫폼을 사용하고 응용 분야가 넓은 민간 드론에 적용되기 위해서는 추가로 고려되어야 하는 요구사항들이 존재한다. 우선 다양한 하드웨어 플랫폼과 그에 따른 여러 종류의 운영체제를 지원할 수 있어야 한다. 범용운영체제(예, 리눅스, 윈도우)와 RTOS(Real-Time Operating System)는 실행환경과 프로그래밍 인터페이스가 상당히 다르기 때문에 한 쪽에 구현된 ARINC 653을 다른 쪽에 적용하기는 쉽지 않다. 또한 응용 분야마다 실시간성 요구사항(예, 스케줄링 알고리즘)이 다를 수 있기 때문에 그에 맞게 시스템 소프트웨어의 기능을 재구성할 수 있는 구조를 제공해야 한다.

본 논문에서는 이러한 사항들을 고려해서 이식성과 확장성이 높은 ARINC 653을 구현하고, 그 성능을 분석한다. 다양한 운영체제에 대한 이식성을 높이기 위해서 제안된 ARINC 653 구현은 OS 추상화 계층을 정의한다. 해당 계층은 여러 가지 범용운영체제와 RTOS를 수용할 수 있도록 설계되었다. 또한 다양한 스케줄링 알고리즘을 플러그인할 수 있는 구조를 제공하여 응용 분야에 따른 실시간 요구사항을 효율적으로 수용할 수 있도록 하였다. 성능측정 결과, 구현된 ARINC 653은 이식성과 확장성을 제공함에도 불구하고 적은 오버헤드와 낮은 지터를 보임을 확인할 수 있다.

본 논문은 다음과 같이 구성되어 있다. 본 서론에 이어 Ⅱ장에서는 ARINC 653을 소개하고, 관련연구에 대해서 살펴본다. Ⅲ장에서는 민간 드론에 적용되기에 적합한 ARINC 653 구현 구조를 제안한다. 제안된 구조는 이식성과 확장성을 극대화할 수 있도록 설계된다. Ⅳ장에서는 구현된 ARINC 653의 성능을 측정하고, 실제 드론에 적용하기 위한 계획을 설명한다. 마지막으로 Ⅴ장에서 본 논문의 결론을 맺는다.

Ⅱ. 배경지식

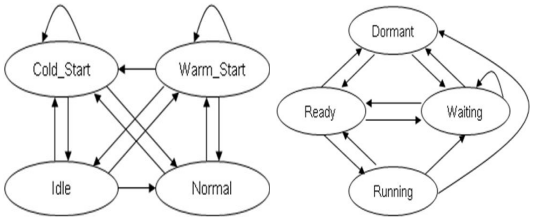
2.1 ARINC 653

통합 모듈 항공전자(IMA; Integrated Modular Avionics) 시스템은 응용프로그램 오류 및 자원 활용 측면에서 다른 응용프로그램에 영향을 주지 않고 안전하게 실행할 수 있는 환경을 제공한다. 따라서 통합 모듈 항공전자 구조는 항공전자 소프트웨어에 대한 모듈화, 이식성 그리고 재사용성을 제공하기에 유용하다.

ARINC(Aeronautical Radio, Incorporated)는 미국 소유의 비영리 단체이며, 항공, 공항, 국방, 정부, 수송 등의 5개의 분야를 주 업무 영역으로 두고 있다. LRU (Line-Replaceable Units)에 대한 표준을 확립한 단체로서 지상 기지국과 항공기 간의 통신 서비스 및 항공전자 표준규격을 정의하고 있다. 이 중 ARINC 653은 IMA를 위하여 운영체제와 응용프로그램간의 APEX (APplication/EXecutive) 인터페이스에 대한 표준을 정의한다[8].

ARINC 653은 IMA를 위하여 파티셔닝이라는 핵심적인 기능을 정의한다. 파티셔닝이란 각각의 응용프로그램들을 하나의 파티션으로 인식하고 각 파티션에 대하여 시간적 자원 분할(Temporal Partitioning)과 공간적 자원 분할(Spatial Partitioning)을 제공하는 것이다. 시간분할은 파티션에 할당된 CPU 자원을 다른 파티션에서 간섭할 수 없도록 하는 것을 의미한다. 유사하게 공간분할은 각기 다른 파티션이 서로의 물리적 메모리 자원에 영향을 끼치지 못하도록 하는 것을 말한다. 이러한 파티셔닝 개념은 항공 전자 시스템과 같이 중요한 임무를 수행하는 환경에서 하나의 응용프로그램 오류가 전체 시스템에 악영향을 미치는 것을 방지하여 높은 신뢰성을 제공할 수 있다.

ARINC 653은 파티션들이 시스템 초기화 단계에서 일괄적으로 생성되며 동적으로 생성되거나 제거될 수 없도록 정의하고 있다. 또한 파티션들은 일정한 주



(a)Partition (b) Process

그림 1. ARINC 653 파티션과 프로세스의 상태 전이도[8]
Fig. 1. State transition diagrams of ARINC 653 partition and process

기성을 가지며 사전에 정의된 XML 파일을 바탕으로 반복적으로 수행된다. 파티션들은 그림 1(a)에서와 같이 Cold_Start, Warm_Start, Idle, Normal 상태들 중 하나의 상태를 수행한다. 초기화를 할 때 파티션은 Cold_Start 상태를 갖게 된다. Warm_Start 상태는 Cold_Start 상태와 비슷하지만 실행 이미지가 메모리에 이미 적재되어 있는 상태로 전원 인터럽트가 발생한 후에 메모리에 다시 적재할 필요가 없는 상태를 나타낸다. Idle 상태는 실행되지 않은 상태를 나타내며, Normal 상태는 스케줄링 정책에 의해 파티션에 포함되는 프로세스들을 수행할 수 있는 상태를 나타낸다. 따라서 일반적으로 초기화 후 성공적으로 파티션이 구동된다면 파티션은 Normal 상태를 유지하게 된다.

하나의 파티션은 하나 혹은 그 이상의 프로세스들로 구성된다. 파티션 내에서 실행되는 각각의 프로세스는 우선순위 레벨을 가지고 있으며, 높은 우선순위의 프로세스에 의해 선점 될 수 있다. 또한 프로세스 스케줄링 정책은 각 프로세스가 생성 될 때 설정된 정책에 따라 주기적(Periodic) 또는 비주기적(Aperiodic) 프로세스가 될 수 있다. 프로세스는 그림 1(b)에서 나타낸 것처럼 Dormant, Ready, Waiting, Running 상태를 가질 수 있다. Dormant 상태는 프로세스가 시작하기 전후의 상태를 나타낸다. Ready 상태는 프로세스가 스케줄링이 가능함을 나타내는 상태이며, Waiting 상태는 I/O 대기와 같은 블록킹 상태를 나타낸다. Running 상태는 현재 프로세스가 실행중인 상태를 나타낸다.

2.2 관련연구

ARINC 653을 다양한 환경에 적용하기 위한 연구들이 많이 존재하고 있다. VanderLeest는 ARINC 653을 Xen에 적용하였으며[9], Masmano는 XtratuM에 ARINC 653을 구현하였다[10]. Han의 연구에서는 ARINC 653을 커널레벨, 가상화레벨, 유저레벨에 구현하고 이들의 성능을 비교하였다[11]. 하지만 이들 ARINC 653 구현들은 하나의 특정한 운영체제에 의존적인 구조를 가진다. ARINC 653을 특정 운영체제가 아닌 다양한 운영체제에서 지원하기 위한 연구 또한 진행되었다[11]. 하지만 이러한 연구들은 POSIX를 지원하는 운영체제에서만 ARINC 653을 지원한다.

ARINC 653을 항공기에 적용한 사례도 많이 존재하고 있다. ARINC 653 표준을 따르는 Windriver의 VxWorks 653은 보잉 787 드림라이너를 포함한 40가지 이상의 기체에서 180개 이상의 서브시스템에 적용되어 사용되고 있다[1]. GreenHills의 Integrety-178B는 미국의 전폭기인 JSF(Joint Strike Fighter)에 사용되었고[2] LynuxWorks의 LynxOS-178와 LynxOS-SE RTOS 등은 보잉 777기에 사용되었다[3]. 하지만 이러한 ARINC 653 구현들 또한 운영체제 의존적이며 대형 또는 유인 항공기 관점에서 다루고 있다.

소형 무인비행기인 드론은 대형 또는 유인 항공기보다 SWaP의 문제가 더욱 심각하기 때문에 IMA의 구조가 효과적일 수 있다[12]. 따라서, 드론에 ARINC 653을 적용하기 위한 연구들이 진행되었다[13,14]. 하지만 이들 역시 운영체제 의존적인 설계와 구현을 제시하고 있다.

최근 드론이 감시보안, 운송, 기상관측, 농업 등의 여러 산업 분야에서 각광받고 있다. 따라서 이러한 흐름에 맞춰 드론 소프트웨어에 대한 연구가 진행되었다[15,16]. 하지만 이들 연구에서는 드론의 소프트웨어 플랫폼이 아닌 드론의 제어와 영상관련 응용에 관한 내용들을 주로 다루고 있다. 또한 드론의 종류와 목적이 다양해지고 부가기능이 많아지면서 소프트웨어 기능 또한 증가함하고 있다. 본 연구에서는 드론에서 다양한 운영체제에 ARINC 653을 지원하도록 운영체제의 의존성을 제거하고 이식성과 확장 가능한 구조를 제안한다. 본 연구에서 제안하는 구조는 소량 다품종으로 생산되는 드론에서 다양한 운영체제를 기반으로 ARINC 653을 지원하고, 여러 시스템 환경에 맞게 소프트웨어를 쉽게 구현할 수 있는 장점을 가진다.

Ⅲ. 드론을 위한 소프트웨어 플랫폼

본 장에서는 이식성과 확장성을 제공하기 위한 ARINC 653 구현 구조를 제안하고, 초기화 과정과 계층적 스케줄러 및 APEX에 대하여 설명한다.

3.1 이식성과 확장성을 위한 구조

본 논문에서 제안하는 이식성과 확장성을 위한 ARINC 653 구현 구조는 그림 2와 같은 형태를 가지고 있다. 그림 2의 Configuration Data를 통해 초기화 환경 설정을 제공한다. Configuration Data에는 파티션, 프로세스, 스케줄러를 초기화하는데 필요한 정보들이 명시되어있다. Initialization Process는 Configuration Data에 저장된 내용을 기반으로 시스템을 구동하기 위한 초기화 작업을 시작한다. 더 자세한 내용은 Ⅲ.2에서 다룬다. Partition Scheduler와 Process Scheduler는 초기화 작업 후 실행된다. 이들은 다양한 스케줄링 알고리즘을 지원할 수 있도록 확장 가능하게 구현되었다. 더 자세한 내용은 Ⅲ.3에서

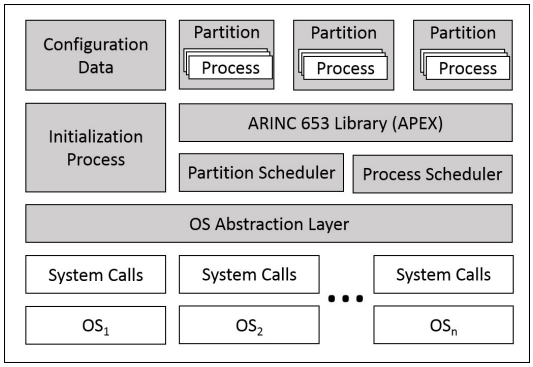


그림 2. 전체 시스템 구조
Fig. 2. System architecture

다룬다. ARINC 653 라이브러리는 ARINC 653 APEX를 구현한 라이브러리이다. 프로세스들은 운영체제에 관계없이 이 라이브러리에서 지원하는 함수를 이용하여 작성된다. 해당 내용은 Ⅲ.4에서 다룬다.

OS 추상화 계층은 대상 운영체제의 라이브러리 함수와 시스템 호출들을 추상화하여 상위 기능들을 수정없이 재사용 할 수 있도록 한다. OS 추상화 계층는 표 1과 같이 시그널, 프로세스 생성, 파일 입출력, 그리고 시간 등으로 구분된 기능을 제공한다. 대상 운영체제는 컴파일 시점에 명시하고, 설정된 운영체제에 맞는 OS 추상화 계층이 컴파일 된다.

표 1. OS 추상화 계층
Table 1. OS Abstraction Layer

Wrapper function	Description
OSAL_PartitionCreate	Creates an ARINC 653 partition
OSAL_ProcessCreate	Creates an ARINC 653 process
OSAL_SendSignal	Sends SIGNAL_STOP or SIGNAL_CONT to a specific process
OSAL_FileOpen	Opens configuration data
OSAL_FileClose	Closes configuration data
OSAL_FileRead	Reads from configuration data
OSAL_TaskGetId	Gets process identifier
OSAL_TaskDelay	Delays process
OSAL_GetTimeUsec	Gets current time in micro second
OSAL_TimerCreate	Creates interval timer
OSAL_DaemonCreate	Creates daemon process in system partition
OSAL_InterProcessComm	Used to communicate between processes

3.2 시스템 초기화

시스템 초기화는 그림 3과 같이 진행된다. Configuration Data는 텍스트파일 또는 헤더파일로 정의할 수 있다. 파일 시스템을 지원하는 운영체제에서 동작하는 경우 텍스트 파일로 작성하고 파일 시스템을 지원하지 않는 경우에는 헤더파일로 정의하였다. 이는 운영체제에서 파일시스템 지원여부에 관계없이 초기화를 진행할 수 있도록 하여 이식성을 제공한다. 각 항목에 대한 설명은 다음과 같다. 그림 3의 ①은 지원하는 파티션 스케줄러 중 사용할 스케줄러를 명시한다. ②는 파티션과 프로세스를 초기화하기 위해 사용할 메모리 할당방식을 명시한다. 메모리 할당방식이 Dynamic인 경우에는 구조와 기능을 동적할당을 통해 제공하고, Static인 경우에는 정적할당을 통해 자료구조를 생성한다. 이는 운영체제의 동적할당 지원여부에 따라서 선택할 수 있도록 하여 이식성을 높이기 위함이다. ③은 Type, ID, Period, Runtime, Deadline, Path, Policy 순서로 명시한다. Type는 P(Partition) 또는 T(Task) 중 하나로 지정된다. ID는 각 파티션과 프로세스에 임의로 부여되는 식별자이고, Period, Runtime, Deadline은 스케줄링할 때 필요한 주기, 실행시간, 데드라인을 나타낸다. Path에는 프로세스 실행파일 경로를 지정한다. 파일 시스템을 지원하지 않는 경우에는 헤더파일의 Path에 프로세스 실행 함수명을 지정한다. 파티션은 실행할 파일이 없으므로 Path에 null값을 넣는다. Policy는 프로세스의 스케줄링 정책을 지정한다. 프로세스 스케줄러에서는 Policy로 초기화된 프로세스의 스케줄링 정책에 따라서 스케줄링을 수행한다. 파티션의 경우에는 ①에서 파티션 스케줄링 정책을 명시하기 때문에 Policy를 null로 지정한다. ④는 Parser를 실행하기 위해 Configuration

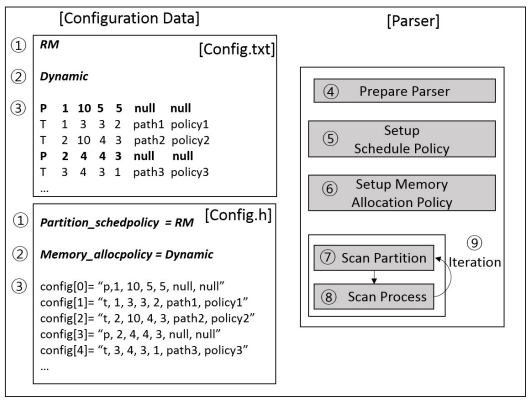


그림 3. 시스템 초기화 과정
Fig. 3. System Initialization Process

Data 텍스트파일을 열고 파일 포인터를 초기화한다. 파일 시스템이 없는 경우에 ④는 생략된다. ⑤는 Configuration Data에 명시된 파티션 스케줄러 정책으로 파티션 스케줄러를 설정한다. ⑥은 메모리 할당 방식에 따라 파티션 및 프로세스 자료구조를 배열 또는 리스트로 초기화한다. ⑦은 Type이 P인 경우 파티션을 초기화 한다. ⑧은 Type이 T인 경우 실제 프로세스를 생성한다. 생성된 프로세스는 파티션 스케줄러가 깨울 때까지 Dormant상태가 된다. ⑨는 Configuration Data의 특성 상 파티션 P 이후 다음 파티션 P가 등장 할 때까지 프로세스를 초기화 하도록 되어있다. ③에서 보면 파티션 P 이후에 나오는 태스크 T들은 그 파티션에 속하게 된다. 다음 파티션 P가 나오면 앞서 나온 파티션의 초기화가 끝난다. 그리고 다시 ⑦로 돌아가 다음 파티션 P의 초기화를 시작한다. ⑨와 같이 파일이 끝날 때까지 초기화를 반복한다.

3.3 계층적 스케줄러

파티셔닝을 지원하기 위해서는 파티션 스케줄러와 프로세스 스케줄러가 계층적으로 동작해야한다. 파티션 스케줄러는 CPU 자원을 할당할 파티션을 정한다. 프로세스 스케줄러는 파티션 스케줄러에서 스케줄된 파티션 내부의 프로세스들을 스케줄링한다.

그림 4는 Ⅲ.1의 OS 추상화 계층 라이브러리를 이용한 파티션 스케줄러의 스케줄링 동작 과정을 나타낸다. 제안된 구조는 파티션 스케줄링 기법을 설정할 수 있도록 파티션 스케줄러를 모듈화하여 확장성을 높였다. 현재 구현에서는 RM(Rate Monotonic) 스케줄링과 EDF(Earliest Deadline First) 스케줄링을 제공하며 다른 파티션 스케줄링 기법을 구현하여 추가할 수 있다. Ⅲ.2에서 명시한 바와 같이 Configuration Data를 기반으로 파티션 스케줄러를 초기화한다.

파티션 스케줄러는 Configuration Data에 명시된 스케줄링 정책을 따라 다음에 실행할 파티션을 결정한다. 파티션 스케줄러는 초기화 과정에서 각 파티션 및 파티션 내부의 프로세스들의 식별자 및 주기와 실행시간 그리고 데드라인 등의 정보를 갖는다. 이 정보들은 APEX 라이브러리를 통하여 접근 및 변경이 가능하며 이 정보들을 이용하여 파티션 스케줄러는 다음 스케줄링 될 파티션을 결정한다. 파티션 스케줄러는 그림 4의 스케줄링 과정을 반복한다.

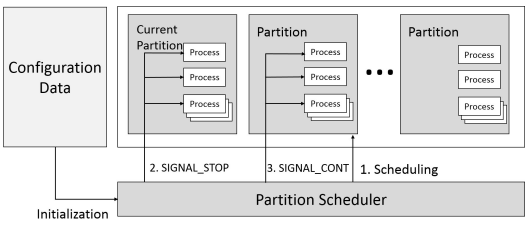


그림 4. 파티션 스케줄링 과정
Fig. 4. Partition Scheduling Process

프로세스는 할당된 스케줄링 정책에 따라 주기적인 프로세스와 비주기적인 프로세스를 가질 수 있다. 제안된 구조에서는 프로세스 스케줄링 기법을 설정할 수 있도록 모듈화하여 확장성을 높였다. 프로세스 스케줄러에는 사용자가 정의한 스케줄러를 사용할 수 있으며, 운영체제에서 제공하는 스케줄러도 사용할 수 있다. 운영체제에서 제공하는 스케줄러를 사용한다면, 예를 들어 운영체제가 리눅스인 경우 주기적인 프로세스가 비주기적인 프로세스에 우선하여 스케줄링 된다. 이러한 경우, schedulability 검사에는 편리하나, 중요도가 높은 프로세스가 비주기적인 프로세스로 할당 되면 주기적인 프로세스와 비교하여 반응성이 낮은 문제가 생길 수 있다. 사용자가 스케줄러 모듈을 작성하여 사용할 경우, 주기적인 프로세스가 비주기적인 프로세스보다 우선순위가 높아야 하는 제약을 가지지 않아, 위와 같은 문제는 발생하지 않는다. 하지만 schedulability 검사를 하기에는 더 복잡하다. 따라서 사용자가 정의한 스케줄러 또는 운영체제에서 제공하는 스케줄러를 사용할 때는 주의가 필요하다.

3.4 ARINC 653 APEX 지원

APEX는 OS 추상화계층을 사용하여 구현함으로써 운영체제에 대한 의존성을 제거하였다. 현재까지 구현된 APEX 표준 인터페이스는 대부분 표 2와 같이 파티션 및 프로세스의 정보를 저장, 접근 그리고 변경하는 데 이용된다. 이외에도 서로 다른 프로세스를 제어하는 데도 이용된다.

Ⅳ. 성능평가

본 장에서는 논문에서 제안한 ARINC 653 구현 구조를 구현하고 실험을 통해 스케줄링 성능 및 ARINC 653 APEX의 성능을 측정한다. 대상 운영체제는 Ubuntu 10.04를 사용하였고, 커널 버전은 2.6.32이다. 실험장비는 Intel Mobile 1.4Ghz 프로세서를 장착하였다.

표 2. APEX
Table 2. APEX

APEX	Description
CREATE_PROCESS	Creates a process in the partition
GET_MY_ID	Returns one process identifier
START	Starts or resumes a process from another process
STOP	Stops a process from another process
GET_PARTITION_STATUS	Gets information of partition
GET_PROCESS_STATUS	Gets information of process
SET_PARTITION_MODE	Sets the partition mode
GET_TIME	Gets the current time

4.1 스케줄링 성능

그림 5는 파티션이 전환될 때의 오버헤드를 측정한 것이다. 두 개의 파티션을 두고 각 파티션에 들어가는 프로세스의 개수를 다르게 하여 측정한 결과 프로세스의 개수가 증가할수록 오버헤드가 증가한다. 파티션이 전환될 때, 기존 파티션의 상태 값과 스케줄된 파티션의 상태 값을 변경해준다. 그리고 기존 파티션 내부의 프로세스들에 SIGNAL_STOP을 보내고 스케줄된 파티션 내부의 프로세스들에 SIGNAL_CONT를 보낸다. 파티션 내부의 프로세스 각각에 시그널을 보내기 때문에, 파티션 안의 프로세스 개수에 따라 파티션이 전환될 때 시그널을 보내는 횟수가 달라진다. 따라서 파티션 전환 시 각 파티션에 속한 프로세스의 개수가 증가함에 따라 그림 5와 같이 오버헤드가 증가한다. 하지만 그 오버헤드가 크지 않고 소형 드론에서 동작하는 파티션에 포함되는 프로세스의 개수가 많지

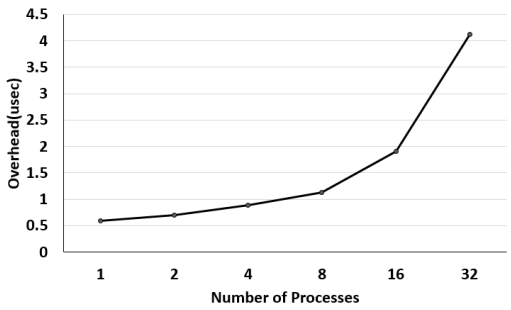


그림 5. 파티션 전환 오버헤드
Fig. 5. Partition Switching Overhead

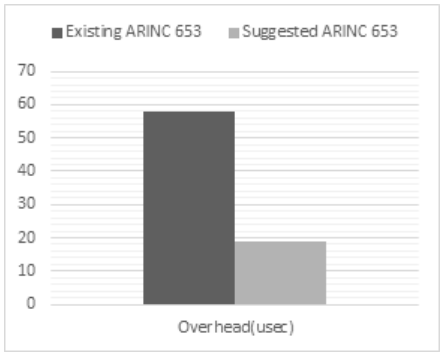


그림 6. 파티션 전환 오버헤드 비교
Fig. 6. Partition Switching Overhead Comparison

않을 것으로 예상되기 때문에 관찰된 오버헤드는 치명적이지 않다고 할 수 있다.

또한 그림 6은 기존 리눅스에 의존성을 갖는 사용자 수준에서의 ARINC 653 구현 구조[11]와 본 논문에서 제안한 구조의 파티션 전환 오버헤드를 비교한 그래프이다. 두 개의 파티션에 각각 4개의 프로세스와 1개의 프로세스를 할당하고 성능측정을 하였다. 실험 결과 기존 구조보다 3배 정도의 성능 향상을 보인다.

4.2 APEX 성능

표 3은 APEX의 실행시간의 평균과 편차를 나타낸다. 표 3에서 볼 수 있듯이 본 논문에서 구현된 ARINC 653 APEX는 이식성과 확장성을 위해 계층적 구조를 갖고 있음에도 불구하고 낮은 오버헤드를 보여주고 있다. GET_MY_ID, GET_PARTITION_STATUS, GET_PROCESS_STATUS, 그리고 SET_PARTITION_MODE는 사용자 레벨에서 저장된 자료구조에만 접근하기 때문에 낮은 오버헤드를 보인다.

표 3. APEX 오버헤드 비교
Table 3. APEX Overheads Comparison

APEX	Suggested ARINC 653		Existing ARINC 653	
	Ave. (usec)	Dev. (usec)	Ave. (usec)	Dev. (usec)
CREATE_PROCESS	36.55	14.37	6.2	1.2
GET_MY_ID	0.51	0.09	1.0	0.6
START	0.69	0.39	18.2	8.6
STOP	0.78	0.36	35.4	27.0
GET_PARTITION_STATUS	0.60	0.24	38.2	29.9
GET_PROCESS_STATUS	0.48	0.30	39.8	11.7
SET_PARTITION_MODE	0.54	0.45	37.4	27.9

그림 7. APM 드론
Fig. 7. APM Drone

CREATE_PROCESS는 OS 추상화 계층의 함수들을 연속적으로 호출하기 때문에 비교적 큰 오버헤드를 보이고 있다. 제안된 ARINC 653에서는 실제로 프로세스를 생성하는 시스템콜을 CREATE_PROCESS 내부에서 호출하는 반면 기존 논문에서는 파티션 별로 초기화 시점에 호출하고, CREATE_PROCESS에서는 단순히 자료구조를 변경하는 기능을 수행하여 성능차이가 있음을 볼 수 있다. 하지만 일반적으로 프로세스들은 초기화 단계에서 대부분 생성되기 때문에 런타임 오버헤드를 증가시키는 원인은 아니다.

Ⅴ. 결론 및 향후계획

본 연구에서는 ARINC 653을 지원하기 위한 이식성 및 확장성을 제공하는 구조를 설계 및 개발하여 운영체제에 대한 의존성 없이 구동되는 ARINC 653을 제안하였다. 또한 성능측정을 통하여 APEX의 낮은 오버헤드를 확인하였다. 향후 계획으로는 본 논문에서 제안한 구조를 소형 드론에 적용하여 비행을 수행하고자 한다. 그림 7과 같이 Elre-brain사의 보드를 이용하여 드론을 조립했고 APM(ArduPilot Mega)[17]에서 제공하는 자동비행을 위한 오픈소스 소프트웨어를 올릴 예정이다. 비행을 위해 사용된 보드의 운영체제는 Debian이며 ARM Cortex-A7 프로세서를 장착하였다. 현재는 APM에서 제공하는 오픈소스 소프트웨어를 분석하여 소프트웨어를 구성하고 있는 프로세스들을 서로의 의존도와 기능에 따라 파티셔닝을 진행하고 있다. 각 파티션이 사용하는 센서 및 디바이스의 진동수를 이용하여 파티션의 주기를 파악하고, WCET(Worst-Case Execution Time)을 측정하여 파티션의 실행시간을 분석할 예정이다. 그리고 분석된 파티션의 주기와 실행시간을 이용하여 스케줄링 가능성 또한 분석할 것이다. 그 다음 분석된 내용을 이식성과 확장성을 위한 ARINC 653에 적용하여 비행시험을 수행할 계획이다.

References

[1] Windriver, Retrieved August 8. from http://www.windriver.com.

[2] GreenHils safety critical product: INTEGRITY -178B RTOS, Retrieved August 8. from http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[3] LynuxWorks, Retrieved Aug., 8. form http://lynuxworks.com/rtos/rtos-178.php.

[4] H.-J. Park, K.-C. Go, and J.-H. Kim. "Design method for integrated modular avionics system architecture," *J. KICS*, vol. 39, no. 11, pp. 1094-1103, 2014.

[5] P. Edgar, J. Rufino, T. Schoofs, and J. Windsor, "Amoba ARINC 653 simulator for modular based space applications," *Eurospace DASIA*, Oct. 2008.

[6] T. Schoofs, S. Santos, C. Tatibana, and J. Anjos, "An integrated modular avionics development environment," in *Proc. IEEE/AIAA DASC*, Oct. 2009.

[7] A. Dubey, G. Karsai, and N. Mahadevan, "A component model for hard real-time systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517-1550, 2011.

[8] Aeronautical Radio Inc., *Avionics Application Software Standard Interface(Part 1): Require Services*, ARINC Specification 653P1-2, Dec. 2005.

[9] S. H. VanderLeest, "ARINC 653 hypervisor," in *Proc. IEEE/AIAA DASC*, Oct. 2012.

[10] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "XtratuM: A hypervisor for safety critical embedded systems," *Real-Time Linux Workshop*, Sept. 2009.

[11] S. Han and H.-W. Jin, "Resource partitioning for integrated modular avionics: comparative study of implementation alternatives," *Software: Practice and Experience*, vol. 44, no. 12, pp. 1441-1466, Dec. 2014.

[12] H.-W. Jin, S.-H. Lee, S. Han, H.-C. Jo, and D. Kim, “WiP abstract: challenges and strategies for exploiting integrated modular avionics on unmanned aerial vehicles,” in *Proc. ACM/IEEE ICCPS*, Apr. 2012.

[13] H.-C. Jo, K. Park, D. Jeon, H.-W. Jin, and D.-H. Kim, “Integrated system of multiple real-time mission software for small unmanned aerial vehicles,” *Telecommun. Rev.*, vol. 24, no. 4, pp. 468-480, Aug. 2014.

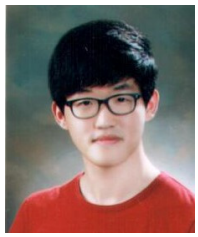
[14] H.-C. Jo, S. Han, S.-H. Lee, and H.-W. Jin, “Implementing control and mission software of UAV by exploiting open source software-based ARINC-653,” in *Proc. IEEE/AIAA DASC*, Oct. 2012.

[15] H. Shi, H. Park, H.-H. Kim, and K.-H. Park, “Vision-based trajectory tracking control system for a quadrotor-type UAV in indoor environment,” *J. KICS*, vol. 39, no. 1, pp. 47-59, 2014.

[16] W.-M. Park, J.-H. Choi, S.-G. Choi, N.-D. Hwang, and H.-C. Kim, “Real-time shooting area analysis algorithm of UAV considering three-dimensional topography,” *J. KICS*, vol. 38, no. 12, pp. 1196-1206, 2013.

[17] APM(ArduPilot Mega), Retrieved August 8. from http://ardupilot.org.

김 주 호 (Jooho Kim)



2016년 8월 : 건국대학교 컴퓨터 공학과 졸업
2016년 9월~현재 : 건국대학교 컴퓨터 공학과 석사과정
<관심분야> 실시간 시스템, 운영체제

조 현 철 (Hyun-Chul Jo)

2012년 2월 : 건국대학교 컴퓨터 공학과 졸업
2014년 2월 : 건국대학교 컴퓨터 공학과 석사
2014년 9월~현재 : 건국대학교 컴퓨터 공학과 박사과정
<관심분야> 실시간 시스템, IoT, 운영체제, 임베디드 시스템

진 현 욱 (Hyun-Wook Jin)

1997년 2월 : 고려대학교 전산학과 졸업
1999년 2월 : 고려대학교 컴퓨터학 석사
2003년 8월 : 고려대학교 통신시스템공학 박사
2003년 9월~2006년 1월 : 미국 오하이오 주립대학교 Research Associate
2006년 3월~현재 : 건국대학교 컴퓨터공학과 교수
<관심분야> 운영체제, Cyber-Physical Systems, 클라우드 컴퓨팅

이 상 일 (Sangil Lee)

1994년 2월 : 성균관대학교 정보공학과 졸업
1996년 2월 : 성균관대학교 정보공학과 석사
2010년 2월 : 성균관대학교 전기전자 및 컴퓨터공학과 박사
1996년 1월~현재 : 국방과학연구소 책임연구원
<관심분야> C4I체계, 무기체계 상호운용성, 서비스 아키텍처, 인지무선통신