탑재운영절차서 실행환경을 위한 Lua 인터프리터 기반의 가상머신 설계

강수연* 정회원, 구철회**, 주광혁***, 박시형****, 김형신****

Design of a Virtual Machine based on the Lua interpreter for the On-Board Control Procedure Execution Environment

Sooyeon Kang* *Regular Member*, Cheolhea Koo**, Gwanghyeok Ju***, Sihyeong Park****, Hyungshin Kim****

요 약

본 논문에서는 탑재운영절차서 실행환경을 위한 Lua 인터프리터 기반의 가상머신 설계와 기능 및 성능분석 결과를 나타낸다. 한국항공우주연구원에서 계획 중인 달 탐사 임무를 온보드상에서 자율적으로 운영하기 위해 탑재운영절차서 실행환경의 개발이 요구되어졌다. 탑재운영절차서는 위성에 탑재되어 지상 간섭없이 자율적으로 임무 수행을 가능케 함으로써 전파 지연과 제한된 데이터 통신용량을 갖는 심우주 임무들에서 이미 적용되고 있다. 가상머신의 실행엔진인 인터프리터는 고급언어로 작성된 원시코드를 한 줄씩 번역하고 실행하므로 컴파일러에 의해 생성된 코드가 실행되는 것에 비해서 실행 속도가 현저하게 느리다. 이를 극복하기위해 레지스터 기반의 Lua 인터프리터를 적용하여 탑재운영절차서 실행환경 설계 및 구현하였으며 실험을 통해 여러 요소들에 따른 성능분석을 수행하였다. 성능분석 결과는 탑재운영절차서 스케줄링 방안 뿐 아니라 Lua 인터프리터를 적용하는 시스템에 적용될 수 있을 것으로 기대된다.

Key Words : On-Board Control Procedure (OBCP), OBCP Execution Environment (OEE), Virtual Machine, Lua interpreter, On-Board Software (OBS)

ABSTRACT

In this paper, we present the design, functions and performance analysis of the virtual machine (VM) based on the Lua interpreter for On-Board Control Procedure Execution Environment (OEE). The development of the OEE has been required in order to operate the lunar explorer mission autonomously which is planned by Korea Aerospace Research Institute (KARI) autonomously. The concept of On-Board Control Procedure (OBCP) is already being applied to the deep space missions with a long propagation delay and a limited data transmission capacity since it ensure he autonomy of the mission without the ground intervention. The interpreter is the execution engine in the VM and it interpreters high-level programming codes line by line and executes the VM instructions. So the execution speed is very more slower than that of natively compiled codes. In order to overcome it, we design and implement OEE using register-based Lua interpreter for execution engine in OEE. We present experimental results on a range of additional hardware configurations such as usages of cache and floating point unit. We expect those to utilized to the OBCP scheduling policy and the system with Lua interpreter.

I. Introduction

In deep space missions like the lunar explorer, low SNR, and signal distortion, a high level of autonomy in the operations is required because contact time between space and ground is limited due to long propagation delay.

Spacecraft operations are traditionally performed by using Flight Control Procedures (FCPs) and Mission Time Line (MTL). FCPs are executed step-by-step by a ground operator, which involves sending Telecommands (TC)s to

*한국항공우주연구원 위성비행소프트웨어팀 (sykang@kari.re.kr)
**한국항공우주연구원 달탐사기술팀 (chkoo@kari.re.kr), ***한국항공우주연구원 달탐사연구실 (ghju@kari.re.kr),
****충남대학교 컴퓨터공학과 임베디드 시스템연구실 (sihyeong@cnu.ac.kr, hyungshin@cnu.ac.kr)
접수일자 : 2014년 12월 11일, 수정완료일자 : 2014년 12월 22일, 최종게재확정일자 : 2014년 12월 22일

the spacecraft and checking Telemetry (TM) downlinked to ground. MTL is a sequence of time-tagged TC loaded from ground and executed by the OBS, and is planned to be executed when the time tag expires. While MTL allows for autonomous on-board TC execution, the concept is limited as it consists in concept of success-oriented commanding and it is nearly impossible to react immediately to unexpected behavior such as failed TC[1]. The complexity of the OBS is increased when autonomous capability is required to cope with non-nominal situations. Although such recovery function can be applied to the OBS on orbit through reload and patch from ground, it is clearly impractical and risky for the mission due to hazard and complexity coming from the operation. Further it is one of major reasons of that cost of ground operation is hard to be saved.

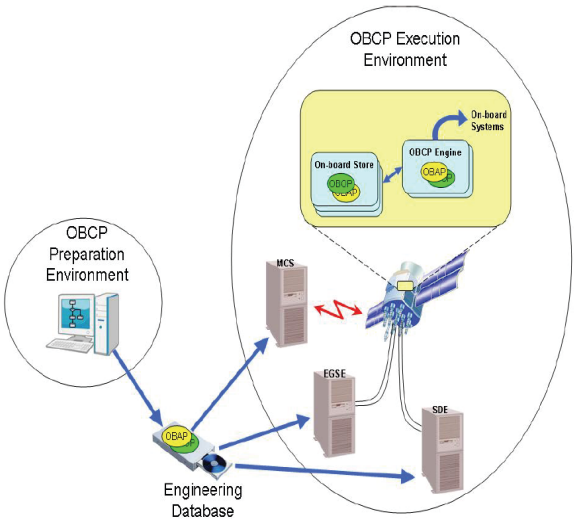

Figure 1. OBCP System [2]

As a solution to the above problems, ESA has introduces and released "Spacecraft On-Board Control Procedures" for space standardization in order to define the OBCP concept for the OBCP system that can be applied for any mission. It provides some sorts of functionalities that are useful for controlling the spacecraft through small script-like programs written in a specific language. The OBCP system consists of an OBCP preparation environment located on the ground and an OEE located on-board[2]. Figure 1 shows the OBCP system. After OBCPs are programmed, compiled on the OBCP preparation environment and uploaded on-board, they are handled by the OEE platform, which is responsible for loading, scheduling, executing, controlling and monitoring OBCPs. Particularly the OEE shall satisfy the reusability, the portability, the operability and the maintainability for the space mission systems in terms of lowering of complexity, handling un-expectation situation, performing deep space exploration, saving cost and short development period.

Therefore OBCPs can be used to implement functionality late in the project, when OBS is frozen and the modification of OBS is difficult to apply due to development schedule or cost. An amount of ground operations activities can be simplified or reduced by using OBCPs. For example a sequence uplink budget and the amount of MTL commands can be substituted by operations of OBCPs[3].

The rest of this paper is organized as follows. In section 2, we describe the related projects which utilize the OBCP systems at their mission. In section 3, we introduce the design of the VM based on the Lua interpreter for the OEE of the OBCP system in order to meet the operational autonomy requirements and capabilities for the Korea lunar mission. In section 4, we present the experimental evaluation of our designed VM. Finally, in section 5, we conclude with a summary of our work presented in this paper. Figure 1. OBCP System[2]

Ⅱ. Related Works

The first ESA mission to fly OBCP-like facilities was the European Retrievable Carrier (EURECA) in 1992. Although EURECA was a mission in the earth orbit, limited ground station contact time demanded for a significant amount of spacecraft operations have been executed autonomously by OBCP[1]. Rosetta and Venus Express missions are successfully using OBCPs. The OBCP concept for Rosetta mission is based on the successful EURECA OBCP experience. As Rosetta encounters long propagation delays, unstable communication link and low data rates throughout the mission, advanced spacecraft is utilized, and it is achieved through OBCPs. Venus Express inherits major parts of the Rosetta data handling OBS, including the OBCP facility[1][4]. OBCPs embedded in Rosetta and Venus Express mission is written in Spacecraft Control Language (SCL) which is specially designed for simplicity and safety and is suitable for mission-critical applications. SCL is based on the syntax and semantic of Ada 83, which is a structured, strongly and statically typed imperative language[5]. The Herschel/Planck and GOCE

missions support another OBCP environment based on On-Board Command Language (OCL) to define the OBCP source code. OCL is similar but equal to ANSI C[6]. The OBCP concept has been applied for Communication Ocean Meteorological Satellite (COMS) mission. COMS is the first Korea multi-mission geostationary satellite. COMS has an Interpreter Program Environment (IPE) as a part of OBS. IPR takes charge of managing, scheduling and interpreting interpreted programs (IPs) written in Application Program Language (APL)[7][8].

Table 1. OBCP technologies and capabilities in Space Missions

	Rosetta	Venus	Herschel, Plank, GOCE, Gala	COMS
Language	SCL	SCL	OCL	APL
Max.Code Size	8 Kb	4 Kb	64 Kb	16 Kw
Max.Num.of OBCPs concurrently running	20	10	16	21
Scheduling Policy	NP RR	NP RR	P&P	NP RR
NP : Non-Preemptive, RR : Round-Robin P&P : Priority based Preemptive				

Ⅲ. Related The virtual Machine for the OEE

OBS is composed of Basic Software (BSW), Application Software (ASW) and OEE. Figure 2 presents the general OBS architecture for OBCP.

OBS runs on the real hardware target processor (e.g. SPARC, PowerPC, etc.) with real-time operating system. OEE is a VM for supporting and managing OBCP functionalities. It provides the execution environment which is responsible for loading, scheduling, executing, controlling and monitoring OBCPs. OEE has three functions, which are OBCP manager, OBCP scheduler and OBCP interpreter.

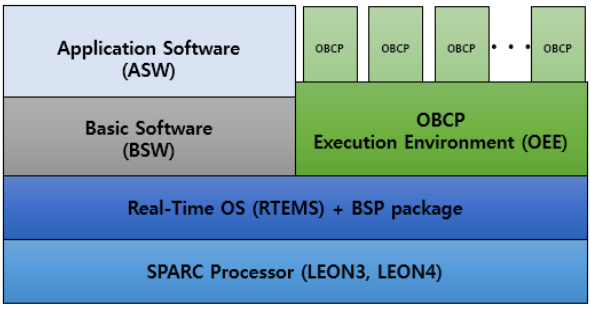

Figure 2. OBS Architecture for OBCP

The context in which the OBCP operates is shown in Figure 3. TCs and events from ground and other functions of OBS to OEE and a specific OBCP are processed by OBCP manager. OBCPs are executed by OBCP scheduler which is activated by scheduling tasks according to activation levels. The run-time library (RTL), which provides a set of functions mainly acting as interface to the BSW. These functions are used as library function in order to extend the Lua language. RTL is made of interface library, data access library, message library, communication interface library between OBCP and devices and error library.

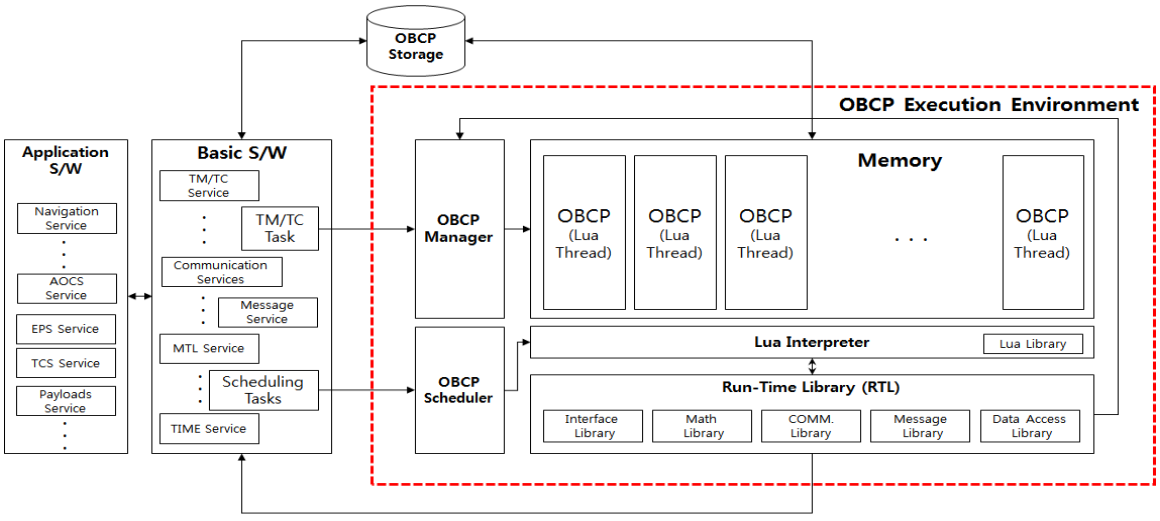

Figure 3. OBCP Execution Environment

1. OBCP Manager

The OBCP manager receives token codes of specified OBCP through TC interface function in BSW and loads it. It creates OBCP Control Block (OCB) from the header data of OBCP in order to provide run-time information for each OBCP. Static memory allocation mechanism is implemented for OBCPs. Figure 4 shows the OBCP state transition diagram. All OBCPs follow the same behavior based on the following states.

An OBCP can be one of following states. Empty state indicates that no OBCP is loaded with the given ID. The ID can be used to load a new OBCP. Loading state is transient state during the OBCP loading. The ID is reserved, the memory requested by the OBCP is allocated, the OBCP is not yet available for execution until it is fully loaded and the loading is validated. Stopped state is that the OBCP is not activated by the interpreter. Running state is that the OBCP is activated by the interpreter. Paused state is that the OBCP is paused, an OBCP can enter this in order to wait for an external event before resuming its execution.

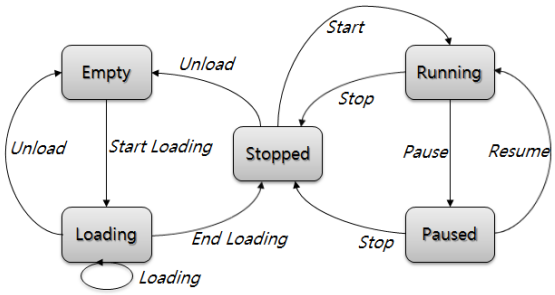

Figure 4. OBCP State Diagram

2. OBCP Scheduler

The policy of OBCP scheduler is cyclic round-robin, so the scheduler calls the OBCP interpreter to execute an OBCP for each OBCP duration according to each OBCP's activation level and duration information which are stored in each OCB. The OBCP can be executed on different activation levels, high frequency (HF, 10Hz), normal frequency (NF, 1Hz) and long frequency (LF, 0.1Hz). The execution of an OBCP is performed cyclically according to each OBCP's activation level. The execution time must be controlled by its cyclic frequency and allocated execution time in order to avoid a cyclic overload, so as not to endanger the rest of the OBS. If a whole OBCP can't be executed on a single cycle, its execution is spread on several consecutive cycles. The OBCP execution on one cycle is limited for a maximum duration which is predefined by writer of OBCP or user.

3. OBCP Language and Interpreter

In general, an interpreter has slow translation speed compared to natively compiled code. So, we investigated many interpreters of supporting script language.

Lua is a tiny script language and one of languages which is easily embedded in C program. Due to its simplicity and extensibility, Lua is widely ported to multi-platform and software even embedded application. Lua is embedded script engine and really lightweight: for instance, on Linux its stand-alone interpreter, complete with all standard libraries, take less than 150 Kbytes: the core is less than 100 Kbytes[9]. Independent benchmark shows Lua as one of the fastest language implementations in the field of scripting languages[10]. Since Lua 5.0, Lua employs register-based bytecode. It can leads to more efficient interpretation due to fewer bytecode instructions fetched for interpretation, reducing the fetch overhead which is crucial to the interpreter performance. A register-based instruction set architecture (ISA) leads to a better performance than a stack-based ISA, because it can remove many register moves corresponding to pushes and pos in a stack-based ISA, reducing the number of interpreted instructions[11].

We chose the Lua interpreter for OBCP execution engine due to above advantages and it provides methods for the execution and control of the OBCPs thanks to existing functionalities in Lua script engine. Each thread runs an instance of the interpreter in which the OBCPs run, and typically one main OBCP runs on a separate thread. So even if one of these threads fails, the execution of the others continues. Of course, it is also possible to stop or abort the execution of an OBCP via TC or events as shown in Figure 4.

Ⅳ. Experiment Evaluation

This chapter describes the development environment and experimental result. For the present work, we used Lua 5.2 as a execution engine to implement the VM for the OEE. Lua is free software distributed under the terms of the MIT license. It is certified Open Source software. Its license is simple and liberal and is compatible with GPL. Since Lua 5.0, Lua employs the virtual register machine

instruction set and provides the interpreter. Table 2 shows the software and hardware configuration for the experiments and development environment. We use the some of WCET benchmark programs for measurement of the performance[12]. Methods are translated to Lua programming codes and then they are executed on the target processor specified in Table 2.

Table 2. Hardware and Software Configuration

Item	Description
Processor	LEON3-FT 25MHz
Real-Time OS	RTEMS 4.10
Cross Compiler	Sparc-rtems 4.4.6
Interpreter	Lua 5.2
Debugger	GRMON
Development Env.	Eclipse IDE for C/C++

Figure 5 shows the OBS development environment. OBS including OEE has been developed using Eclipse IDE. The execution image is loaded and executed on target processor by GRMON. We can validate the execution time using GRMON.

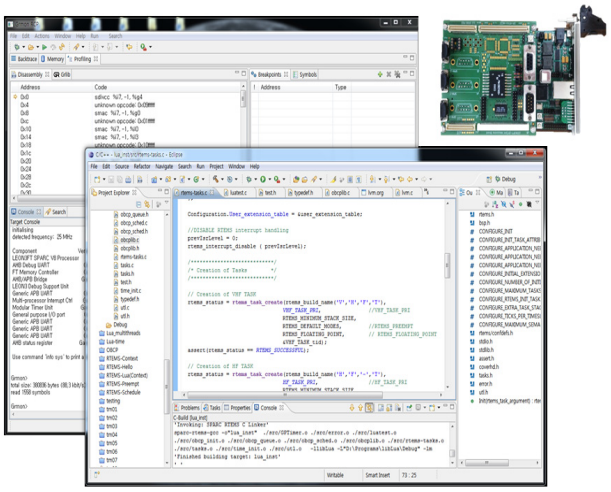

Figure 5. OBS Development Environment

To measure the benchmark running times of the Lua interpreter in OEE, we loads the benchmark programs on the OEE and executes them by executing OBS. We compare the performance of the Lua interpreter with using cache or not and floating point unit (FPU) or not. There are four results per each benchmark program. Table 3 shows the average execution time per one instruction.

In case of usage of cache, the OEE has an average speedup of 2.8 on the execution time. In case of the usage of FPU, it has an average speedup of 2.45 on the execution time. When the cache and FPU are used at the same time, it has an average speedup of 7.0 on the execution time. The usage of cache and FPU has a great influence on the execution time. We can validate that the best execution time is an average 6.384 (μsec) per one Lua instruction.

Table 3. Average execution time per one instruction (μsec)

Benchmarks	FPU		No FPU	
	Cache	No Cache	Cache	No Cache
sqrt	5.605	17.276	16.571	43.372
fac	6.187	18.934	17.249	41.828
fft1	5.984	17.909	20.930	51.225
fibcall	6.461	21.835	12.883	39.012
insertsort	8.755	26.423	23.502	56.715
janne_cimplex	3.979	14.070	13.219	41.037
qurt	7.431	21.857	21.191	52.992
recursion	6.673	19.551	12.643	34.441
Average	**6.384**	**19.732**	**17.273**	**45.078**

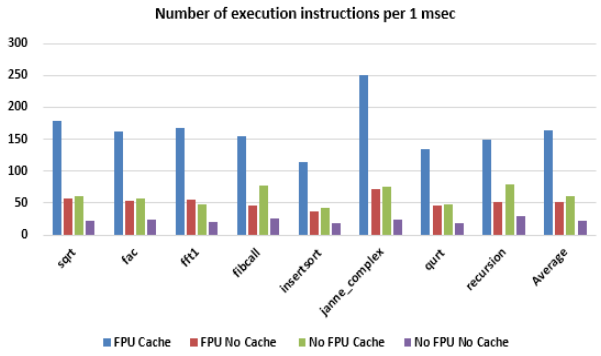

Figure 6. Number of execution instructions during 1 msec.

Figure 6 shows the number of execution instructions during 1 msec. It may be inferred from these data that the minimum 114 instructions are executed during 1 millisecond if the cache and FPU are used at the same time.

V. Conclusions

OBCP system enables autonomous operations without ground contact in the field of deep space missions such as Korea lunar explorer. OEE can be built as a form of the VM in OBS. Interpreter is an execution engine of interpreting and executing instructions in the VM. But interpreter has slow translation speed compared to natively compiled code. We adapted Lua interpreter with good execution performance to execution engine in the OEE and designed the overall of OEE. The development

environment has been configured and built for implementing OEE. OBS including OEE has been developed, ported on hardware target and evaluated the execution performance. We presented experimental results for OEE based on the Lua interpreter.

We found that the execution time is an average 6.384 (μsec) per one Lua instruction on the development environment. In case of usage of cache, the OEE has an aveage speedup of 2.8 on the execution time. In case of the usage of the floating point unit, it has an average speedup of 2.45 on the execution time. When the cache and floating point unit are used at the same time, it has an average speedup of 7.0 on the execution time. The usage of cache and floating point unit has a great influence on the execution time.

It is foreseen that the OEE for Korea lunar mission provides means to control spacecraft through OBCP which is script programs to be written in the Lua programming language, compiled to token and executed by Lua interpreter on-board. Because the creation and execution of an OBCP is independent with OBS during the mission, great flexibility, reduced development time and reduced risk for mission operations can be achieved besides adaptability to different missions and portability to different processors and real-time operating systems. It will be expected to use the developed OEE in the deep space exploration missions.

REFERENCES

[1] C. Steiger, R. Furnell, and J. Morales, "OBSM Operations Automation through the use of On-board Control Procedures," SpaceOps, May 2004.

[2] ECSS-E-ST-70-01C, "Spacecraft on-board procedures", ESA-ESTEC, April 2010.

[3] G.M. Lautenschläger, A. Hefler, R. Eilenberger, and J. Schandl, "The OBCP Concept used by ROSETTA", Proceedings of DASIA 2004, ESA SP-570, August 2004.

[4] C. Steiger, R. Furnell, and J. Morales, "On-Board Control Procedures for ESA's Deep Space Missions Rosetta and Venus Express", Proceedings of DASIA 2005, August 2005.

[5] F. Trifin, C. Steiger, A. Rudolph, and W.Heinen, "Simplying On-Board Control Procedure Development: A Generic Tool Based on ESOC Experience", AIAA 2008-3543, June 2008.

[6] M. Ferraguto, T. Wittrock, M. Barrenscheen, M. Paakko, V. Sipinen, "The On-Board Control Procedures Subsystem for the Herschel and Planck Satellites", Annual IEEE International Computer Software and Application Conference, COMPSAC.2008.218, pp.136-1371, 2008.

[7] S.Y Kang, K.H Yang and S.B Choi, "IP function development in COMS Flight Software", International Symposium on Remote Sensing, Jeju Korea, pp.171-174, November 2007.

[8] S.Y Kang, B.G Park and K.H Yang, "The On-Board Software Function for Meteo-Imager Images Planning Management in COMS", International Symposium on Remote Sensing, Jeju Korea, October 2010.

[9] R. Ierusalimschy, L.H. de Figueiredo, W. Celes, "The Implementation of Lua 5.0," Journal of Universal Computer Science 11 No.7, pp. 1159-1176, 2005

[10] D. Bagley. "The great computer language shootout. http://www.bagley.org/~doug/shootout/

[11] Yunhe Shi, Kevin Casey, M.Anton Ertl, David Gregg "Virtual Machine showdown: Stack versus registers" ACM transactions on Architecture and Code Optimization (TACO), Vol. 4, No. 4, Article 21, January 2008

[12]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

저자

강 수 연(Sooyeon Kang) **정회원**

- 1994년 2월 : 경희대학교 전자계산공학 학사졸업
- 1996년 2월 : 경희대학교 전자계산공학 석사졸업
- 1996년 3월 ~ 현재 : 한국항공우주연구원 선임연구원

<관심분야> : 임베디드 시스템, Fault-Tolerant 시스템, 병렬처리

구 철 회(Cheolhea Koo)

- 1997년 2월 : 충남대학교 전자공학과 학사졸업
- 1999년 2월 : 충남대학교 의용전자공학과 석사졸업
- 1999년 12월 ~ 현재 : 한국항공우주연구원 선임연구원

<관심분야> : 위성 하드웨어, 소프트웨어, 통신 프로토콜

주 광 혁(Gwanghyeok Ju)

- 1985년 2월 : 서울대학교 항공우주공학과 학사졸업
- 1992년 2월 : 서울대학교 항공우주공학과 석사졸업
- 2001년 2월 : Texas A&M 항공우주공학과 박사졸업
- 2001년 ~ 현재 : 한국항공우주연구원 책임연구원

<관심분야> : 항공우주 공학, 별추적, 우주탐사

박 시 형(Sihyeong Park)

・2014년 : 충남대학교 컴퓨터공학과 학사졸업
・2014년 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정

<관심분야> : 임베디드 시스템

김 형 신(Hyungshin Kim)

・1990년 : 한국과학기술원 전산학과 학사 졸업
・1991년 : Univ. of Surrey, U.K 위성통신학과 석사졸업
・1992년 ~ 2001년 : 한국과학기술원 인공위성센터 선임연구원
・2003년 : 한국과학기술원 전산학과 박사졸업
・2003년 ~ 2004년: Carnegie Mellon University Post Doc.
・2004년 ~ 현재 : 충남대학교 컴퓨터공학과 교수

<관심분야> : 항공우주 시스템, 저전력 컴퓨팅, 임베디드 시스템 소프트웨어