Software Development Framework for Small Satellite On-board Computers

By Sotaro KOYAYASHI1), Jun’ichi TAKISAWA2), Shinich NAKASUKA2) and Shinichi KIMURA1)

1)Tokyo University of Science, Noda, Japan
2)The University of Tokyo, Tokyo, Japan
(Received June 24th, 2013)

As the missions and functions of satellites differ, on-board computers used in small satellites are frequently developed individually at the relevant organizations. Although the satellite missions may differ, their complicated software may consist of common functional elements when the software structure is segregated into its functional elements, such as the equipment interfaces, ground communications, and attitude estimation. By recursively using a common software platform for many different satellites, we can expect an improvement in reliability through the mutual utilization of the platform and an increase in productivity from its inter-utilization. Therefore, we have developed a software development framework for the on-board computers used in small satellites. We created this software development framework through the following procedure. First, the basic composition of the on-board software was modularized for each function. The modularized components were then converted into class structures by extracting the similarity of each module. Using class structures, the software can be developed with a reduced amount of coding, and new software modules can be generated easily by simply modifying the code. In addition, the proposed software development framework was created for portability, allowing it to operate on many on-board computers. In this paper, the basic concept of the software development framework and its application to small satellites are described.

Key Words: Small Satellite, On-board Computers, Software Development Environment

1. Introduction

The development of satellites has resulted in significant benefits to humanity, for example, GPS communications and weather observations. Owing to their cost-effectiveness and short development time, the number of small satellites in use has increased extensively in recent years.1, 2)

For example, constellation-based small satellites or those with formation flight capability can play different roles when cooperating with the operation of a larger satellite. However, such cooperation requires that the satellite calculators have a greater capacity for processing data from their sensors and for controlling the calculations. Furthermore, for remote sensing using a camera, the calculators need to have a high capacity to measure a specified point on a complicated schedule.3-6)

On the other hand, because it is difficult to repair a satellite in orbit, the on-board computers should be highly reliable. In particular, they should be designed for high performance, in terms of not only hardware but also software.

In general, the software reliability increases if the software is recursively utilized and verified under varying circumstances. However, because the missions and functions of each satellite differ, many of the on-board computers used in small satellites are developed individually by the relevant organization.

Although the satellite missions may differ, their complicated software may consist of common functional elements then the software structure is segregated into its functional elements, such as the equipment interfaces, ground communications, and attitude estimation. Because such basic structures can be defined as fundamentally independent of the hardware and satellite structures, differences in hardware are expected to be covered by libraries and/or middleware. By recursively using a common software platform for many different satellites, we can expect improvements in reliability, i.e., the organizations in charge can mutually utilize the platform and bring about an increase in the productivity from the sharing of such improvements and additions to the platform. As shown in Fig. 1, this approach is also effective for developing highly complicated software because the different software components are modularized and their structures are easily understandable.

Therefore, we developed a software development framework for the on-board computers used in small satellites. We utilized software modules in C language, the structure of
which is easy to understand and is commonly used in software development.

We created the software development framework through the following procedure. First, the basic composition of the on-board software was modularized for each function. The modularized components were then converted into class structures by extracting the similarity of each module. Using these class structures, the software can be developed with a reduced amount of coding, and new software modules can be generated easily by simply modifying the code.

Furthermore, the proposed software development framework was created for portability, allowing it to operate on many different on-board computers.

As the first application, the software development framework was employed in the interface software of the on-board computers of small satellites used in the Funding Program for World-Leading Innovative Research and Development on Science and Technology (FIRST program).

As the second application, the software development framework was confirmed to operate on SH4-BoCCHAN-1, which is being developed by Kimura Laboratory.7)

In this paper, the basic concept of the proposed software development framework and its application to small satellites are described.

2. Concept of Modularity

It is generally a good idea to divide a large function into smaller individual functions and then assemble them. Equipment interfaces, including sensors and actuators, can be roughly classified into such structures as “initialization of the interface,” “periodic transmission (TX) of data requesting messages,” “periodic monitoring and receiving (RX) of status information,” or “non-periodic transmission and receiving of command messages.” These classifications enable the interfaces to be placed in order. Of course, not all equipment requires the “transmission of periodic data requesting messages.” Therefore, these functions have a null return allowing them to be put in order as the same structure.

Furthermore, interfaces other than the “initialization of the interface” can often be divided into the following processing forms: Received data usually have a header, footer, and in a few cases, checksums. In addition, data conversion with an application layer is required. The function should then be divided into sections such as the input, checking, processing, and output.

When receiving data from the equipment, the input section receives the data and delivers them to the checking section. The checking section then checks whether the (proper) data have arrived, and if so, delivers the data to the processing section. The processing section decodes and checks for errors within the delivered data, and then delivers the data to the output section. The output section converts the decoded data into a format understood by the application layer and stores the data in prescribed memory addresses.

When transmitting the data to the equipment, the division of each section is considered in reverse order. However, in this case the input and checking sections are not required because the processing section converts the data or adds headers and/or footers before delivering the data to the output section.

Therefore, the “periodic transmission of data requesting messages” is composed of processing and output sections; the “periodic monitoring and receiving of status information” is composed of the input, checking, and output sections; and the “non-periodic transmission and receiving of command messages” is composed of the processing of the transmission section as well as the input, checking, processing, and output sections of the output and receiving sections (Fig. 2).

3. Concept of Class Structure

In general, the class structure is a feature of object-oriented programming languages such as C++. In this research, however, we use C, which is not object-oriented but rather a standard language system. Therefore, it is necessary to put (non-object-oriented) C into an object-oriented class structure, which provides the advantages of class inheritance and the use of instances.

When a class is inherited, the contents of the superclass are copied to the derived class without alteration. As discussed in the previous section, the input, checking, and output sections that were separately possessed by the “periodic transmission of data requesting messages,” “periodic monitoring and receiving of status information,” and “non-periodic transmission and receiving of command messages” can be commonly used by the interfaces if modularization is applied. The processing section can be processed in the derived class of the inheriting class, not in the superclass, because the method of decoding the data is specific to each piece of equipment. Of course, frequently used procedures such as an endian conversion should be prepared as a separate module. This class is shown in Fig. 3.

An instance provides a class, which is only a blueprint. It helps connect similar types of equipment, for example.

Therefore, the creation of a class structure enables basic software to be commonly used more reliably under various situations. Furthermore, creating a class structure can minimize the changes to the interface even when new equipment are placed on-board or when the specifications are changed. It also enables the concept of multiplatforming.
4. Concept of Multiplatforming

Many on-board computers used in small satellites have different CPUs, interfaces, and development environments. This situation makes the reuse of software difficult. Therefore, software development framework has to be multiplatform capable. In general, programs who operate on multiplatform. Concretely, libraries put between hardware-dependent part and application part absorbs differences among hardware and operating system. Such libraries must be created using a model. Therefore, the following two items are necessary for creating a multiplatform: the creation of the library itself, and creation of a program that utilizes the library.

The difference in languages also becomes an obstruction when creating a multiplatform. As mentioned above, C is therefore used because it can be utilized in many different development environments.

This time, making a wrapper of the hardware interface driver is to be the library. The hardware interface driver differs according to the on-board computers used. Therefore, when making such a wrapper, the hardware interface driver is sorted into “initialization,” “receiving,” and “transmission,” to allow their calls to be made into a common library.

5. Hardware Composition

The on-board computers and equipment mentioned below form a Table Satellite as shown in Fig. 4.

5.1. On-board computers (FIRST program)

For the first hardware platform used in the software development system, we selected small-sized silicon on insulator (SOI)-system-on-a-chip (SOC) on-board computers (SOBCs) for the small satellites used in the FIRST program. An SOBC consists of three segments: the central processing unit (CPU) board, interface (IF) board, and power board, as shown in Fig. 5. Fig. 6 shows a schematic of the SOBC structure: the CPU board is the main processing unit, consisting of 64 MB of RAM, 2 MB of program flash memory, and an FPGA for the chip set and SOI-SOC processor. The SOI-SOC processor was developed for dual use in space and consumer applications, and it is highly tolerant to radiation. The IF board consists of FPGAs and various types of line drivers. It functions as a translator between SpaceWire and various types of interfaces. In contrast, the CPU board uses SpaceWire as the dedicated interface, whereas the IF board is suitable for various interfaces, depending on the applications, and acts as a SpaceWire translator. Because the CPU board obtains four SpaceWire ports, it can utilize a maximum of four IF boards, and can easily expand its interface capability. The power board generates power from several CPU- and IF-board sources. Moreover, it performs the hardware functions triggered by an external reset cue.

Fig. 3. Concept of class structure.

Fig. 4. Table satellite.

Fig. 5. The SOBC engineering model.
5.2. On-board computers (SH4-BoCCHAN-1)

For the second hardware platform used in the software development system, we selected SH4-BoCCHAN-1. SH4-BoCCHAN-1 OBC is a high-performance on-board computer developed by Kimura laboratory for CANSAT and small satellites (Fig. 7).

To deal with the complex and advanced mission requirements for small satellites, SH4-BoCCHAN-1 OBC has been developed as an on-board computer with a large memory space and high computing performance beyond the 8-bit one-chip microcomputers generally selected for the on-board computers of small satellites, such as PIC and H8. This OBC has many interfaces (UART×15, I2C×2, SPI, CAN, 10-bit A/D, GPIO, PWM) required for communication with satellite instruments and superior scalability, in addition to a high performance compatibility, low power, and small size. Therefore, it can be installed on CANSAT. The board was developed for installation on many types of small satellites including CANSAT, and can contribute to the high compatibility between satellites loading an OBC at the system level.

6. Software Composition

6.1. Macroscopic structure

The macroscopic structure of the on-board software constructed using the proposed software development system is illustrated in Fig. 8. The software is constructed at four major levels: (1) the operating-system level, (2) driver and middleware level, (3) SDK level, and (4) application level.

The operating system performs basic functions such as memory access and task management. The on-board computers of the FIRST program adopt TOPPERS, which is a μITRON-based real-time operating system, whereas SH4-BoCCHAN-1 adopts Linux. The operating system level compensates for the difference in the CPU structure.

The driver and middleware level function as the physical level interface between the CPU and peripheral components. As mentioned in the previous section, the interface of the SOBC can be customized using the IF board. Such variations in the interface structure are compensated by the FPGA, which acts as a link between various kinds of interfaces and SpaceWire. For the software, the driver and middleware level perform the setting and interface function of the FPGA.

The SDK level provides interconnected applications and the driver and middleware level in an abstract manner. The SDK level intermittently accesses the sensors, and the sensor data are converted into a generalized structured memory map. The applications can obtain sensor information by accessing a structured memory map. They are free from the hardware interface process of the peripheral equipment. When the applications need to send commands to the peripheral equipment, an application request calls the generalized function. The SDK level translates the commands for transmission to the peripheral equipment with proper timing, and makes up for the hardware dependencies and timing issues. Other tasks performed at the SDK level include the following: (1) automatically dispatching commands and telemetries; (2) performing command frame extraction, checking for errors, extracting command codes and parameters, and transmitting user-defined command tables; (3) packing the user-defined parameters in the operational document into a telemetry frame and automatically transmitting them; and (4) performing command and telemetry networking between on-board computers.

Based on these three-level functions, a user can create programs at the application level without considering the hardware structure and physical interface. The user program can obtain sensor information that accesses predefined

Fig. 6. Schematic structure of the SOBC.

Fig. 7. SH4-BoCCHAN-1.

Fig. 8. Macroscopic software structure.
parameters and can send a command to the actuator, which calls the predefined function. The sensor and actuator interfaces are abstracted into a generalized memory map and functions.

The software development system is supplied as a software framework from the operating system level to the SDK level. The user can implement personalized software at the entry points of the software skeleton. This system supports three types of entry points: time-driven, command-driven, and event-driven (Fig. 9). Time-driven entry points are evoked at certain time intervals and are mainly utilized for real-time control, such as altitude control. Command-driven entry points are triggered by commands from the ground or on-board computer networks. Event-driven entry points are triggered by predefined conditions, such as situations in which certain parameters are lower than a predefined threshold. Several entry points are predefined, such as a 50-ms interval time-driven entry point and a UVC event-driven entry point, and users can also add their own entry points.

![Three types of entry points.](image)

**Fig. 9.** Three types of entry points.

### 6.2. Class structure

Because C is used, there is no concept of “class” in the proposed system. Therefore, “structure” was chosen instead. The structure is similar to the class concept. However, the structure can have only variables as its members. The methods are therefore treated as if they are class members by describing prototype declarations of the functions in the header files of the on-board equipment, such as the following: Drivers/Driver_Super_class/interface class.

These functions are named as follows:

- [equipmentname]_[methodname]()
- Driver_Super_[methodname]()
- [interfacename]_[methodname]().

Class inheritance is available by including the header files and defining structures of the superclass in the structures within the child class.

### 6.3. Class member variable

Data exchanged between equipment, such as in sensor–actuator data exchanges, sometimes include a header or a checksum. In addition, data not used for control are sometimes received. Such data are unnecessary because they do not relate to the control part of the program. As mentioned in Section 6.1, delivering data by accessing the memory structures has the advantage in that an application is completely separated into the hardware, interface, and process.

Only necessary data are abstracted from the received data and stored in the [equipmentname]_STRUCT structure by each device. While transmitting commands, only necessary data stored in the structure and header, or in other locations, are added to the transmission. Other structure members, that specify the interface used, such as RX and TX counter inherit superclass, as well as [interfacename]_STRUCT structure members (for example, the baud rate and number of stop bits in the case of RS422), also exist. Fig. 10 shows the equipment name_STRUCT structure and the flow for the storage of the acquired values into a structure when receiving data from the equipment.

![Structure member variable and flow of data.](image)

**Fig. 10.** Structure member variable and flow of data.

### 6.4. Class member method

As mentioned in Section 3, the superclass function is assumed, as shown in Fig. 11. In this case, Driver_Super_init/rec/cycle/conf() is a prototype declared as a public function in the header file of the superclass, whereas Driver_Super_TX/RX/Analyze() is not a prototype declared as a private function. Among the drivers that have inherited the superclass, the Reaction Wheel (RW) driver is shown in Fig. 12 as an example. This RW_init/rec/cycle/conf() function internally calls the Driver_Super_init/rec/cycle/conf() function of the superclass. The Driver_Super_init/rec/cycle/conf() function then finally calls the [interfacename]_init/TX/RX() function of the specified interface. At this time, the pointer to DRIVER_SUPER_STRUCT in RW_STRUCT is delivered as the first argument. The above process enables dynamic actions even when the data analysis parameters differ among the on-board equipment.

![Driver_Super class method.](image)

**Fig. 11.** Driver_Super class method.
6.5. Wrapper of the hardware interface driver

In both the FIRST program and SH4-BoCCHAN-1, as the wrapper of the hardware interface driver, a common function [interfacename]_init(TX/RX()) is made as the library. As a result, when installing the equipment driver, the same program can be used completely in both the FIRST program as well as the SH4-BoCCHAN-1 (Fig. 13).

Moreover, Print() and Scanf() functions used for the debugging console were made as a wrapper. For the input/output from the debugging console, different procedures are required for the FIRST program and SH4-BoCCHAN-1, respectively; however, making the wrapper enables all other parts of the program to be entirely common.

7. Conclusion

This paper showed that the modularity and class structure of the basic composition of the on-board software has enabled the basic parts of the software system. In addition, by preparing the libraries according to the models used, a software development framework independent from the platforms has been realized. It was confirmed that the above is effective for making such a programming framework for use in the development of satellites. As future work, we intend to expand the software development framework into a widely used high-performance system through an enhancement of the class and library.

Acknowledgments

This research was supported by the Japan Society for the Promotion of Science (JSPS) through the Funding Program for World-Leading Innovative R&D on Science and Technology (FIRST program), which was initiated by the Council for Science and Technology Policy (CSTP).

References