Article

Architectural Process for Flight Control Software of Unmanned Aerial Vehicle with Module-Level Portability

TSogbayar Jargalsaikhan [1], Keonpyo Lee [2], Yong-Kee Jun [3,*] and Seongjin Lee [2,*]

[1] Department of Informatics, Gyeongsang National University, 501 Jinjudaero, Jinju 52828, Korea; tsogbayr@gnu.ac.kr
[2] Department of AI Convergence Engineering, Gyeongsang National University, 501 Jinjudaero, Jinju 52828, Korea; gnurvy2@gnu.ac.kr
[3] Division of Aerospace and Software Engineering, Gyeongsang National University, 501 Jinjudaero, Jinju 52828, Korea
* Correspondence: jun@gnu.ac.kr (Y.-K.J.); insight@gnu.ac.kr (S.L.); Tel.: +82-55-772-1371 (Y.-K.J.); +82-55-772-1378 (S.L.)

Abstract: To apply UAVs (Unmanned Aerial Vehicle) into different fields, including research and industry, and expand it quickly, reliable but modular software is required. The existing flight control software (FCS) of the UAV consists of various types of modules categorized into different layers, and it is responsible for coordinating, monitoring, and controlling the vehicle during its flight. This paper proposes *mpFCS*, a structure of UAV flight control software, which provides portability to its modules and is easy to expand. The *mpFCS* consists of four segments and several modules within the segments. *mpFCS* provides portability for each module within the segment. Existing software does not provide portability for its modules because of the tight coupling resulting from its different and private interfaces. The *mpFCS* uses interfaces of the standard airborne software architecture to transfer data between its modules. Moreover, the structure provides portability for its modules to run in the standard airborne software environment. In order to verify the *mpFCS*, we tested the *mpFCS* with the conformance test suite of the airborne software that provides the testing environment for the interfaces and modules of the software. The *mpFCS* passed the test. Test results show that all modules of the *mpFCS* are portable. Additionally, portable modules can be interoperable with other software, and the structure is expandable with new modules that use standard airborne software interfaces.

Keywords: unmanned aerial vehicle; flight control software; portability

check for updates

Citation: Jargalsaikhan, T.; Lee, K.; Jun, Y.-K.; Lee, S. Architectural Process for Flight Control Software of Unmanned Aerial Vehicle with Module-Level Portability. *Aerospace* **2022**, *9*, 62. https://doi.org/10.3390/aerospace9020062

Academic Editor: Sergey Leonov

Received: 30 November 2021
Accepted: 20 January 2022
Published: 25 January 2022

1. Introduction

UAVs are currently becoming popular due to their high maneuverability, good performance, low cost, and reliability [1,2]. By 2021, the target of the drones market is estimated at 4.5 billion US dollars and will reach 10.4 billion US dollars by 2031 [3]. Controlling a UAV from a ground control station (GCS) provides a new possibility for the UAV to reach the furthest areas with few human resources needed and require minimal energy, time, and effort. Recent technological advances create more new possibilities for UAVs [4,5], and UAVs are employed commonly in tasks such as surveillance [6], search and rescue missions [7] in the disaster area, wildfire, tracking, borderline security [8], remote sensing of the environment [9], etc. This is one of the biggest reasons UAVs are being adopted worldwide, especially by these sectors: personal, commercial, military, academia, and future technology.

Flight control software (FCS) is required to control and monitor UAVs from GCS [10]. Necessary characteristics of the software are (1) navigation, (2) guidance, and (3) control. Navigation is the process of data acquisition, data analysis, and estimation of information about vehicle state and its surrounding environments. Several fundamental sensors are used in the navigation process: an accelerometer, gyroscope, magnetometer, GPS, and other peripheral sensors such as airspeed sensor, barometer, etc. Guidance is the process concerned with user input, such as path planning, mission planning, flight mode changing, etc. The control component ensures that the flying vehicle follows the desired path and altitude by manipulating control surfaces. Employing UAVs in different fields requires modular and portable software that can expand quickly based on increased functionality. When functionalities and requirements of the software increase, the software can fulfill new functionalities and requirements by developing (or porting) a new module (from other software).

Software (or software module) portability is an important feature of the software, as it enables the software (or software module) to be reused in other software tools and/or operating environments [11]. When the design of a software does not consider portability and was not developed in modules, it is not possible to adopt the software in a new environment. This is because the software must be redesigned, reconfigured, and modified significantly to be able to run in a new system. Converting non-portable software to portable software is very costly because most of the conversion needs to be processed manually, and it also has to be tested for conformity [12,13]. By porting the software (or expanding with modules of other software) from one environment to another, major economic savings can be achieved by dropping out the development and modification costs of the software [14].

In order to be portable, the software must satisfy the portability requirements such as layered and modular structure, loosely coupled and high cohesion on its modules, standard interfaces, and hardware abstraction layer (HAL) [15]. These requirements result in easy expandability of the software. In order to be expandable and portable, many UAV software followed a layered structure with different modules and centralized all message-passing services between its modules as a middleware. The structure of existing UAV software [6–10,16–23] consists of common layers that contain different modules, such as

1. a hardware layer responsible for interface to the device,
2. a middleware layer, which works as a bridge between the hardware layer and application layer, and
3. an application layer that contains various functional modules, including navigation, guidance, and control modules.

Unfortunately, existing software does not satisfy all requirements and fulfills only some of them. Each software provides unique interfaces for communication between modules. These unique interfaces introduce a barrier to the portability of the modules and provide more complexity to the software and bring tight coupling into modules.

Due to its tightly coupled modules, the existing software cannot provide portability into its modules, and it is not easy to extend the software with new modules and transfer its modules to a different environment. Despite the technical advances, many software today result in the tightly coupled integration of software modules without regard to portability. This lack of focus on portability results in software modules that are unable to be reused from one software to another without significant modification.

This paper proposes a new structure of the flight control software of UAV named *mpFCS* (module-level portable Flight Control Software) that can provide portability to the software and each software module. In order to eliminate tight couplings and provide portability for its modules, we isolated its modules and designed the structure of the software. To design a new structure, we analyzed the structure of existing software at its module level. To do this, we used the standard FACE (Future Airborne Capability Environment) architecture for airborne software, and a new structure is based on specifications and requirements of the architecture that provides portability to the software. The FACE standard architecture [24] defines interfaces intended to develop software made up of portable components. The *mpFCS* consists of different portable modules in different segments, which can send and receive a message between its modules and between the software and GCS. We implemented a new structure with its core modules: flight mode, attitude, position, servo control, sensor, communication, transmitter, and IO module. Each module uses the FACE standard interfaces to transfer data between its modules. We evaluated the software with the FACE conformance test suites.

The result shows that each module of our software is portable to the different software that runs in the FACE environment, and they are also interoperable with the different software modules that use the same interfaces. Furthermore, to prove its expandability, we developed a sample module and ported it into our software. The result shows that the structure is expandable with a new module.

The main contributions of the work are as follows:

- Framework for providing module-level portability for the software: we propose the framework that can provide portability to the software. The framework not only provides portability to the software, but also provides its modules.
- Portable and expandable software with module-level portability: The proposed structure *mpFCS* provides portability for the software as well as portability for its modules to run in the standard FACE environment. The software based on the structure can expand with any new user-defined module or previously well-developed module that uses the standard FACE interfaces.
- The software can be used as a prototype of actual FACE software. It means developers can test their software or module with actual FACE-based software.
- Guideline for creating a portable module or converting the current module into a portable module. The guideline provides steps of analysis and how to do it.

The remainder of this paper is organized as follows. Section 2 explores related work, and Section 3 introduces a problem of the existing software and its causes. The solution for the problem and the proposed structure *mpFCS* are described in Section 4. Then, Section 5 explains the overall design of the *mpFCS* and its modules. Section 6 discusses the implementation of *mpFCS*. Section 7 provides experiments, results, and discussion. Finally, we conclude the paper in Section 8.

2. Related Work

UAVs are growing rapidly, and their software functionalities are increasing. Applying UAVs in different fields requires portable and modular software that can expand quickly based on increased functionality and is transferable to different platforms. Nowadays, various types of software are developed by academia, industry, and hobbyists, including open source and commercial software. First, this section explores the definition of portability and its characteristics. Then, the section continues with an analysis of existing software and summarizes how developers enable portability to their software.

2.1. The Portability and Its Characteristics

According to ISO/IEC 25010 [25], portability is the "degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another." Additionally, Mooney, J.D described [11,26] portability as a desirable attribute of the software and is typically concerned with reusing complete software (or part of the software) on new platforms. In other words, if the software is written without regard to hardware implementations and other dependencies, the software can be called portable [27]. As a view of Mooney, J.D, the primary goal of software portability "is to facilitate the activity of porting" of software from the "environment in which it currently operates to a new or target environment" before allowing "reuse of the complete existing application in the new environment" [11].

For achieving portability into software and its modules, software needs to fulfill requirements. Within the European standards for the aerospace industry (ECSS) [28], IEEE-830 [29], and ISO 2382-1 [30], and several other concepts are provided to describe various types of requirements of portability. There are similar requirements such as independence of (1) the operating system, (2) the middleware, and (3) the programming language. Haile [31] mentioned that the software could not provide portability to itself and its module due to (1) different types of programming framework, (2) proprietary application programming interface (API), and (3) its data format. Thus, Haile [31] listed up requirements as (1) common programming framework, (2) standardized API, and (3) data format for the standard API. In this paper, we followed characteristics of the portability described by Beningo, J. [15], which can be summarized as follows:

1. Simple and understandable structure (layered structure).
2. Separated into different and independent modules (modular structure).
3. Loosely coupled and high cohesion on its modules.
4. Standard interfaces between its modules to transfer data.
5. Hardware abstraction layer (HAL), the standard function set that can be used to access hardware functions without a detailed understanding of how hardware works.
6. Well-organized and detailed documentation for developers to understand its structure easily.

2.2. General Structure of the Existing Software

In this subsection, we summarized the structure of UAV software. In order to be portable, many software follows a layered structure with different modules and use different messaging mechanisms between its modules [15]. Figure 1 shows the most common layers and their modules in the structure of the existing UAV software. The software runs on different operating systems, such as open-source real-time operating systems including NuttX, Free RTOS, and ChibiOS. Most of them are open-source software, but some of the open-source software in academic work uses commercial operating systems such as VxWorks. Each software uses its message mechanism in the middleware layer. The software has various types of modules in the application layer. Figure 1 included the most common modules of existing software, such as control, position, attitude, motor, sensor, navigation, and communication modules.

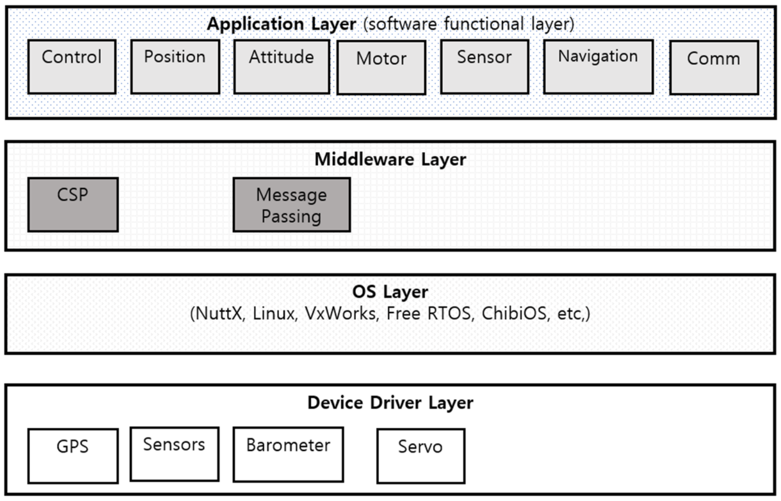


Figure 1. The most common structure of the existing software, which consists of layers and modules.

Hedge et al. [10] developed flight control software for UAVs with two primary layers and four primary modules. Chong et al. [2] designed and developed flight control software for small UAVs based on the real-time operating system. The software is divided into three layers and consists of six application modules: navigation, control, fault manage, sensor, servo, and communication module. He et al. [16] created an open-source and lightweight flight control software for a UAV, consisting of two layers and main flight modules for initialization, sensing, PID control, and PID actuating. Ying et al. [17] designed and developed a distributed flight control software for UAVs. The software architecture is divided into four layers according to its functions.

There are many open-source flight control software for UAVs [32]. We list the popular software for completeness of the paper: ArduPilot [18] started by Chris Anderson, PX4 [19] created by Lorenz Meier, PaparazziUAV [20] from ENAC (École Nationale de l'Aviation Civile, France), MultiWii series [21], including Baseflight, Betaflight, INAV, Hackflight, and Cleanflight, developed by Alexandre Dupus [33], dRonin [22] and OpenPilot series [34] from TauLabs. Among these open-source software, we choose the most popular software: ArduPilot, PX4, PaparazziUAV, and dRonin.

2.3. Enabling Portability in the Software

The messaging mechanism in the middleware layer decreases the coupling of the modules and increases the portability capability. Additionally, it provides independent and separate modules and expands the software with a new module [15]. Table 1 compares the existing flight control software based on their documentation and shows its general structure, messaging mechanism, written language, and supported operating system. Chong et al. [2] and Ying et al. [17] use message bus in the middleware layer. Theile et al. [23] provide a modular autopilot framework named uavAP for UAVs. As the core functionality of the uavAP, cpsCore is responsible for managing its modules through its propriety interface, which is defined in the framework.

Table 1. Comparison of the flight control software and its features.

Features		**He** [16]	**Ying** [17]	**Chon** [2]	**Hegde** [10]	**Theile** [23]	**AP** [18]	**PX4** [19]	**P-UAV** [20]	**dRonin** [22]
General Structure	Layered	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Modules	Y	Y	Y	Y	Y	Y	Y	Y	Y
Messaging		Direct Calls	Middle-ware	Middle-ware	Direct Calls	Middle-ware	Direct Calls	Middle-ware	Middle-ware	Middle-ware
Language		C++	N/A	N/A	N/A	C++	C++	C	C	C++
OS supported		Linux, ChibiOS	VxWorks	VxWorks	Free RTOS	Arch-Linux	Linux, ChibiOS	NuttX, ROS	ChibiOS	PiOS
Is it Portable? (Based on its documentation)		N	N	N	N	N	Y	Y	N	N

AP—ArduPilot, P-UAV—PaparazziUAV, N/A—Information not available, Y—Yes, N—No.

The popular open-source software PX4 [19] uses its messaging API named uORB for its internal communication to transfer data between modules. uORB is automatically started on the bootup, and many modules depend on it. Messages are defined as separate .msg files in the msg/folder. For its communication module, to connect UAV to GCS, it uses the MAVLink protocol. PaparazziUAV [20] uses proprietary middleware named ABI (AirBorne Ivy) to transfer data between modules. ABI gives an easy way to allow the software modules to exchange data. Messages are defined in the XML file with a unique name and id. For its communication module, to connect UAV to GCS, it uses PPRZLINK. dRonin [22] defines the data representation named UAVObject, and it is used for inter-module communication. Different functions are used to transfer data defined in UAVObject. UAVObjects are generated dynamically from XML definitions. For its communication module, to connect UAV to GCS, it uses the UAVTalk protocol.

3. Portability Issues of the Current Software

Due to the increase of tasks and functionalities of UAVs, its software is becoming more complex. The general structure of the existing UAV software is divided into common layers that contain various types of different modules. However, each software uses different interfaces to transfer data between its modules, resulting in tight couplings of modules. Due to the tight coupling of the modules, the software does not provide portability for its modules. This section discusses the status of existing UAV software and explains the problem related to portability and its cause.

3.1. Current Status of the Software

In order software to be portable, the software must satisfy the characteristics mentioned in Section 2. First, the general structure of existing UAV software follows a simple structure that [2,6–19] consists of common layers. Those common layers include (1) the hardware layer (the main IO device module), (2) the middleware layer that is responsible for connecting the application and hardware layers, and (3) the main flight application layer with different modules that control UAV's movement. The software consists of different and separated modules based on their functionalities located in the application layer. To decrease the tight coupling and increase the high cohesion of the modules, the software uses different types of middleware that we explained in Section 2, which centralizes all message passing in one place. Nevertheless, each software uses its proprietary interfaces between its modules when it transfers data. Additionally, only Ying [17], ArduPilot [18], PX4 [19], and dRonin [22] have HAL. One of the non-functional requirements of being portable is its documentation. Academic works do not provide any documentation. Open-source software provides documentation, but because of its complex functionality and huge numbers of source lines of code, it is not easy to understand its documentation, and it takes much time. Only the main source code contains more than the following source lines of codes (SLOC): ArduPilot 350K, PX4 290K, PaparazziUAV 240K, dRonin 300K.

3.2. Portability Issue

The existing software does not fulfill the main requirements of portability that we mentioned in Section 2. A comparison of the existing software based on portability requirements and its summary is shown in Table 2. Each software followed a simple (layered) and modular structure, and most of them centralized their messaging in the middleware layer. Nevertheless, each software uses different types of messaging mechanisms that have proprietary interfaces in its inter-module communication. These different interfaces introduce a barrier to the portability of the modules and software modules between different software. Due to proprietary interfaces, expanding the software is becoming more complex, and it requires understanding the structure of each software to port the user-defined module to the software and run the software on the different platforms. Some modules use direct call of functions of other modules. A few of them provide hardware abstraction layers (HAL). The comparison result shows that each software cannot satisfy the portability requirements, thus it cannot provide portability not only to the software, but also to its modules.

Table 2. Comparison on requirements of the portability of the existing software.

Requirements		**He [16]**	**Ying [17]**	**Chon [2]**	**Hegde [10]**	**Theile [23]**	**AP [18]**	**PX4 [19]**	**P-UAV [20]**	**dRonin [22]**
Simple structure		Y	Y	Y	Y	Y	Y	Y	Y	Y
Modular		Y	Y	Y	Y	Y	Y	Y	Y	Y
Centralized Messages		N	Y	Y	N	Y	N	Y	Y	Y
Standard interface		N	N	N	N	N	N	N	N	N
HAL		N	Y	N	N	N	Y	Y	N	Y
Well Documented		N/A	N/A	N/A	N/A	N/A	Y *(ab)	Y *(ab)	Y *(abcd)	Y *(abc)
Analysis (Does it satisfy the requirements of portability?)	Software Level	N	N	N	N	N	N	N	N	N
	Module Level	N	N	N	N	N	N	N	N	N

AP—ArduPilot, P-UAV—PaparazziUAV, N/A—Information not available, Y—Yes, N—No; * Documented, but it (a) is partially outdated, (b) is complicated to understand, (c) has an incomplete description, (d) has anomitted description.

Figure 2 shows proprietary interfaces between modules and layers that cause tight coupling of its modules. The module-level portability provides the easiest way to expand the software by porting a new module. Using the standard architecture reduces the coupling of the modules and brings portability into the software and its modules.

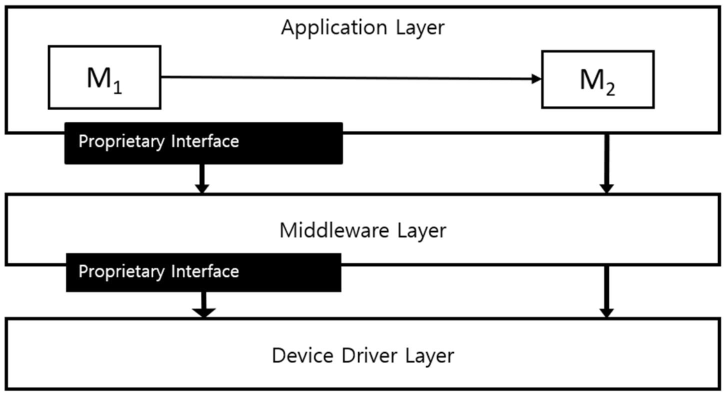


Figure 2. The proprietary interface between layers and its modules.

4. Solution

This section introduces a new structure of the software named *mpFCS* that provides portability to its modules. First, the section describes the main specification of the common modules, and then the section continues our design strategy for bringing portability to modules. We analyzed the structure of the existing flight control software, its modules, and connection methods between its modules and then segmented it into the FACE architecture [24], which provides airborne software's standard interfaces and architecture. Bringing the standard architecture into the UAV flight control software provides portability to the module.

4.1. Standard for the Portability Problem

Owing to the software's tightly coupled modules in the different layers, it is not easy to port the software and its modules in a different environment without modifying the source code. In order to solve this problem and enable portability to the software, we choose the FACE (Future Airborne Capability Environment) architecture. Figure 3 explains the overall design of the FACE architecture, which can provide portability into the software modules.

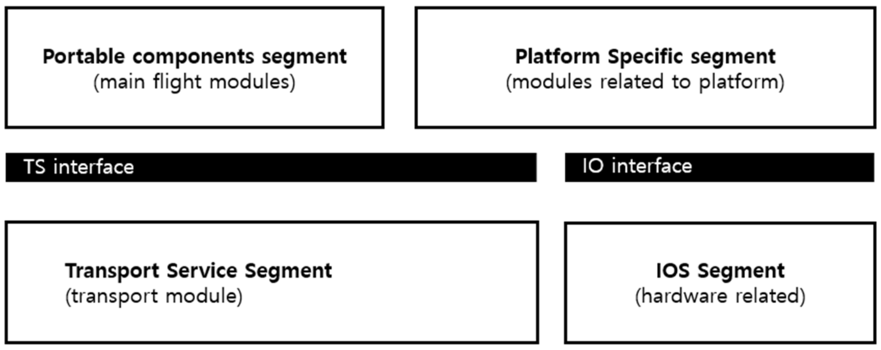

Portable components segment
(main flight modules)
Platform Specific segment
(modules related to platform)
TS interface
IO interface
Transport Service Segment
(transport module)
IOS Segment
(hardware related)

Figure 3. The overall design of the FACE architecture.

The FACE architecture from the Open Group provides portability to airborne software that can be redeployed on different computing hardware and/or the standard airborne software environment. The FACE architecture is composed of five logical segments:

1. The Operating System Segment (OSS): It provides and controls access to the computing platform for the other FACE segments.
2. Input/Output Services Segment (IOSS): It is a bridge for data from the device drivers to the PSSS.

3. Platform-Specific Services Segment (PSSS): It creates an infrastructure unique to the platform that provides device data to the components located in the Portable Component Segment.
4. Transport Services Segment (TSS): It provides movement of data between PCS and PSSS.
5. Portable Components Segment (PCS): It is a set of portable components.

The architecture defines a set of standardized interfaces providing connections between the FACE architectural segment such as Operating System Segment Interface (OSS Interface), the Input/Output Services Interface (IOS Interface), the Transport Services Interfaces, and Component-Oriented Support Interfaces. Additionally, the architecture defines the Unit of Portability (UoP), a set of components that provides one or more mission-level capabilities. The main characteristic of UoPs is portability to the other software that follows the FACE architecture. In this paper, we use the word module instead of the abbreviation UoP, which means the word module and UoP are interchangeable and have the same meaning in our work [24].

4.2. Restructuring Process

To follow the FACE architecture, first, we need to understand the structure of the existing software and its module's dependencies and design its structure via following the specification and requirements of the FACE architecture. To achieve the above specification, we analyzed various UAV software and matched similar software layers into the FACE segments depending on each segment's functionality and requirements. Figure 4 shows the matching process of the existing software into the FACE architecture. We called this process the AMR-Process (Analyze, Match, Restructure Process). It consists of two phases: analyzing the structure phase and redesigning the structure phase.

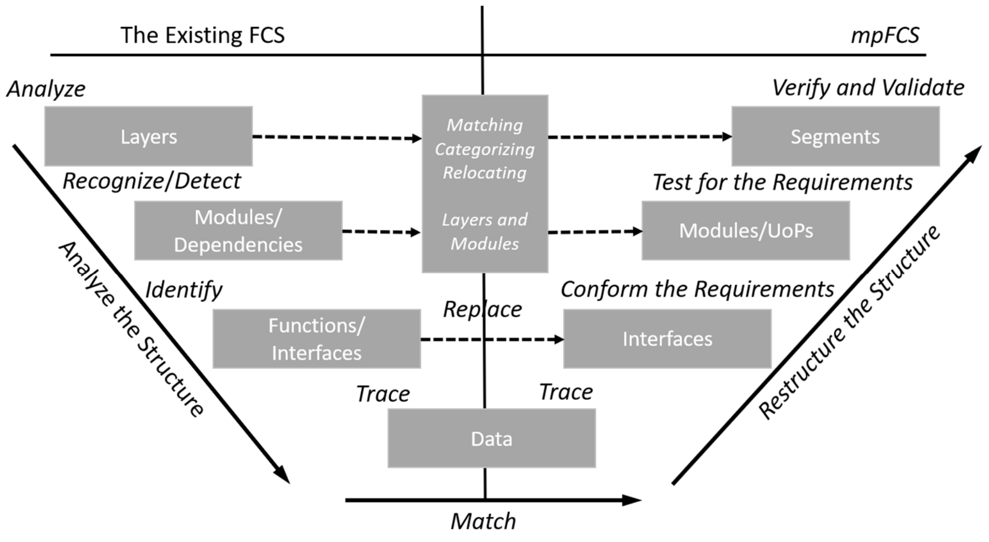


Figure 4. The AMR-Process of matching the existing FCS into the FACE architecture.

In the phase of analyzing the structure, we followed the steps below to analyze the structure of the existing software.

1. Analyze the software layers.
2. Recognize the modules and detect their dependencies.
3. Identify the functions and interfaces between modules.
4. Trace data between the functions to match the data for the next phase.

In the restructuring phase, we match the modules with the FACE segments based on analyzed data from the previous phase, the main requirements of the FACE architecture segments, and interfaces. Due to functions, dependencies, interfaces between modules, and input/output data, modules in the application layer match the PCS of FACE architecture.

The modules in the middleware layer are divided into the TSS and PSSS, depending on their functionality and dependencies. The device driver layer matches with IOS because of its role in the software.

After identifying and analyzing module dependencies, we replace the software's propriety interfaces with the FACE standard interfaces to transfer data. All the above processes result in portability to the software and its modules on the FACE software environment.

4.3. The Proposed Structure: mpFCS

In order to validate the AMR-Process on different types of software, we choose the software that (1) is open-source, (2) is currently effective, (3) follows a common structure, and (4) has good documentation. The software that fulfill the criteria are PaparazziUAV, ArduPilot, and PX4. The result of the analysis phase is shown in Figure 5a.

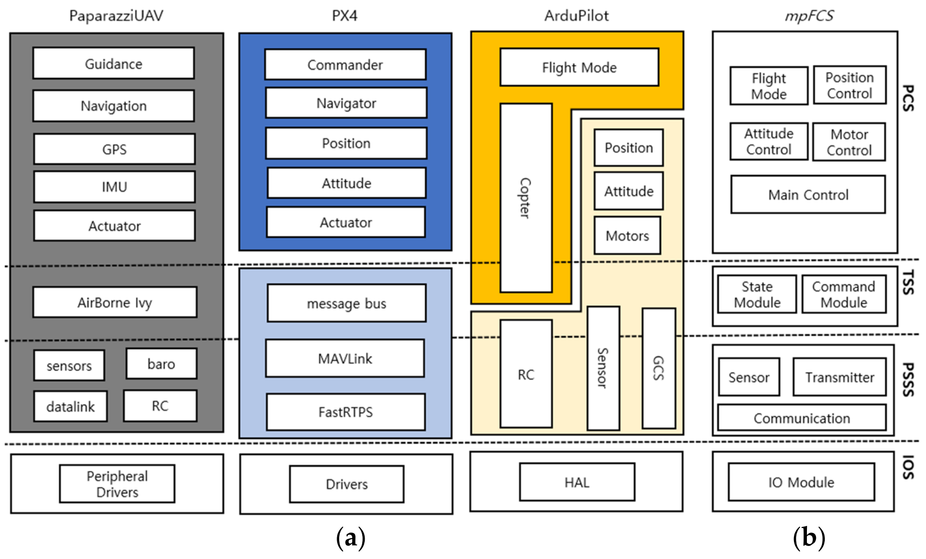


Figure 5. The analysis of the modules of open-source software and the *mpFCS*. (**a**) Open Source (PaparazziUAV, PX4, ArduPiolot). (**b**) *mpFCS*.

Based on the result of the analysis phase, we execute the restructuring process. Our proposed structure resulting from the redesigning phase has different layers and is modular, same as the existing FCS. It uses the FACE standard interfaces to transfer data instead of proprietary interfaces. Due to the standard architecture, it results in portability of the software and its modules. Table 3 shows a general comparison of our solution with the existing FCS.

Table 3. Comparison of the existing FCS and the *mpFCS*.

Characteristics		**Existing FCS**	***mpFCS***
Layered		Yes	Yes
Modular		Yes	Yes
Centralized Messages		Yes *	Yes
Data Transfer	Interface	Proprietary Interfaces	Standard Interfaces
	Direct Call of Functions	Yes	No
Portability	Software	No *	Yes
	Module Level	No	Yes

* some FCS does not centralize messages.

5. Design of the Proposed Structure: *mpFCS*

In order to solve the portability problem of modules, we followed the AMR-Process mentioned in the previous section on the structure of different flight control software and designed the structure. This section explains the overall design of the *mpFCS* based on the result of the analysis. Then, we explain each segment of the structure and its modules. The FACE architecture defines only the segments, interfaces, and requirements of each segment and interface. Thus, we placed modules of the software in the segments based on their requirements and specifications to follow the FACE architecture.

5.1. The General Structure of the mpFCS

Figure 5b shows the overall design of *mpFCS* that proposed structure of UAV software with module-level portability, which is based on FACE architecture. Figure 6 describes the data flow of all modules in the segments. The IO segment receives data and sends it to the platform-specific service segment via the IO interface. Based on the IO service segment's output, platform-specific service segments calculate data and send the calculated result to the portable component through the transport service segment. The transport service segment uses transport interfaces to transfer data between the portable component and platform-specific service segments. On the other hand, to send a state message from UAV to GCS, the portable component segment transfers state data (GPS location, altitude, heading, IMU data, mission state) to the platform-specific services segment through the transport service segment. When the platform-specific services segment receives data, it calculates, converts, and sends the data to the IO services module via the IO interface.

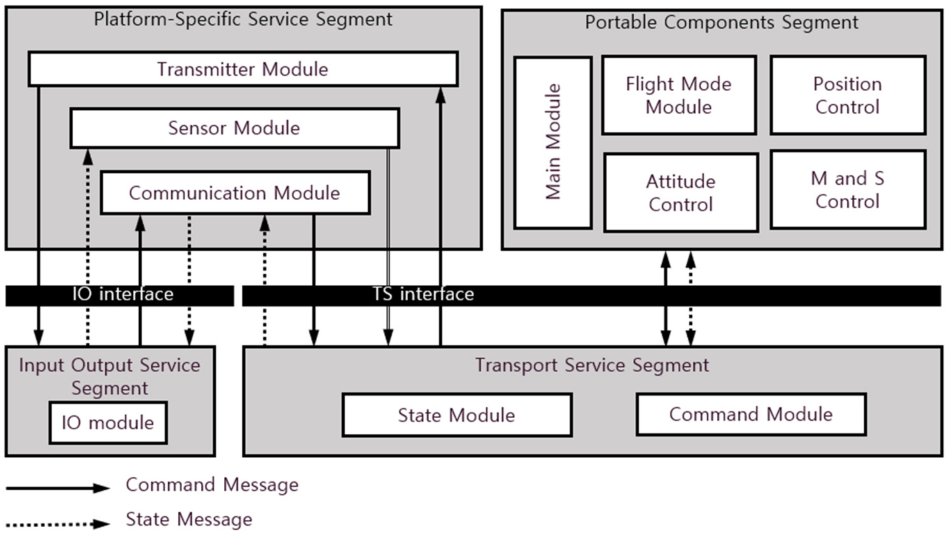


Figure 6. Data flow of the *mpFCS*.

Running the common and open source UAV software does not require a specific operating system, such as a real-time operating system (ARINC-653, POSIX, etc.). The software runs on open-source operating systems, including Robot Operating System (ROS), NuttX, ChibiOS, etc. The Operating System Segment of the FACE architecture is not included in our design and is not implemented. Nevertheless, since we use the FACE interfaces between the modules, porting the structure into such operating systems is not a problem.

5.2. IO Service Segment

The IO module in the I/O Service Segment (IOSS) is responsible for interface devices, and it connects hardware drivers with platform-specific modules, which is the basis of information between platform-specific modules and external devices. The module receives command data from the I/O device and passes them to the module in the Platform Specific

Service Segment (PSSS). Concurrently, the module receives state data from the module in the PSSS. All input and output data between modules in IOSS and PSSS goes through the IO interface.

5.3. Platform Specific Service Segment

We designed modules in the Platform Specific Service Segment (PSSS) that depend on the IO module data, and all data flow in the PSSS is shown in Figure 7.

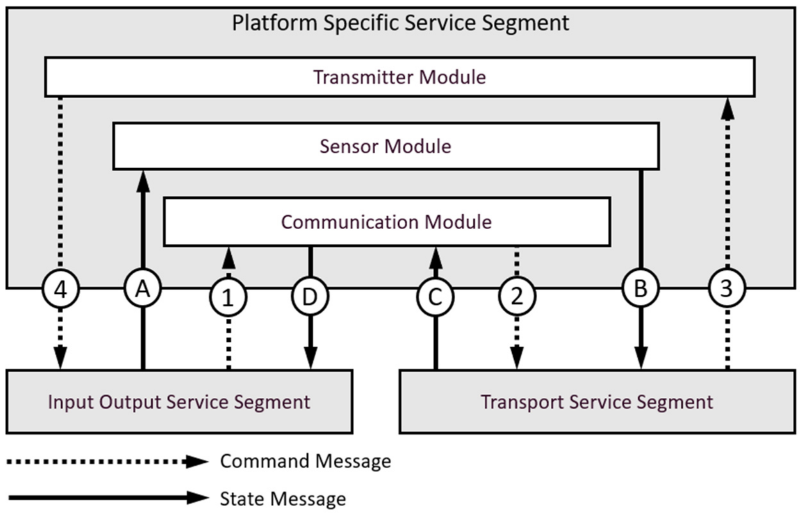


Figure 7. Data flow in the PSSS.

The segment consists of three modules: sensor-calculation, communication, and transmitter modules. The segment receives data from the Transport Service Segment (TSS) or IOS and sends them to IOS or TSS.

- The communication module is responsible for sending and receiving a message between UAV and GCS. It receives a command message from IOSS via the IO interface, then passes a message to TSS using the TS interface. Simultaneously, it receives a state message from TSS through the TS interface and passes it to IOS through the IO interface.
- The sensor-calculation module is responsible for data from various sensors, and it receives data from IO devices via the IO interface, converts them, and transfers them to the transport module in the TSS via TS-interface.
- The transmitter module is responsible for converting and transferring data to actuators, which comes from the portable component segment.

5.4. Transport Service Segment

The transport module in the Transport Service Segment (TSS) can transfer data between PSSS and the Portable Components Segment (PCS). Figure 8 shows the data flow between TSS and PSSS. The transport module consists of two submodules which are StateModule and CommandModule.

The StateModule receives a state message from the main flight module and transfers it to the communication module in the PSSS. Additionally, it receives sensor data from the sensor-calculation module. At the same time, the CommandModule receives command data from the communication module, and it checks data and transfers data to the specific module in the main flight module depending on the message-id. Besides, it receives a message from the main flight module and transfers a message to the transmitter module in the PSSS. All data between PSSS, TSS, and PCS are transferred through the FACE TS interfaces.

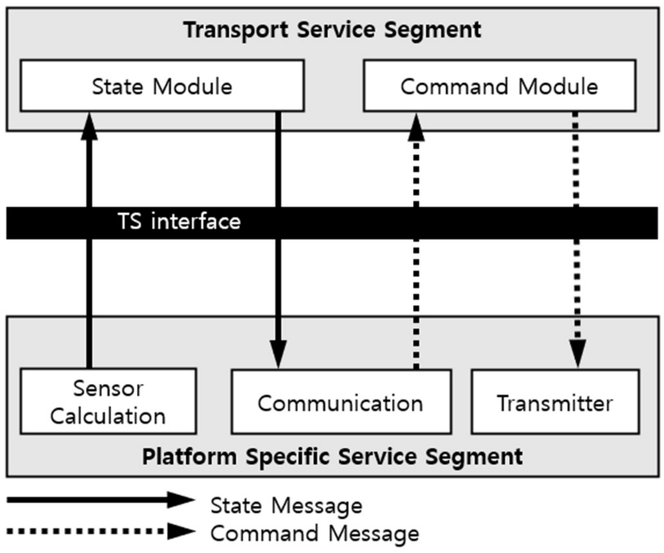


Figure 8. Data flow between TSS and PSSS.

5.5. Portable Component Segment

The Portable Component Segment, designed at the top of the software structure, is the key to realizing the UAV flight control function. The main functions, such as UAV flight status management, navigation, and control management, sensor and servo management, and peripheral device management are achieved at this segment, and it consists of five modules: flight control, altitude, position, servo-motor, and main control modules. The modules in the PCS receive and send data from/to TSS via the TS interface.

- Main control module: The main control module in the PCS is responsible for controlling the UAV, managing, and showing the vehicle's current state.
- Flight control module: is responsible for changing its flight modes and converting the user's input into a lean angle, rotation rate, climb rate, etc., that is appropriate for this flight mode.
- Attitude control module: calculates attitude and target roll, pitch, yaw rotation rates, converts to high-level motor requests.
- Position control module: calculates path and the position, velocity, and acceleration, and updates position.
- Servo-motor control module: receives high-level motor request data from modules in the PCS and converts it into individual motor outputs.

6. Implementing the *mpFCS*

This section explains the implementation steps of the *mpFCS* stated in Section 5. First, we explain how we implement the structure, and then we continue with verification of the implementation.

To implement modules, we analyzed various types of existing software for UAVs with different layers and modules. We choose ArduCopter, part of ArduPilot flight control software suites, an advanced open-source software autopilot system for multicopters, helicopters, and other rotors vehicles. Internal functionalities of our implementation are based on the open source, flight control software ArduCopter [18].

ArduCopter consists of different layers, and it provides libraries that are responsible for its flight, device drivers, and communication. For example, GCS_Copter and GCS_MAVLink in the application layer are responsible for sending a state message to the communication module and receiving a command message. GCS_MAVLink in the shared library provides a possibility to send and receive messages between GCS and UAV, and it uses MAVLink protocol on its communication. The hardware abstraction layer provides interfaces for various devices, including sensors, boards, etc. The flight codes in the application layer provide possibilities for UAV's control and management of its movement.

The FACE architecture provides its requirement for each segment and interfaces between segments. To meet the requirements, we follow the matching process described in Section 5. This process provides the matching table of the software with FACE segments. Table 4 shows the codes of ArduCopter and our matching of the software with the FACE segments. We matched the basic modules of the ArduCopter into the proposed structure. PCS consists of five modules (flight mode, position, attitude, motor, and main control) that are the main modules of the ArduCopter. In the PSSS, we choose the most basic modules related to the data of IOSS, which are communication, sensor, and transmitter modules.

Table 4. Modules of flight control software and their matching of the FACE segments.

FCS	**Segment**	***mpFCS***
File/Library Name	**Name**	**Module Name**
Flight mode	PCS	Flight Mode Control
Position		Position Control
Attitude		Attitude Control
Motors		Motor Control
Main Flight Code		Main Control
Main Flight Code	TSS	Transport Module
MAVLink		
MAVLink	PSSS	Communication Module
Sensor		Sensor Module
RC		Transmitter Module
HAL	IOSS	IO Module

6.1. Implementation of the Portable Modules

We implemented the structure with modules that consist of portable component segments (PCS), transport services segment (TSS), platform-specific services segments (PSSS), an input-output services segment (IOSS), and the core functions of each segment, which are responsible for sending and receiving a message.

In order to implement the structure with the FACE segment, we performed the following steps:

1. Identify code dependency and its coupling with different functions in other modules.
2. Match libraries and modules with the FACE segments.
3. Match its proprietary interface with FACE interfaces.
4. Mapping FACE IDL interface files into C++.

We implemented each segment and interface based on the reference implementation guide for FACE architecture [35].

To implement modules, we divided messages between modules into two main categories: internal and external messages. First, the external message is related to GCS, which can be a raw command message sent by GCS (a message from bottom modules to upper modules) to UAV or a UAV real state message sent to GCS (a message from top modules to bottom modules). Second, the internal message is related to the sensors and motors, which can be a real command message that goes to the UAV motor and actuator (a message from top modules to bottom modules) or a raw state message from the sensor, which needs to be calculated (a message from bottom modules to upper modules).

In order to send and receive the internal and the external message, we implemented the following interfaces:

1. IO interfaces such as FACE::IO::Write and FACE::IO::Read to transfer a message between the IO module and modules in the PSSS and
2. FACE::TS::Send_Message and FACE::TS::Receive_Message to transfer a message between modules in the PSSS, TSS, and PCS [35].

For implementing the structure, this paper uses Ubuntu Linux 16.04 64-bit operating system and GCC v7.5.0 as compiler and linker.

6.2. Verification

To prove our implementation works faultlessly and messages between modules and GCS are correct, we traced, collected, and compared the *mpFCS* and the general FCS data. We tested the following test case: (1) external message test (a message to/from GCS), (2) internal message test (a message to/from motor/sensor). A state message on GCS must be equal to the state data that the main flight module sends, and a command message on the flight module must equal a message that GCS sends.

To check and compare the specification of portable modules, we run both the general software and the *mpFCS* in the simulated environment. Figure 9 shows the process of comparing the general software and the *mpFCS* in the simulation environment. The simulation environment allows the software to run on the PC without any special hardware. The *mpFCS* runs on the simulation environment, which runs on the system with Intel i5-10400F 2.9 GHz CPU and 12 GB main memory, and Ubuntu 16.04 LTS-64bit OS. On the other hand, we run the GCS on the second system with Intel i5-4590 3.3 GHz CPU and 8 GB main memory, and Ubuntu 16.04 LTS-64bit OS to control and exchange data between GCS and the *mpFCS*. The *mpFCS* connects to GCS via TCP. Simultaneously, we run the general FCS in the simulation environment and connect it to GCS on another system with the exact specification. When communications are established, we make the test for sending and receiving data. GCS sends a command message to both simulators and receives a state message from both simulators.

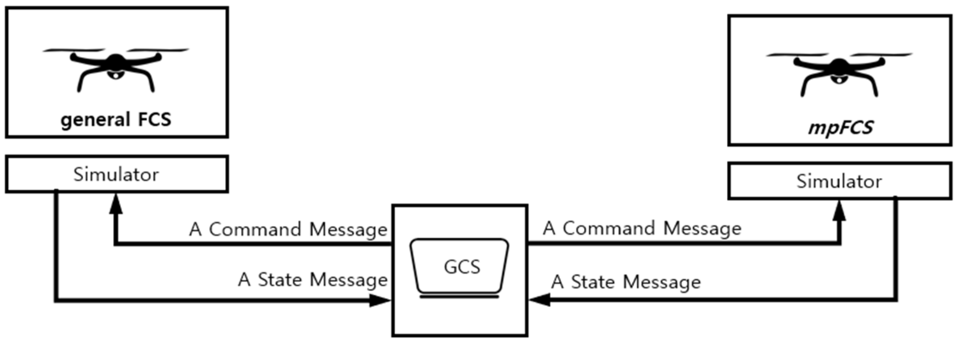
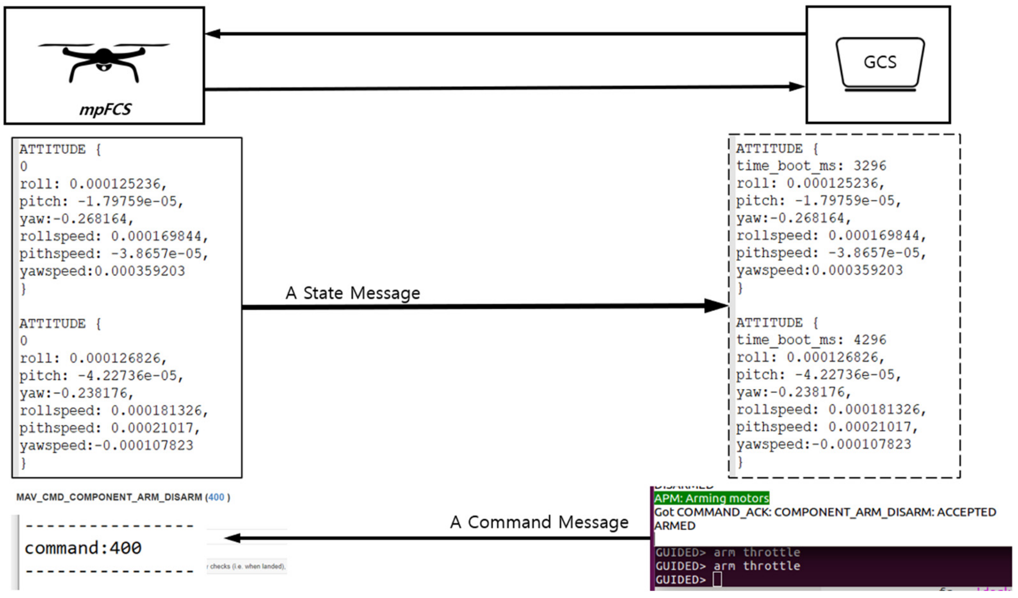


Figure 9. The process of the compared running of original FCS and *mpFCS*.

To verify the external message, first, we traced a state message on the communication module and GCS. A test result of the state and command message through its communication modules is shown in Figure 10. During the flight, the communication module of the software encodes a state message and sends it to GCS. When GCS receives a state message, we compare this message to a message on the communication module. Concurrently, we send a command message to the software. Same as previous steps, when the software receives a command message, we compare this message to a message that we send to the software. The result shows that our solution works appropriately, and its final data between GCS and the simulator are precisely the same.


mpFCS
GCS
A State Message
A Command Message
MAV_CMD_COMPONENT_ARM_DISARM (400)
checks (i.e. when landed).

```
ATTITUDE {
0
roll: 0.000125236,
pitch: -1.79759e-05,
yaw:-0.268164,
rollspeed: 0.000169844,
pithspeed: -3.8657e-05,
yawspeed:0.000359203
}

ATTITUDE {
0
roll: 0.000126826,
pitch: -4.22736e-05,
yaw:-0.238176,
rollspeed: 0.000181326,
pithspeed: 0.00021017,
yawspeed:-0.000107823
}
```

```
ATTITUDE {
time_boot_ms: 3296
roll: 0.000125236,
pitch: -1.79759e-05,
yaw:-0.268164,
rollspeed: 0.000169844,
pithspeed: -3.8657e-05,
yawspeed:0.000359203
}

ATTITUDE {
time_boot_ms: 4296
roll: 0.000126826,
pitch: -4.22736e-05,
yaw:-0.238176,
rollspeed: 0.000181326,
pithspeed: 0.00021017,
yawspeed:-0.000107823
}
```

```
----------------
command:400
----------------
```

```
APM: Arming motors
Got COMMAND_ACK: COMPONENT_ARM_DISARM: ACCEPTED
ARMED
GUIDED> arm throttle
GUIDED> arm throttle
GUIDED>
```

Figure 10. A test result of the state and command message through the communication module.

To verify the internal message, we traced the internal state and command messages between modules. First, we trace the state message from the sensor to the main flight module on both software. During the flight, the sensor module of the software gets a state message and transfers it to the application layer. Second, we traced the command messages among modules. When the main flight module receives a command message, it transfers a message to the motor and actuator. We compare the input and output of each module. The result proves that the *mpFCS* works correctly, and its final data among modules are correct. The comparison of data flow between the *mpFCS* and general software is shown in Table 5. This comparison shows that FligtControl from the general FCS depends on the motor, sensor, and communication module and is coupled with them. On the other hand, modules in the PCS of *mpFCS* depend only on modules of the TSS.

Table 5. A comparison of data flow between two software.

Message Type		General FCS	*mpFCS*
External	State	FlightControl → Communication → HAL	PCS → TSS → PSSS → IOS
	Command	HAL → Communication → FlightControl	IOS → PSSS → TSS → PCS
Internal	State	Sensor → FlightCode	IOS → PSSS → TSS → PCS
	Command	FlightControl → Motor	PCS → TSS → PSSS → IOS

→ arrow indicates the direction of the data flow.

7. Experiments

This section discusses the experiment and its result. To evaluate modules, we made two experiments as (1) A test of the FACE conformance test suite (CTS) [36], for verifying conformance of modules to the FACE standard and testing its interoperability with FACE software and its module portability, (2) a test of adding a new module to the structure, for proving extendibility of the software with different modules from FACE software and other software, (3) a test of running the *mpFCS* on the Windows system, for verifying portability of the *mpFCS*. Table 6 shows our experiment environments. In order to perform experiments, we use two systems for the environments of UAV simulation and GCS, which have the same software installed on them.

Table 6. Software and hardware environment of experiments.

Software		
Operating System		Ubuntu Linux 16.04 64-bit and Windows 10 64-bit
Compiler/Linker		GCC v7.5.0 (GNU Compiler Collection)
Testing Tool		FACE CTS v2.1.1
Hardware		
PC1: GCS for UAV	CPU	i5-4590 3.30 GHz
	Memory	16 GB
	GPU	GeForce GTX 650
PC2: UAV simulator	CPU	i5-10400F 2.90 GHz
	Memory	12 GB
	GPU	GeForce GTX 1660

7.1. The FACE Conformance Test Suite Test

The FACE CTS is a testing suite for checking the FACE standard requirements of units in the FACE segments. It provides the testing environment for the FACE standard conformance, and it can test the comfortability of the FACE segments and their interfaces with the FACE software. The software module must be linked with FACE test interfaces and interfaces to run the test most likely linked against FACE test applications provided by the FACE CTS. If the compiling and linking of the software pass the test, the software code is conformant concerning the requirements tested. If the test result fails, the software is not conformant with the FACE, and the software cannot provide portability to the software. Results that test output will show in the browser.

We tested our modules with the FACE CTS, which can test PCS, PSSS, TSS, IOSS, and interfaces. We run the test on the Linux-based computer system, which can also be Windows. After configuring testing environments, we test each segment with the interfaces. To test modules, we made the following steps and configurations.

1. Create an object file of each module using the makefile,
2. Link the FACE CTS library to the object file of each module,
3. Configure the general settings for the testing environment and specific segment settings on the FACE CTS GUI.

Figure 11 shows how the FACE CTS works and its flow. After following the above steps and configuration, we compile and link each segment with the FACE CTS. The module successfully passed the FACE architecture comfortability test. The process of compiling and linking passed the test, and the test output shows detailed information. Table 7 shows the test results of each module, its interface, and its data in the environment. Each module passed FACE CTS, and the test result on FACE CTS shows that portable modules can be interoperable with the FACE software. Figure 12 shows the test result of FACE CTS. The test result includes passed or failed for each test. The result also includes the code used for the test and the log generated in performing the test. All results are shown on the web browser after the successful test.

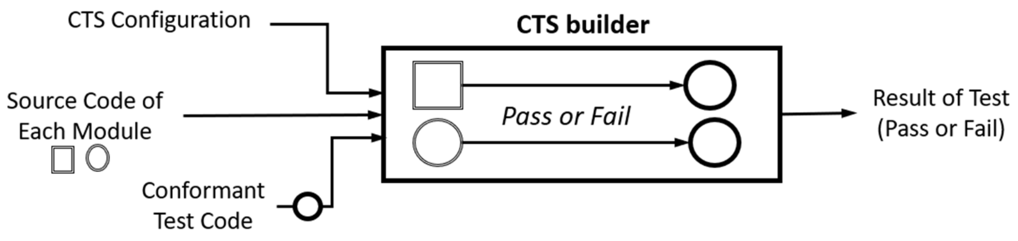
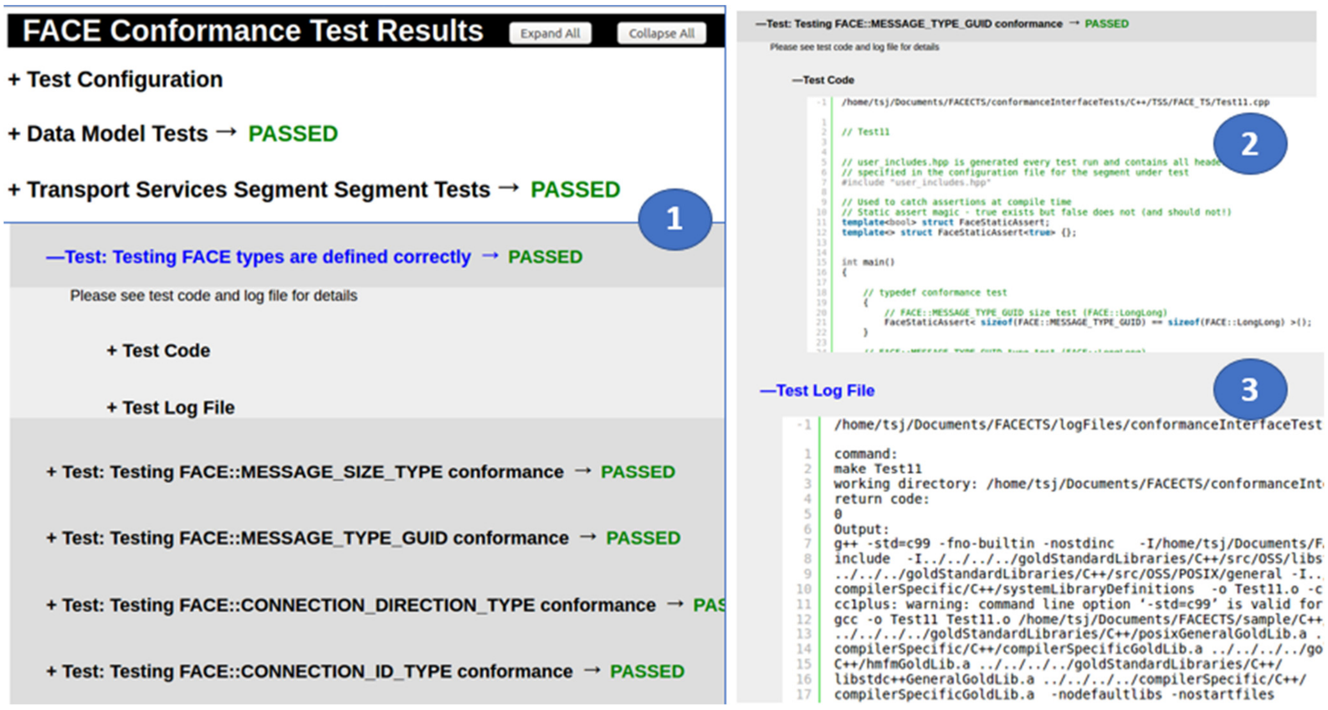


Figure 11. Specification of the FACE CTS.

Table 7. A test result of the FACE CTS.

Segment	Module	Conformances		
		Interface		Data
		TS	IO	
IOSS	IO	N/A	N/A	Passed
PSSS	Communication	Passed	Passed	Passed
	Transmitter	Passed	Passed	Passed
	Sensor	Passed	Passed	Passed
TSS	StateModule	N/A	N/A	Passed
	CommandModule	N/A	N/A	Passed
PCS	Main	Passed	N/A	Passed
	Flight	Passed	N/A	Passed
	Position	Passed	N/A	Passed
	Attitude	Passed	N/A	Passed
	Servo-motor	Passed	N/A	Passed



+ Test Configuration

+ Data Model Tests → PASSED

+ Transport Services Segment Segment Tests → PASSED

—Test: Testing FACE types are defined correctly → PASSED

Please see test code and log file for details

+ Test Code

+ Test Log File

+ Test: Testing FACE::MESSAGE_SIZE_TYPE conformance → PASSED

+ Test: Testing FACE::MESSAGE_TYPE_GUID conformance → PASSED

+ Test: Testing FACE::CONNECTION_DIRECTION_TYPE conformance → PAS

+ Test: Testing FACE::CONNECTION_ID_TYPE conformance → PASSED

—Test: Testing FACE::MESSAGE_TYPE_GUID conformance → PASSED

Please see test code and log file for details

—Test Code

```
/home/tsj/Documents/FACECTS/conformanceInterfaceTests/C++/TSS/FACE_TS/Test11.cpp

// Test11


// user_includes.hpp is generated every test run and contains all heade
// specified in the configuration file for the segment under test
#include "user_includes.hpp"

// Used to catch assertions at compile time
// Static assert magic - true exists but false does not (and should not!)
template<bool> struct FaceStaticAssert;
template<> struct FaceStaticAssert<true> {};


int main()
{

    // typedef conformance test
    {
        // FACE::MESSAGE_TYPE_GUID size test (FACE::LongLong)
        FaceStaticAssert< sizeof(FACE::MESSAGE_TYPE_GUID) == sizeof(FACE::LongLong) >();
    }

```

—Test Log File

```
/home/tsj/Documents/FACECTS/logFiles/conformanceInterfaceTest

command:
make Test11
working directory: /home/tsj/Documents/FACECTS/conformanceInt
return code:
0
Output:
g++ -std=c99 -fno-builtin -nostdinc   -I/home/tsj/Documents/F
include  -I../../../../goldStandardLibraries/C++/src/OSS/libs
../../../goldStandardLibraries/C++/src/OSS/POSIX/general -I..
compilerSpecific/C++/systemLibraryDefinitions  -o Test11.o -c
cc1plus: warning: command line option '-std=c99' is valid for
gcc -o Test11 Test11.o /home/tsj/Documents/FACECTS/sample/C++
../../../../goldStandardLibraries/C++/posixGeneralGoldLib.a .
compilerSpecific/C++/compilerSpecificGoldLib.a ../../../../go
C++/hmfmGoldLib.a ../../../../goldStandardLibraries/C++/
libstdc++GeneralGoldLib.a ../../../../compilerSpecific/C++/
compilerSpecificGoldLib.a  -nodefaultlibs -nostartfiles
```

Figure 12. The test result of CTS is shown in the web browser. (1) results of the FACE CTS, (2) the code used for the test, and (3) test log file generated in performing the test.

7.2. Adding a Module into the mpFCS

The software's expandability is the ability that increases the software's functionality by adding or porting new modules. The module extends the software by adding new functionality or modifying existing functionality, and it must extend the software without ever modifying the core base code. Since we focused on the expandability of the software, we adhere the standard architecture and exploit its interfaces on the *mpFCS*; thus, the *mpFCS* can expand with any module that uses the same interfaces. In order to port a new module in the PCS, a module needs to use the TS-Interface. Additionally, modules in the PSSS must use TS-Interfaces and IO-Interfaces. Thus, to prove this, we developed a sample module, ported it into the software based on the proposed structure, and extended it to show its expandability. Additionally, we added a module from the existing software and showed how to add the existing module into the structure.

7.2.1. Adding a New Module into the *mpFCS*

A sample module named distCalc module receives the current navigation data, and it calculates and shows the percentage of the distance between the endpoint and the UAV. Figure 13 shows the general design of the module and its interface to the transport module. In order to work as a portable module, we located the module in the PCS. To port the module into the software and receive navigation data from the transport module, it must use the TS interface. Thus, we added distanceCal in the distCalc module, and it invokes FACE::TS::Send_Message from the stateModule in the transport module to receive a message. The module receives data from the transport module in every set of time intervals and calculates the distance. Table 8 shows the specification of the module and sample data collected during its running. To execute and port the module, first, we run the *mpFCS* in the simulated environment. When the software runs, the distCalc receives data and calculates it. At first, the module receives the coordination of the starting point that is $S(x,\ y)$ (S—the starting point, x—longitude, y—latitude) and endpoint data: $E(x,\ y)$ (E—the endpoint) from TSS. Then, the module calculates the distance between two points. Second, the module calculates and shows the percentage between the endpoint and the current location when the UAV moves. In order to calculate this, the module collects the current navigation data that is a new starting point data $S_i\ (x, y)$, which changes in a set of time intervals from TSS.

Table 8. A specification of distCalc and sample data.

Specification	Input	Output
Main Specification	Starting Point	Percentage to the endpoint from current location
	Navigation Data	
	Endpoint (constant data)	
Endpoint $E(x, y)$	47.9222,106.9183	
Starting point $S\ (x, y)$	47.9178, 106.9192	100%
A New Starting Point (it changes in a set of time intervals) $S_i\ (x, y)$	47.9182, 106.9191	90%
	47.9186, 106.9191	80%
	47.9193, 106.9189	70%
	47.9200, 106.9188	60%
	47.9201, 106.9187	50%
	47.9203, 106.9187	40%
	47.9205, 106.9187	30%
	47.9212, 106.9185	20%
	47.9217, 106.9184	10%
	47.92225,106.9183	0% Reached

E: Endpoint, S: Starting Point, S_i : Current Location, x—Longitude, y—Latitude.

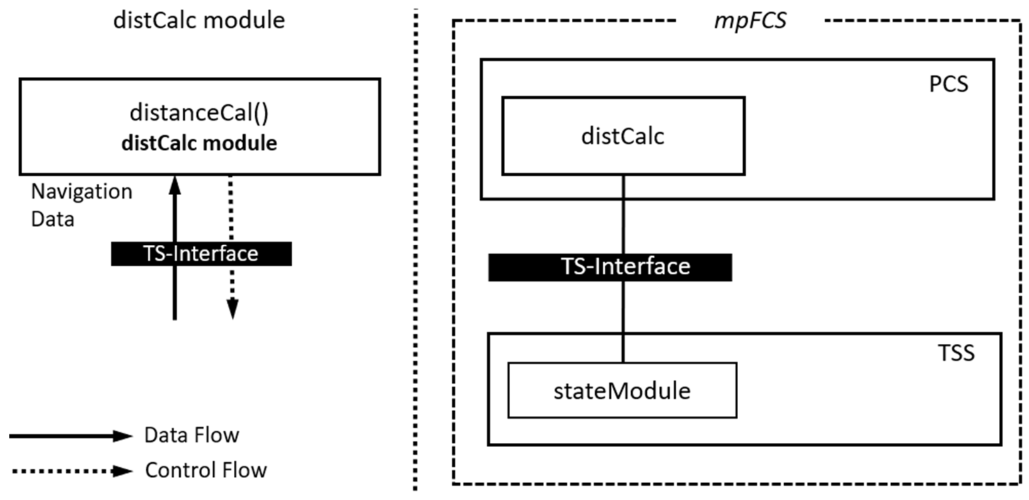


Figure 13. distCalc module and its location in the *mpFCS*.

After developing the module, we tested it on the FACE CTS. To test a module, we followed the steps mentioned in the FACE CTS test. The distCalc module must pass the test of the TS interface, and its data must be portable with the FACE standard. The module passed the conformance test, and its result is shown in Table 9.

Table 9. The test result of distCalc module on the FACE CTS.

Segment	**Module**	**Conformance**	
		TS Interface	**Data**
PCS	distCalc	Passed	Passed

Same as adding distCalc into the structure, the *mpFCS* can expand with any different modules that use the standard FACE interfaces.

7.2.2. Adding an Existing Module into *mpFCS*

The *mpFCS* can be expanded by adding a module from existing software. In order to add an existing software module to the structure, its proprietary interfaces must be replaced with the FACE interfaces. However, due to different data structures and functions, this process requires manual work for each module. Thus, we created a manual process named Interface and Data Translator, which analyzes a module from existing software and provides it for the *mpFCS*. Figure 14 shows a flow of the process. This process consists of two phases: (1) analyzing phase, which analyzes the module from the existing software, and (2) translating phase that replaces interfaces and changes data structure. The result of this process can provide a new module for the structure from the existing software. We followed the above process, converted the module to the *mpFCS*, and added the module into the structure.

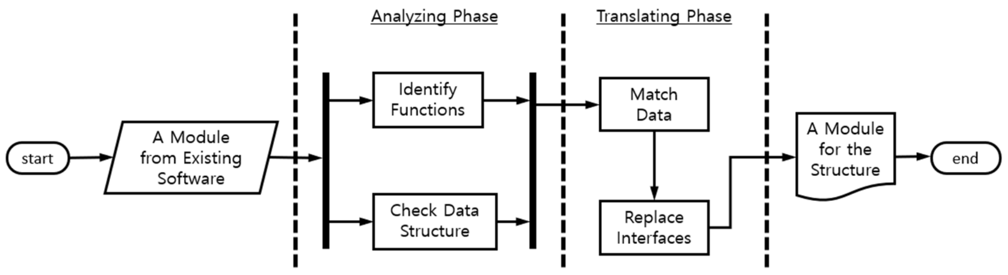


Figure 14. The flow of the Interface and Data Translator.

7.3. Running the mpFCS in a Different Environment

Since our approach follows the standard FACE architecture and its interfaces along with programming interfaces, the compiler guarantees portability. It also allows the implemented structure to run on different environments seamlessly. The process of deploying of a FACE-compatible software to any environment is shown in the Figure 15.

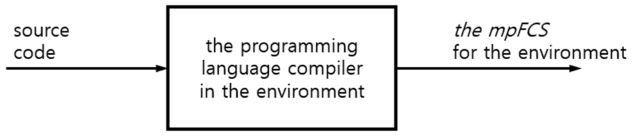
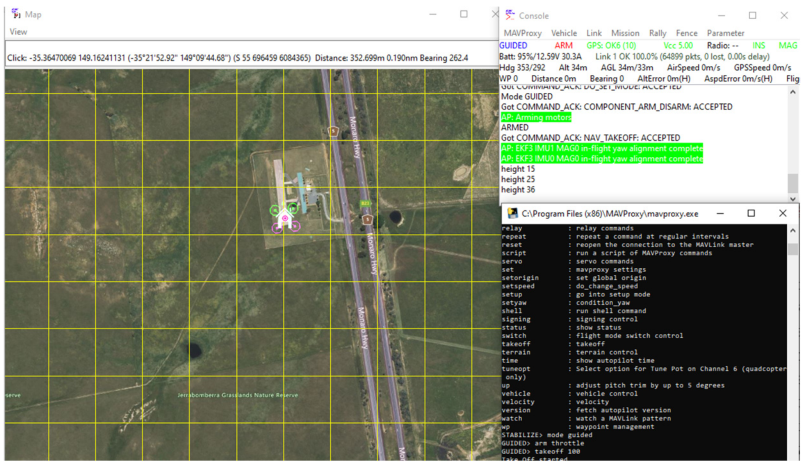

source code
the programming language compiler in the environment
the mpFCS for the environment

Figure 15. The deploying process of the *mpFCS* into any environment.

We implemented the mpFCS in Ubuntu Linux 16.04 64-bit operating system. To verify the portability of the mpFCS, we recompile and deploy the mpFCS on a Windows system that has the same hardware specifications we used in our experiments. Figure 16 illustrates the running of the mpFCS on a Windows system without any issues. Additionally, the distCalc module is successfully ported into the mpFCS that runs on the Windows system. The distCal module works correctly, and its data is the same as on the Ubuntu system. We only present the result acquired in Linux system because the results are the same on both environments.


Map
View
Monaro Hwy
Jerrabomberra Grasslands Nature Reserve
Console
MAVProxy
Vehicle
Link
Mission
Rally
Fence
Parameter
C:\Program Files (x86)\MAVProxy\mavproxy.exe

```
Click: -35.36470069 149.16241131 (-35°21'52.92" 149°09'44.68") (S 55 696459 6084365)  Distance: 352.699m 0.190nm Bearing 262.4
```

```
GUIDED        ARM    GPS: OK6 (10)       Vcc 5.00     Radio: --     INS     MAG
Batt: 95%/12.59V 30.3A     Link 1 OK 100.0% (64899 pkts, 0 lost, 0.00s delay)
Hdg 353/292     Alt 34m     AGL 34m/33m     AirSpeed 0m/s     GPSSpeed 0m/s
WP 0     Distance 0m     Bearing 0     AltError 0m(H)     AspdError 0m/s(H)     Flig
Got COMMAND_ACK: DO_SET_MODE: ACCEPTED
Mode GUIDED
Got COMMAND_ACK: COMPONENT_ARM_DISARM: ACCEPTED
AP: Arming motors
ARMED
Got COMMAND_ACK: NAV_TAKEOFF: ACCEPTED
AP: EKF3 IMU1 MAG0 in-flight yaw alignment complete
AP: EKF3 IMU0 MAG0 in-flight yaw alignment complete
height 15
height 25
height 36
```

```
relay          : relay commands
repeat         : repeat a command at regular intervals
reset          : reopen the connection to the MAVLink master
script         : run a script of MAVProxy commands
servo          : servo commands
set            : mavproxy settings
setorigin      : set global origin
setspeed       : do_change_speed
setup          : go into setup mode
setyaw         : condition_yaw
shell          : run shell command
signing        : signing control
status         : show status
switch         : flight mode switch control
takeoff        : takeoff
terrain        : terrain control
time           : show autopilot time
tuneopt        : Select option for Tune Pot on Channel 6 (quadcopter
 only)
up             : adjust pitch trim by up to 5 degrees
vehicle        : vehicle control
velocity       : velocity
version        : fetch autopilot version
watch          : watch a MAVLink pattern
wp             : waypoint management
STABILIZE> mode guided
GUIDED> arm throttle
GUIDED> takeoff 100
Take Off started
```

Figure 16. The running of the *mpFCS* on the Windows system in a simulated environment.

7.4. Analysis

This subsection discusses a comparison of the *mpFCS* and the existing flight control software on their module level. This comparison is based on its expandability, interoperability, and portability.

In order to make software portable, expandable, and interoperable, we need to use standard interfaces. Thus, we use the standard interfaces in each module of segments to send and receive a message between them. In the experimentation, a new module uses the TS interface to receive a message from the transport module in TSSS.

7.4.1. Analysis of Portability

Figure 17 compares layers and interfaces between the existing software and our structure experimentation. The existing software uses its proprietary interfaces for its internal communication to transfer data between modules. PaparazziUAV uses functions from the ABI middleware, which has prefixes such as abiSendMsg and abiBindMsg depending on the message types. dRonin also uses proprietary UAVObj interfaces, and depending on message types, different interfaces of UAVObj are used for transferring messages. uORB messaging API is used in PX4 for internal communication. On the other hand, the *mpFCS* uses the FACE standard interfaces in its modules, such as IO interfaces (FACE::IO::Write() and FACE::IO::Read()) and TS interfaces (FACE::TS::Send_Message() and FACE::TS::Receive_Message()) to transfer data between its modules. It brings portability not only to software, but also to the software modules.

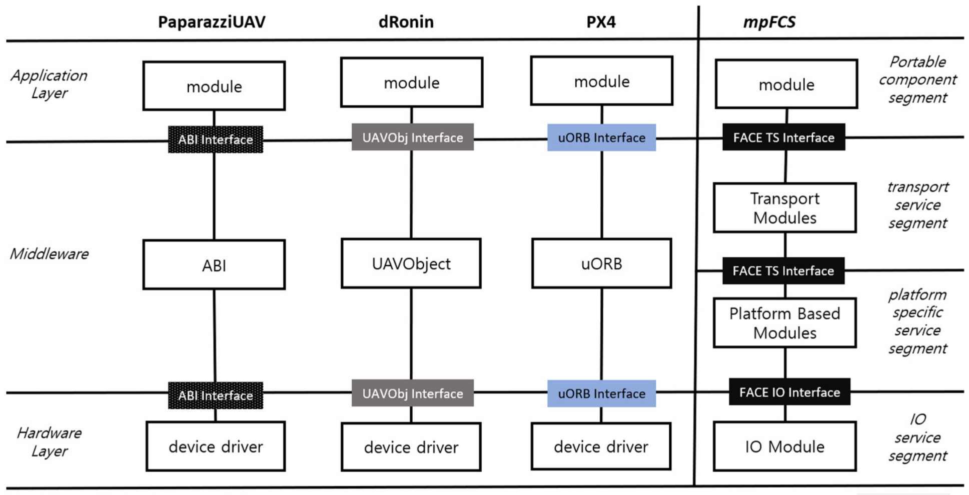


Figure 17. A comparison on layers and interfaces of related work and the *mpFCS*.

7.4.2. Analysis of Expandability

Expandability allows the addition of new capabilities or functionality to the software. The existing software uses its interfaces defined by the software. To expand the software with a user-defined module, the developer needs to understand the structure of the software. To add a new module to PX4 [19], the developer needs to know how its middleware uORB works and its data structure. However, a module works only on the chosen software. Expanding the software with another module from different software takes time, and a large amount of work is required to modify its source code to expand the software with that module. We expand our structure with a user-defined module named distCalc that calculates the distance between the current point and endpoint. The distCalc can expand any other software that runs in the same environment. On the other hand, the *mpFCS* can be expanded by different modules from different FACE software because of the same interfaces. In order to expand the *mpFCS* by adding a module from existing software, we created a manual process named Interface and Data Translator, as shown in Figure 14.

7.4.3. Analysis of Interoperability

Interoperability allows the capability of two or more functional units to process data cooperatively. A new user-defined module needs to interoperate with other modules of the software. Same as expandability, interoperability with other modules from different software requires lots of modification on the software's source code. A new module, distCalc, expands the *mpFCS* and uses the FACE standard interfaces (FACE::TS::Receive_Message) to interoperate stateModule in the Transport Service Segment (TSSS). The current capability of the distCalc is only receiving data from TSSS, but capability can increase by sending data to TSS. Due to the FACE interfaces in a new module, the module can interoperate with any software that runs on the FACE environment.

We compared the interfaces, expandability, interoperability, and portability of *mpFCS* with other software, and Table 10 shows the comparison result. The existing software [2,6–19] is not portable, and its modules are tightly coupled due to its unique interfaces. Expanding the existing software with a new module requires understanding its structure and interfaces. However, a new module only expands the chosen software and interoperates its modules.

Table 10. Comparison of the existing FCS and the *mpFCS*.

FCS Name	Data Transfer (Interface)	Expandability	Interoperability	Portability	
				SW	ML
PaparazziUAV [20]	ABI	Yes *	Yes *	No	No
PX4 [19]	uORB	Yes *	Yes *	No	No
dRonin [22]	UAVObject	Yes *	Yes *	No	No
He [16]	N/A	No	No	No	No
Yi [17]	N/A	N/A	N/A	No	No
Chon [2]	N/A	N/A	N/A	No	No
Hedge [10]	N/A	No	No	No	No
Theile [23]	cpsCore	Yes *	Yes *	No	No
mpFCS	The FACE interface	Yes **	Yes **	Yes	Yes

*—A new module works only on that software; **—*mpFCS* can expand and interoperate with different modules of the FACE software.

The experiment result shows that each module of the *mpFCS* can be ported to other FACE software, which means modules can be ported to any software that runs on the FACE software environment. After porting a module to another software, it can exchange data with them, which means it is interoperable with another software. Compared with the related work (Table 10), our structure not only provides portability for its module but also provides the expandability of the software and interoperability with different software.

7.5. Discussion

The verification result shows that the *mpFCS* works correctly, and experiments show it can provide portability for its software modules to run in the standard FACE environment. Furthermore, experiment results show that it can interoperate with other FACE software and expand with new modules that use FACE interfaces.

Currently, we designed and implemented only the basic modules of the software, but these modules are enough to control the UAV by sending/receiving a message to/from GCS. Additionally, other payloads in the message or an increase of commands will not affect the structure. Since we use the FACE architecture in our solution, more work would be required to port the module with different software types that run in a different environment.

8. Conclusions

UAV flight control software handles the movement of UAVs and is responsible for controlling and monitoring it. Unfortunately, the software does not provide portability for its module due to the tight coupling with different modules. This paper proposes *mpFCS*, a new structure of UAV flight control software, which can provide portability to its modules in the environment of standard airborne software. To evaluate the structure and modules, we redesigned existing UAV flight control software and tested them on the FACE CTS. Then, we compared their specification and functionalities with the original software. Redesigned modules passed FACE CTS testing, and the comparison shows that they are working correctly. Additionally, we developed a sample module named distCalc and ported the module into our implementation of the proposed structure. The experimentation shows that the software based on the proposed structure provides portability to its modules, and it can interoperate with the software that runs on the standard airborne software environment. The software is expandable as long as a new module uses standard airborne software interfaces. Health monitor and fault management is a critical software module for UAV which detects, reports, and handles faults in the system and its modules. Based on these standardized methods, developers can create their health and fault management system module to cover fault tolerance of the system. The Interface and Data Translator is required to port the module with different types of software in the standard airborne software environment. Future works are needed to focus on running the structure implementation on the actual UAV and bringing the automatic process to Interface and Data Translator.

Author Contributions: Conceptualization, T.J.; data curation, T.J.; methodology, T.J. and K.L.; supervision, Y.-K.J.; writing—original draft preparation, T.J.; writing—review and editing, S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This study was carried out with the support of 'R&D Program for Forest Science Technology (Project No. 2021344A00-2123-CD01)' provided by Korea Forest Service (Korea Forestry Promotion Institute), and this work was supported by the Technology Innovation Program (or Industrial Strategic Technology Development Program, 20005378, Open Avionics System Architecture and Software Development for Small to Medium Aircraft Class), funded By the Ministry of Trade, Industry, & Energy (MOTIE, Korea). Additionally, this work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1A2C1014163).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cai, G.; Dias, J.; Seneviratne, L. A survey of small-scale unmanned aerial vehicles: Recent advances and future development trends. *Unmanned Syst.* **2014**, *2*, 175–199. [CrossRef]
2. Chong, M.; Chuntao, L. Design of flight control software for small unmanned aerial vehicle based on VxWorks. In Proceedings of the 2014 IEEE Chinese Guidance, Navigation and Control Conference, Yantai, China, 8–10 August 2014; pp. 1831–1834. [CrossRef]
3. Research and Markets. *Report, Target Drone Market-A Global and Regional Analysis: Focus on End-User, Application, Platform, Mode of Operation, Speed, Target Type, Payload, and Country-Analysis and Forecast*; Research and Markets: Dublin, Ireland, 2021.
4. Bolkcom, E.B.A.C. *Unmanned Aerial Vehicles: Background and Issues for Congress*; Library of Congress, Congressional Research Service: Washington, DC, USA, 2011.
5. Battsengel, G.; Geetha, S.; Jeon, J. Analysis of technological trends and technological portfolio of unmanned aerial vehicle. *J. Open Innov. Technol. Mark. Complex.* **2020**, *6*, 48. [CrossRef]
6. Darwante, S.; Kadam, A.; Talele, H.; Ade, O.; Bankar, A. Border Surveillance Monitoring Application. In Proceedings of the 2019 5th International Conference On Computing, Communication, Control And Automation (ICCUBEA), Pune, India, 19–21 September 2019; pp. 1–6. [CrossRef]
7. Quigley, M.; Goodrich, M.A.; Griffiths, S.; Eldredge, A.; Beard, R.W. Target acquisition, localization, and surveillance using a fixed-wing mini-UAV and gimbaled camera. In Proceedings of the 2005 IEEE international conference on robotics and automation, Barcelona, Spain, 18–22 April 2005; pp. 2600–2605. [CrossRef]
8. Beard, R.W.; McLain, T.W.; Nelson, D.B.; Kingston, D.; Johanson, D. Decentralized cooperative aerial surveillance using fixed-wing miniature UAVs. *Proc. IEEE* **2006**, *94*, 1306–1324. [CrossRef]
9. Iscold, P.; Pereira, G.A.; Torres, L.A. Development of a hand-launched small UAV for ground reconnaissance. *IEEE Trans. Aerosp. Electron. Syst.* **2010**, *46*, 335–348. [CrossRef]
10. Hegde, M.; Raveendra, M. Development of flight control software for Unmanned Aerial Vehicle. *Int. J. Emerg. Technol. Comput. Sci. Electron.* **2015**, *14*, 661–664.
11. Mooney, J.D. Portability and reusability: Common issues and differences. In Proceedings of the 1995 ACM 23rd Annual Conference on Computer Science, Nashville, TN, USA, 28 February–2 March 1995; pp. 150–156.
12. Singh, C.; Sharma, N.; Kumar, N. Analysis of software maintenance cost affecting factors and estimation models. *Int. J. Sci. Technol. Res.* **2019**, *8*, 276–281.
13. Ren, Y. Research on Software Cost Estimation and Its Expert System. Ph.D. Thesis, Liaoning Technical University, Fuxin, China, 2008.
14. Hopsu, A. *Portability of IEC 61499 Compliant Software*; Aalto University: Espoo, Finland, 2019.
15. Beningo, J. A Practical Approach to Code Reuse. In *Reusable Firmware Development*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 277–299.
16. He, Z.; Chen, Y.; Shen, Z.; Huang, E.; Li, S.; Shao, Z.; Wang, Q. Ard-mu-copter: A simple open source quadcopter platform. In Proceedings of the 2015 11th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN), Shenzhen, China, 16–18 December 2015; pp. 158–164. [CrossRef]
17. Ying, F.; Xiujuan, L.; Chuntao, L.; Zhenyu, J. Design of distributed flight control software based on software bus. In Proceedings of the 2016 IEEE Chinese Guidance, Navigation and Control Conference (CGNCC), Nanjing, China, 12–14 August 2016; pp. 725–729. [CrossRef]
18. ArduPilot. Available online: https://ardupilot.org/ (accessed on 20 January 2021).
19. DroneCode. PX4. Available online: https://px4.io (accessed on 20 January 2021).
20. ENAC. PaparazziUAV. Available online: https://wiki.paparazziuav.org/wiki/Main_Page (accessed on 20 January 2021).
21. MultiWii. Available online: http://www.multiwii.com/ (accessed on 20 November 2021).
22. dRonin. Available online: https://dronin.org (accessed on 14 January 2021).

23. Theile, M.; Dantsker, O.; Nai, R.; Caccamo, M.; Yu, S. uavAP: A Modular Autopilot Framework for UAVs. In Proceedings of the AIAA Aviation 2020 Forum, Virtual Event, 15–19 June 2020; p. 3268. [CrossRef]
24. OpenGroup. The FACE Technical Standard. Available online: https://www.opengroup.org/face (accessed on 20 February 2020).
25. *ISO/IEC 25010*; 2011 Systems and Software Engineering@ Systems and Software Quality Requirements and Evaluation (SQuaRE)@ System and Software Quality Models. 2013; ISO: Geneva, Switzerland. Available online: https://www.iso.org/standard/35733.html(accessed on 14 June 2021).
26. Mooney, J.D. Developing portable software. In *Information Technology*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 55–84.
27. Kaindl, H. Portability of software. *ACM SIGPLAN Not.* **1988**, *23*, 59–68. [CrossRef]
28. ECSS-E-40-Part-1B, Space Engineering: Software—Part 1 Principles and Requirements. 2003. Available online: https://ecss.nl/standard/ecss-e-40-part-1b-space-engineering-software-part-1-principles-and-requirements/ (accessed on 14 June 2021).
29. *IEEE-Std-830*; IEEE Recommended Practice for Software Requirements Specifications. IEEE Press: New York, NY, USA, 1998.
30. *ISO 2382–1*; Information Technology—Vocabulary—Part 1: Fundamental Terms. 1993; ISO: Geneva, Switzerland. Available online: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-1:ed-3:v1:en(accessed on 14 June 2021).
31. Haile, N.; Altmann, J. Evaluating investments in portability and interoperability between software service platforms. *Future Gener. Comput. Syst.* **2018**, *78*, 224–241. [CrossRef]
32. Ebeid, E.; Skriver, M.; Terkildsen, K.H.; Jensen, K.; Schultz, U.P. A survey of open-source UAV flight controllers and flight simulators. *Microprocess. Microsyst.* **2018**, *61*, 11–20. [CrossRef]
33. Dupus, A. Multiwii Source Code. Available online: https://code.google.com/archive/p/multiwii/ (accessed on 20 December 2020).
34. OpenPilot. Available online: https://opwiki.readthedocs.io/en/latest/ (accessed on 15 December 2020).
35. OpenGroup. Reference Implementation Guide for FACE™ Technical Standard, Edition 2.1. Available online: https://www.opengroup.org/face/docsandtools (accessed on 20 September 2020).
36. OpenGroup. The FACE Conformance Test Suite. Available online: https://www.opengroup.org/face/conformance-testsuites (accessed on 11 November 2020).