IMPLEMENTING CONTROL AND MISSION SOFTWARE OF UAV BY EXPLOITING OPEN SOURCE SOFTWARE-BASED ARINC 653

Hyun-Chul Jo  Sanghyun Han  Sang-Hun Lee  Hyun-Wook Jin
Department of Computer Science and Engineering, Konkuk University, Seoul, Korea

Abstract
The Integrated Modular Avionics (IMA) architecture has been suggested to address the Size, Weight, and Power (SWaP) issues and provide better software consolidation and testability by means of partitioning. Though the IMA architecture is mainly discussed from the view point of large aircrafts or manned aerial vehicles, small Unmanned Aerial Vehicles (UAV) are one that indeed requires IMA to reduce SWaP [2]. In addition, as civilian UAVs are getting popular, a wider spectrum of UAV applications will be available, which emphasizes the importance of software reusability and seamless consolidation that can be provided by IMA.

ARINC-653 defines essential features for IMA and standardized Application Programming Interfaces (APIs) [3]. Though we can use commercial implementations of ARINC-653 for UAVs, open source software also has significant potential for providing IMA features for civilian UAVs such as university-operated UAVs and small UAVs. It should also be noted that open source-based software is positively considered for military UAVs. For example, US Navy announced recently that Linux was going to be employed in autonomous vertical take-off-and-landing drone system.

In this study, we design and implement UAV control and mission software over ARINC 653. Especially we utilize our Linux-based ARINC-653, which can provide abundant development tools, software libraries, and device drivers due to the nature of Linux. Our ARINC 653 implementation supports partitioning, inter-partition communication, XML-based configuration, and health monitoring. Our control and mission software include Operational Flight Program (OFP), Video Streaming Program (VSP), Ground Control Program (GCP), and Ground Monitoring Program (GMP). We test our programs in a HILS environment and show that these run correctly in terms of functionality and real-time requirements. Our study also suggests few extensions for process scheduling and inter-partition communication of ARINC 653.

Introduction
Most current-generation avionics systems are based on a federated architecture but as the number of computing devices is increasing at an astonishing rate it is very difficult to resolve issues of Size, Weight, and Power (SWaP) efficiently. The Integrated Modular Avionics (IMA) architecture has been suggested to address the SWaP issues and provide better software consolidation and testability by means of partitioning [1]. Partitioning provides an efficient way of integrating several real-time applications transparently on the same computing device while providing isolation of execution environment.

Though the IMA architecture is mainly discussed from the view point of large aircrafts or manned aerial vehicles, small Unmanned Aerial Vehicles (UAV) are one that indeed requires IMA to reduce SWaP [2]. In addition, as civilian UAVs are getting popular, a wider spectrum of UAV applications will be available, which emphasizes the importance of software reusability and seamless consolidation that can be provided by IMA.

ARINC-653 defines essential features for IMA and standardized Application Programming Interfaces (APIs) [3]. Though we can use commercial implementations of ARINC-653 for UAVs, open source software also has significant potential for providing IMA features for civilian UAVs such as university-operated UAVs and small UAVs. It should also be noted that open source-based software is positively considered for military UAVs. For example, US Navy announced recently that Linux was going to be employed in autonomous vertical take-off-and-landing drone system.

In this study, we design and implement UAV control and mission software over ARINC 653. Especially we utilize our Linux-based ARINC-653, which can provide abundant development tools, software libraries, and device drivers due to the nature of Linux. Our ARINC 653 implementation supports partitioning, inter-partition communication, XML-based configuration, and health monitoring. Our control and mission software include Operational Flight Program (OFP), Video Streaming Program (VSP), Ground Control Program (GCP), and Ground Monitoring Program (GMP). We test the whole system in a Hardware-In-the-Loop Simulation (HILS) environment. Our study can present effectiveness of Linux-based ARINC 653 for UAVs in terms of its performance and stability though the certification such as DO-178B [6] is a remaining issue. In addition, we analyze possible extensions of ARINC 653 for better implementations of UAV software.

The rest of the paper is organized as follows. Section 2 briefly summarizes ARINC 653 and related work. Section 3 describes Linux-based ARINC 653. Our design and implementation of UAV control and
mission software are presented in Section 4. In this section, we also show performance of our implementation and its implications. Finally, we conclude this paper in Section 5.

Background

Overview of ARINC 653

The IMA architecture provides an abstraction layer to provide a common execution environment over various hardware platforms and allows the applications to run safely without affecting from other applications in terms of fault and resource utilization. Consequently, it is expected that modularity, portability, and reusability of avionics software can be achieved with the IMA architecture.

The ARINC standards include the vast scope of the air transport avionics equipment and systems. Among them, the standard number 653, called ARINC 653, gives the standardized guideline to implement the IMA architecture, which includes the general-purpose APEX (APplication/EXecutive) interfaces between the operating system of an avionics computer and the application software [3]. The interfaces allow the application software to control the scheduling, communication, and status information of its internal processing elements.

ARINC 653 defines the partition that enables one or more avionics applications to execute independently from each other in terms of memory and processor resources. This partitioning concept is a key for IMA architecture as it provides isolation between applications. The partition code executes in user mode only. This model prevents a fault caused by an application propagating to others. The partitions are created at the system initialization phase and cannot be removed or added dynamically. The scheduling algorithm of partitions is predetermined, repetitive with a fixed periodicity. A partition is in one of Cold Start, Warm Start, Idle, and Normal states. A partition comprises one or more processes that share the resources of the partition but are not visible outside of the partition. Each process has a priority level and can be preempted by a higher priority process. The process scheduling can be either periodic or aperiodic based on the policy configured when each process is created. A process is in one of Dormant, Ready, Waiting, and Running states.

ARINC 653 also defines communication interfaces for inter- and intra-partition communication. The inter-partition communication interfaces provide communication between two or more partitions that can run either on the same node or different nodes. To specify a connection between two or more partitions, ARINC 653 uses a 2-tuple composed by channel and port. A channel defines a unidirectional logical link between source and destination ports of which each provides required resources that allow a specific partition to either send or receive messages. Both channel and port are defined statically by the configuration file. The inter-partition communication supports two communication modes: queuing and sampling. In the queuing mode, the messages are queued in the internal message queue and passed to the application in FIFO manner. Thus there is no intentional message loss in this mode. On the other hand, in the sampling mode, only the last message is saved overwriting the previous one.

ARINC 653 also defines four intra-partition communication methods: buffers, blackboards, semaphores, and events. Buffers store messages in the message queue and deliver to the receiver in FIFO order. On the other hand, a blackboard does not queue messages. Any message written to a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. The semaphores are used to synchronize between processes running in the same partition. An event is a communication mechanism that notifies an occurrence of a condition to processes which are waiting for it.

Related Work

ARINC 653 is implemented in several commercial real-time operating systems, e.g., WindRiver VxWorks 653, LynuxWorks LynxOS-178, and GreenHills Integrity-178B. There are also implementations of ARINC 653 over POSIX [7][8][9]. Recently, researchers try to exploit the virtualization technology to implement the IMA architecture. VanderLeest [10] has implemented a prototype of ARINC 653 over Xen. Masmano et. al. [11] present a design of ARINC 653 over XtratuM. The AIR project tries to support real-time operating systems but also general-purpose operating systems over partition management kernel [12]. While the
previous researches are based on the para-virtualization, Han and Jin [13] suggest ARINC 653 over full-virtualization. The implementation of ARINC 653 used in this paper is a kernel-level implementation [4], which can provide less overhead and jitter. Mason et. al. [14] present design alternatives for implementing device driver for ARINC 653. Though these researches detail possible internal designs of ARINC 653, these do not include discussions about application software that runs on top of ARINC 653.

There are also several researches for designing control and mission software. Popp and Kahler [15] describe software architecture for the flight control system of military airlifter but their discussions are based on a federated architecture. Immanuel and Johnson [16] discuss about an OS-based system architecture for UAV but they also do not consider partitioning. Ferguson et. al. [17] show a very high-level picture of ARINC 653-based software for spacecrafts. Alena et. al. [18] focus on inter-partition communication over Avionics Full-Duplex Switched Ethernet (AFDX). Comparing with previous work, this paper presents implementation-level software designs for an IMA-based UAV.

**Linux-based ARINC 653**

Our study is based on a Linux-based ARINC 653 [4][5]. Its overall design is shown in Figure 1. This implementation supports partitioning, inter-partition communication, XML-based configuration, and health monitoring.

The process scheduler and partition scheduler perform at the kernel level in a hierarchical manner. The process scheduler uses the EDF algorithm while the partition scheduler uses a fixed-priority scheduling algorithm that prevents criticality inversion [19]. Each partition has a separate task queue so that switching between partitions can be done by simply changing a pointer.

The system partition includes startup manager, network manager, and error handler. The system partition runs aperiodically with the highest priority. As specified by the ARINC 653 standard, all system configurations are performed statically by using an XML file that contains detail system information about partitions, processes, and communication channels. The startup manager interprets the configuration file, creates a scheduling table with schedulability test and starts partitions running at the initialization phase. In addition, it provides the network configuration information to the network manager. The network manager implements transparent inter-partition communication over different network links [5]. The current implementation can support Ethernet, Controller Area Network (CAN), Inter-Process Communication (IPC), and Wireless LAN. The health monitor invokes the error handler of the corresponding partition when an error occurs, such as deadline miss.

The ARINC 653 library implements APEX interfaces defined by the standard. The current implementation supports interfaces for process and partition manipulation and inter-partition communication. We are planning to implement interfaces for intra-partition communication.

---

**Figure 1. Linux-based ARINC 653**

**Design and Implementation of Control and Mission Software**

**Overall Design**

The overall design of control and mission software is shown in Figure 1. As shown in the figure, each program runs as a partition. Operational Flight Program (OFP) and Video Streaming Program (VSP) run on the unmanned helicopter, while Ground Control Program (GCP) and Ground Monitoring Program (GMP) run on the ground system. GCP gets commands from a human pilot on the ground and sends them to OFP through wireless network. The
OFP physically controls the unmanned helicopter and provides auto-hovering and auto-landing modes. VSP on the helicopter sends video frames to GMP.

We port the OFP [20] implemented for Voyager GSR-260 of Japan Remote Control Co., Ltd. to our ARINC 653 platform. The target helicopter has a length of 1.4m and a height of 0.63m. Its main rotor diameter is 1.77m. The OFP consists of four tasks. The nav_reader task reads the attitude measurements, angular rates, and body from navigation sensor. Another task called swm_reader collects current state information of helicopter flight controls such as cyclic, rudder, throttle, and collective. The adt_reader task receives control commands from GCP. The GnC task controls servo motors to move the control surfaces of the unmanned helicopter in accordance with commands from GCP and sends back the current state information to GCP.

The VSP has only one process (frame_send) that sends video frames to GMP through wireless network. The camera sensor generates JPEG format frames of 160x120 pixels in 12 fps (frames per second). Each frame has around 5KB to 6KB size.

The state_recv task receives state information of unmanned helicopter from OFP. This state information is processed by the state_update task to provide roll-pitch, yaw, altitude, control power, and velocity. By using this information and Qt library, the gcp_gui task actually represents graphical user interface (i.e., glass cockpit).

The gcp_gui task has different characteristics from other tasks because it interacts with human. This task can be implemented as either periodic or aperiodic process. If this task runs periodically, it can fail to immediately respond human request. To prevent this, we can assign a very short period but then it can waste computation resources uselessly. Thus, in order to determine the optimal period of this process, we should consider periods of GnC and cmd_send tasks that provides state information of the helicopter and sends control commands to the helicopter, respectively. On the other hand, we can implement this task as aperiodic (i.e., event-driven). However, if this task has a lower priority than others, it can suffer from delay. Thus, we need to assign a higher priority to this task. Since the current Qt framework is not suitable to implement a periodic process, we implement the gcp_gui task as an aperiodic process with the highest priority. However, our ARINC 653 currently does not fully support aperiodic processes in terms of temporal partitioning; thus, we emulate a similar environment.

GMP consists of three tasks. The frame_recv task receives video frames from VSP and saves them into a buffer. These frames are packed by the state_update task using Qt library for easy playback. The gmp_gui task shows video frames in the monitor. This task is also implemented as an aperiodic process due to the same reason with the gcp_gui case.

Thanks to the partitioning, we can run several different functions on a computer realizing integrated architecture as shown in Figure 2. Otherwise, on the helicopter for example, the control software (i.e., OFP) and mission software (i.e., VSP) have to run in different computers. This results in increasing requirements for SWaP.

Software Consolidation

Table I shows period and execution time (i.e., Worst Case Execution Time; WCET) we used for each task. This information is specified in an XML
file and tested by the startup process in the system partition during the system initialization phase. Though we have decided periods and WCETs heuristically, we believe that these should be determined more systemically considering capability of sensors and actuators. As we have mentioned in the previous subsection, gcp_gui and gmp_gui are aperiodic processes.

Table I. Period and WCET of processes

<table>
<thead>
<tr>
<th>Partitions</th>
<th>Tasks</th>
<th>Period (ms)</th>
<th>WCET (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmanned Helicopter</td>
<td>GnC</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>nav_reader</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>swm_reader</td>
<td>80</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>adt_reader</td>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td>VSP</td>
<td>frame_send</td>
<td>80</td>
<td>35</td>
</tr>
<tr>
<td>GCP</td>
<td>cmd_send</td>
<td>160</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>state_recv</td>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>state_update</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>gcp_gui</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>GMP</td>
<td>frame_recv</td>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>state_update</td>
<td>160</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>gmp_gui</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

We measure the actual execution time of processes while running in Hardware-In-the-Loop Simulation (HILS) environment, of which more details will be described in a following subsection. Figures 3, 4, 5, and 6 show the measurement results of each partition. As we can see, the execution times of all processes are bounded by their WCET.

Figure 3. Excution time of processes in OFP (log scale)

Figure 4. Excution time of process in VSP

Figure 5. Excution time of processes in GCP

Figure 6. Excution time of processes in GMP

Network Configuration

As shown in Figure 2, there are three channels of which each has two ports. Channel 0 is used to
send control commands from cmd_send to adt_reader. Channel 1 is used to send current state of unmanned helicopter from GnC to state_recv. Video frames are sent from frame_send to frame_recv through Channel 2.

![Diagram of network communication](image)

Figure 7. Worst case of network delay $D$ for DATA($i$)

Figure 7 shows the worst case of data communication between two periodic processes. The receive process in the figure runs in the very first slot in a period and the last slot in the next period. Since DATA($i$) arrives right after the former receive call, it can be received by the latter one. Thus, the network delay $D$ for DATA($i$) can be represented as:

$$D = T_{NET} + T_{Idle} \cdot 2$$  \hspace{1cm} (1)

where $T_{NET}$ is the time to transmit data over the network, and $T_{Idle}$ represents the idle time in a period (i.e., $P_R - T_{EXE}$). If the execution time of receive process $T_{EXE}$ is very small compared with its period, Equation (1) can be represented as:

$$D \approx T_{NET} + P_R \cdot 2$$  \hspace{1cm} (2)

That is, in the worst case, the network data received can be recognized by the receive process after two periods.

$D$ in Equation (2) can be reduced by assigning a smaller period $P_R$ to the receive process; however, this can consume processor resources in vain because several receive calls can happen in between two data arrivals. Thus we fix the period of receive processes into a decent value of 80ms as shown in Table I.

All three channels are configured as queuing mode. However, the sampling mode might be better than queuing mode for our simple control and mission software because only fresher data is meaningful in this system. Nevertheless, we use the queuing mode to utilize the timeout feature, which is not supported in the sampling mode. If there is no data received, the queuing mode blocks the receive process until a data arrives or a timer expires. Since the computation resource is anyway reserved for the receive process, if there is no data, it can be better to wait a data little bit more relinquishing the computation resource to another process instead of giving up this period. In this way, we can improve responsiveness and reduce chances of experiencing $D$ in Equation (2). However, the receive API of sampling mode simply returns if there is no data received. Thus, the current API that follows the ARINC 653 standard is hard to provide such mechanism.

**Hardware-In-the-Loop Simulation**

We test our software and verify its functionality in a HILS environment, where the HIL simulator developed by Korea Aerospace Research Institute simulates the real world, airframe, and sensors so OFP considers that it is controlling a real unmanned helicopter. We use an industrial embedded board for flight control computer of unmanned helicopter, which is equipped with an Intel Core2Duo 2.66 GHz processor and Logitech QuickCam Pro 9000 camera. The ground system is implemented on a laptop equipped with an Intel Centrino Duo 1.66 GHz processor. We use ARINC 653 implemented in the Linux kernel 2.6.32 for both flight control computer and ground system. The HIL simulator runs on a desktop equipped with an Intel Core2Duo 2.66 GHz processor and QNX 6.3.2 operating system.

Figure 8 shows the user interface of GCP during auto-hovering. As we can see, the state of the helicopter becomes very stable by the autopilot. Figure 9 shows the user interface of GMP. In this experiment, the camera looks toward the campus of Konkuk University at the System Software Laboratory.

We also measure the variation of altitude in the auto-landing mode. Figure 10 shows the measurement results. The dotted line represents the time point where we turn on the auto-landing mode.
while the solid line shows actual values we observed. As we can see, it takes around 20 seconds to reach the expected altitude.

![Graphical user interface of ground control program](image1)

**Figure 8.** Graphical user interface of ground control program

![Graphical user interface of ground monitor program](image2)

**Figure 9.** Graphical user interface of ground monitor program

![Variation of altitude](image3)

**Figure 10.** Variation of altitude

### Conclusions

This paper suggests detail design of control and mission programs of unmanned helicopter. We implement them on Linux-based ARINC 653. We test our programs in a HILS environment and show that these run correctly in terms of functionality and real-time requirements. This study further provides insights into issues on exploiting open source software to build IMA architecture for a UAV and its potential.

Our study also suggests few extensions of ARINC 653. As we have described, GCP and GMP have both periodic and aperiodic processes. However, the current standard does not clearly define how to schedule periodic and aperiodic processes in an integrated manner. A simple way to handle this mixed-process case is to allow an aperiodic process to have only either a higher or lower priority than all periodic processes. In our system, we assign the highest priority to aperiodic tasks. To guarantee a consistent behavior of processes on different ARINC 653 implementations, the standard needs to provide a better guideline for mixed-process case.

In addition, we also have discussed about an extension of API for sampling mode communication. If the sampling mode also provides the timeout feature, it can improve responsiveness of applications without sacrificing additional computation resources. Since it is not provided in the current standard, instead we have used queuing mode communication.

As future work, we plan to determine period and execution time of processes more systematically. As we can see in measurement results, the actual execution time is much less than the configured WCET value for most of processes. In addition, harmonizing periods across processes should be explored further. In this way, we can save more computation resources for additional partitions and can guarantee real-time requirements more strictly. Moreover, we intend to apply our software to a real quad-rotor helicopter.

### Acknowledgements

This research was partly supported by Basic Science Research Program (#2012-0001671) and National Space Lab Program (#2012-0006617) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology.
References


31st Digital Avionics Systems Conference
October 14-18, 2012