DDC-I

WHITE PAPER

Maximizing Reuse in Safety-Critical Software using IOI

Overview

Developing DO-178 certifiable avionics software is a very expensive and time consuming proposition. It is not reasonable that a company develop software from scratch for every new product they develop. In order to stay competitive in this market space they need to find a way to reuse as much code as they can from system to system. Ideally, being able to reuse software at the binary level would be ideal. Reuse of software is not a new concept but in the avionics market it is can provide even more value, especially if it does not change from product to product. Minimizing change is the key to driving the cost saving to a maximum.

One of the main reasons for changing the software in avionics systems is that the I/O is volatile and changes frequently. Even in a simple upgrade situation, the I/O interface in a given avionics software application can differ significantly from the interface in the prior configuration. Managing this I/O volatility and the resulting software change impacts presents significant challenges to avionics developers and their software teams.

Deos™ is designed to enhance and enable application software portability, where binary reuse is the ultimate form of portability. Deos was originally developed for use in the aerospace industry where verification and certification costs are notoriously high. The ability to reuse executables and shared libraries without modification on new (compatible) target systems does significantly reduce costs. This binary reuse model is enabled by several capabilities including DDC-I's IO Interface (IOI) library, used in conjunction with its Deos™ Real Time Operating System (RTOS) is designed to help developers meet these challenges in a way that minimizes change impact and maximizing software reuse, while keeping cost and schedule under control.

This paper will focus on the reuse challenges of I/O and how DDC-I's IOI product provides a solution to meet these challenges.

I/O Challenges

Aircraft avionics is an I/O centric environment. There are many types of systems that are dependent on sensors all over the aircraft to provide data of various different types, formats and rates on various different busses. All of this I/O tends to change frequently. Aircraft to aircraft, platform to platform, I/O is almost always different. For example, the sources and destinations of data values can change. Further, engineering units, data types and data rates can change.

In any avionics software system, managing this volatility presents significant challenges to software developers. Specifically, how can one isolate their software from this volatility thereby minimizing change impact and maximizing software reuse, while also keeping cost and schedule under control? In the world of certifiable, safety-critical software, these considerations become even more important when one considers the impact of changes not only on the software, but also on the certification activities and artifacts, including rework of testing, reviews and analysis.

Figure 1 shows a very simple example of I/O, say on Target 1.

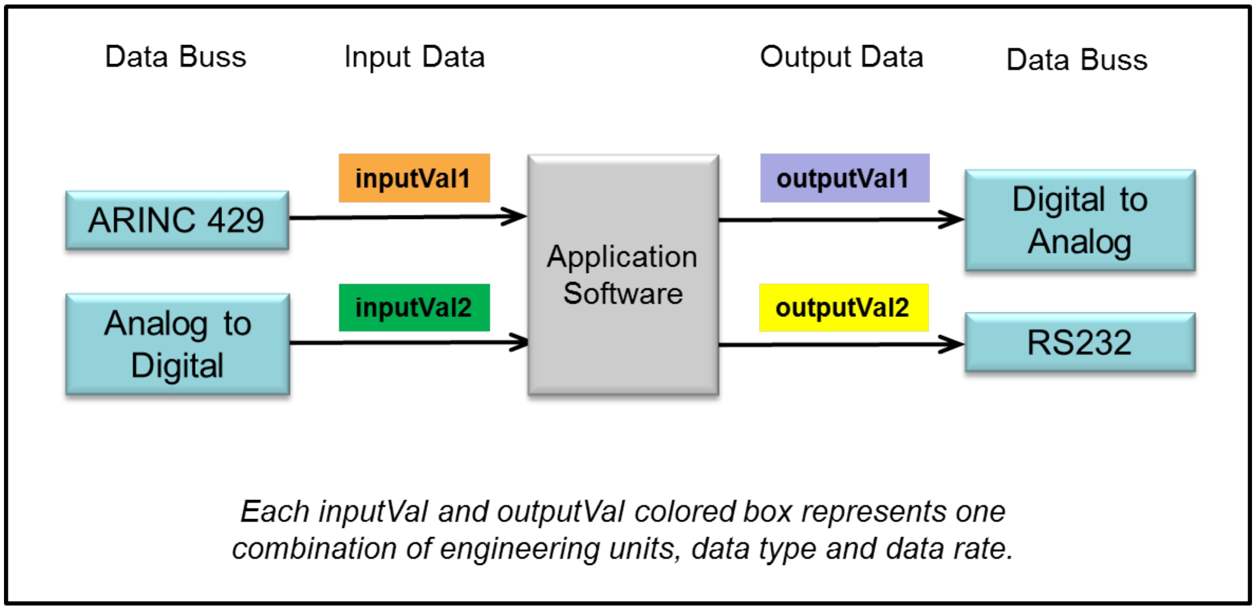

Figure 1 – I/O on Target 1

As shown, Target 1 has two input values, inputVal1 and inputVal2. Orange represents one combination of engineering units, data type and data rate for inputVal1, and green represents a second combination for inputVal2. These two values are sent from other systems in the aircraft (sensor, data concentrator, etc.) to Target 1 via ARINC 429 and an analog-to-digital converter, respectively.

Similarly, Target 1 has two output values, outputVal1 and outputVal2, where purple represents a third combination of engineering units, data type and data rate for outputVal1, and yellow represents a forth combination for outputVal2. These two values exit Target 1 via a digital-to-analog converter and RS-232, respectively.

The avionics application software application (grey) converts these inputs into these outputs. For example, it may be part of a flight control system that uses altitude and vertical airspeed inputs to compute outputs that help maintain an altitude hold flight mode.

Now, assume that the aircraft manufacturer upgrades the aircraft on which the Target 1 application software is deployed. This is often the case when upgrading an aircraft but it can also happens many times before the software gets deployed for the first time. Let's also assume that part of the upgrade involves changing I/O hardware due to improvements in devices, device availability, etc. Finally, assume that the upgraded aircraft requires the same functionality present in Target 1.

These changes result in Target 2, as shown in Figure 2.

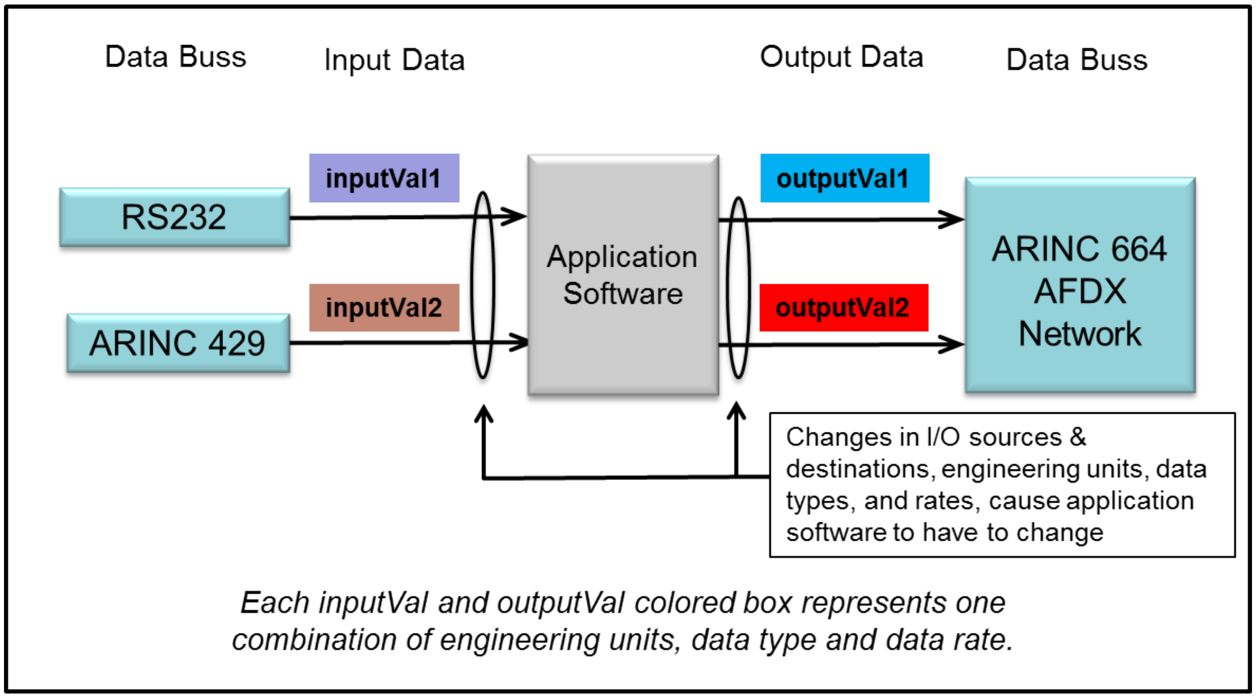
Data Buss
Input Data
Output Data
Data Buss
RS232
ARINC 429
inputVal1
inputVal2
Application Software
outputVal1
outputVal2
ARINC 664 AFDX Network
Changes in I/O sources & destinations, engineering units, data types, and rates, cause application software to have to change
Each inputVal and outputVal colored box represents one combination of engineering units, data type and data rate.

Figure 2 – I/O on Target 2

Target 2 intends to use the same application software and has the same two inputs. But here, inputVal1 is in purple and inputVal2 is in brown (not orange and green, as in Target 1) as before, this means that although the same data is being provided to the application it is being provided differently (engineering units, data types and data rate). Further, these two values enter Target 2 via RS-232 and ARINC 429 (not 429 and A-2-D, as in Target 1).

Similarly, Target 2 has the same two outputs. But here, outputVal1 is in blue and outputVal2 is in red (not purple and yellow, as in Target 1). These two values exit Target 2 via an ARINC 664 bus.

Again, the avionics application software (grey) is expected to convert these inputs into these outputs. However, the sources of the inputs differ from Target 1, as does the destination of the outputs. Further, each value's characteristics differ from Target 1 (e.g., engineering units, data type and data rate).

The question is, can the application software remain unaltered in Target 2, while still performing the same function it performed in Target 1, even though its I/O is completely different?

Data Communication via IOI

DDC-I's IOI product is a software library that provides I/O services to software applications that handles/supports:

- Periodic and Aperiodic data rates
- Data producers and data consumers that run at different rates and manages the required buffering
- The ability for data consumers to access pieces of data within a data structure, without knowledge of the entire structure.
- Sampling, queuing and blackboard behaviors (ARINC 653 API uses IOI for inter-partition communication capabilities)
  - FIFO or last produced
  - Re-read messages
  - Check message freshness
- Fixed and variable length messages
- Combined or "chained" reads and writes
  - Chaining refers to IOI's ability to read or write multiple data values via a single call from user software, even if those values come from, or go to, multiple sources or destinations.
- The ability to inject user-defined formatting functions into the data stream.
  - Formatting functions are primarily used to modify data types, perform engineering unit conversions, and so on.

These properties and more are configured via XML-based IOI configuration files that define each produced and consumed data item and its detailed characteristics. Such as:

- Name of data item (for reference)
- Data type
- Data rate (produced or consumed)

- Buffering and queue depth definition
- Process/partition that produces the data item
- Process/partition that consumes the data item
- Formatting functions for data item and weather it is used when produced or consumed.

I/O Examples

This section of the document will be used to illustrate the use cases for IOI and to further explain the capabilities of IOI and how it can be utilized to enable reuse in your applications.

I/O Example – Baseline Configuration

To illustrate IOI data storage and formatting capability, consider the data value *airspeed*.

Example 1, Step 1 - Data Formatted on Write

In Step 1, an application in partition P1 writes *airspeed*, as shown in Figure 3.

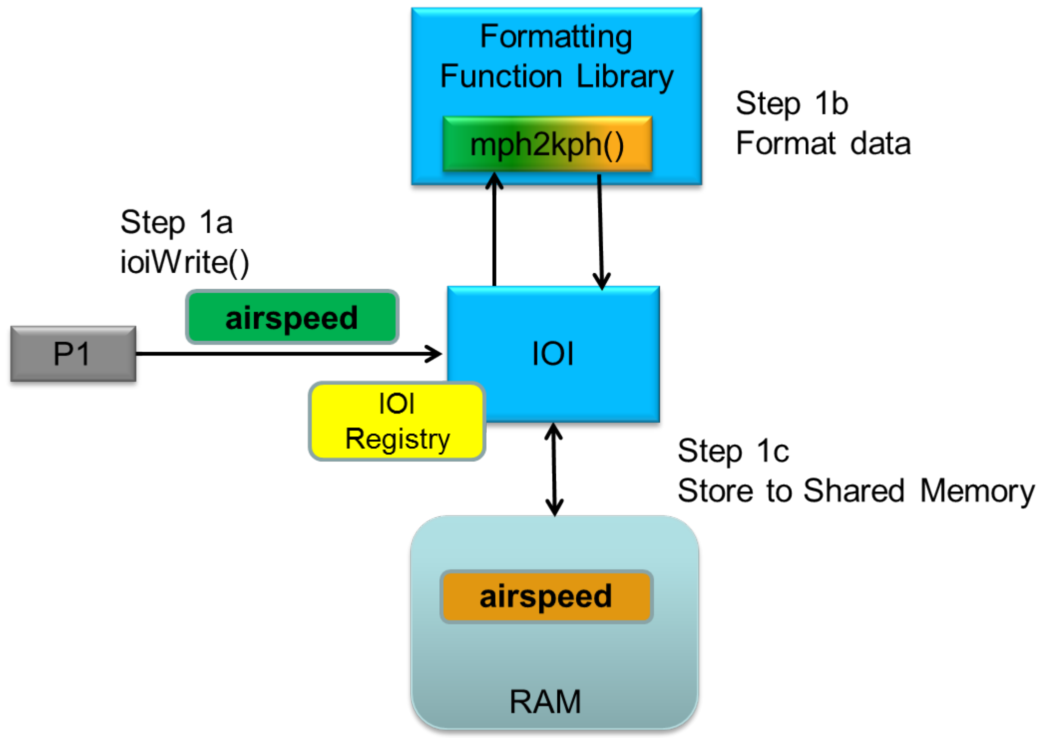

Figure 3 – Data formatted on write

In Step 1a, P1 calls *ioiWrite( )* to write *airspeed*. Green color-coding is used to show that *airspeed*, as produced by P1, is in a given engineering unit. For this example, assume that green represents *miles-per-hour*.

In Step 1b, based on instructions in the IOI registry, IOI invokes *mph2kph( )*, a user-defined formatting function that converts *miles-per-hour* to *kilometers-per-hour*, represented here by green transitioning to orange.

In Step 1c, IOI stores *airspeed* formatted in *kilometers-per-hour*, represented here by orange, in a block of RAM that it controls.

Recall that many I/O-related properties are configured via the XML-based registry files (i.e., IOI Registry). For example, those properties specify that *airspeed* is produced by partition P1 in *miles-per-hour* and that IOI should use the *mph2kph( )* formatting function before storing *airspeed* in RAM.

Example 1, Step 2 - Data Read as Stored in Memory

Next, in Step 2, partition P2 reads *airspeed*, as shown in Figure 4.

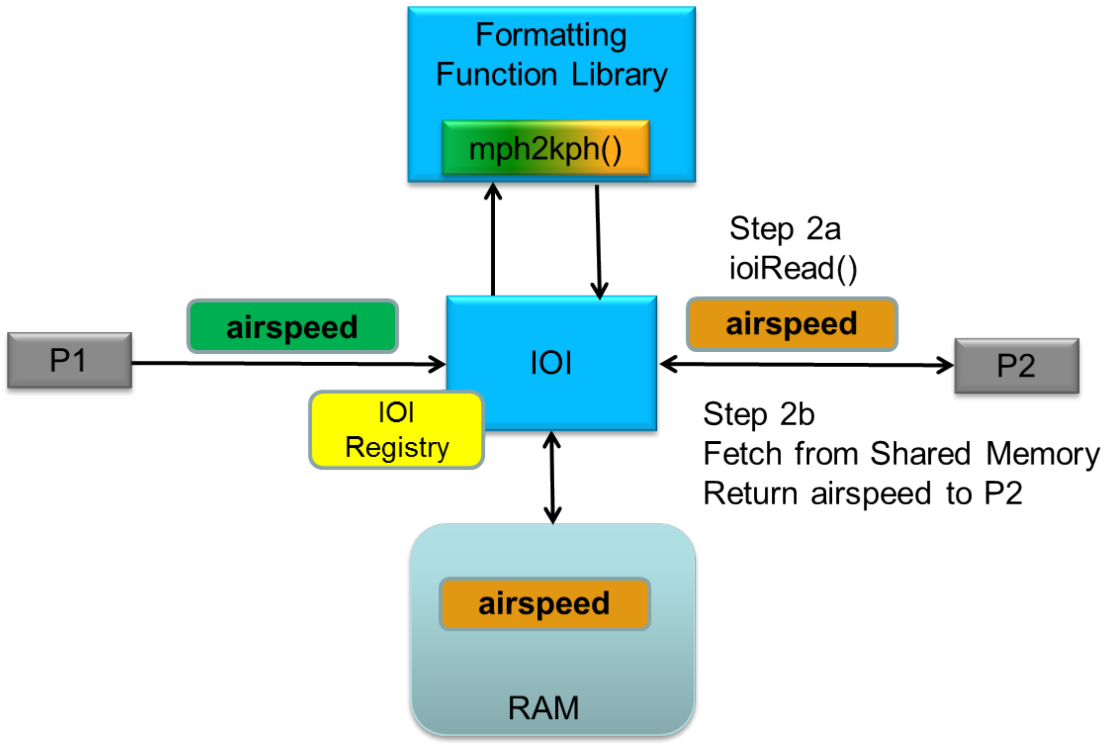

Figure 4 – Data Read as Stored in Memory

In Step 2a, P2 calls *ioiRead( )* to read *airspeed*. Here, assume that P2 wants *airspeed* in *kilometers-per-hour* (again, represented by orange as in Step 1).

In Step 2b, IOI fetches *airspeed* from the RAM that it controls. Since *airspeed* is already stored in *kilometers-per-hour*, no formatting function is required and the value is passed directly to P2.

In this case, the IOI Registry specifies that P2 expects to consume the *airspeed* value produced by partition P1[1], and wants the value in *kilometers-per-hour* (which is how *airspeed* is already stored in RAM).

Example 1, Step 3 – Data Formatted on Read

Finally, in Step 3, partition P3 reads *airspeed*, as shown in Figure 5.

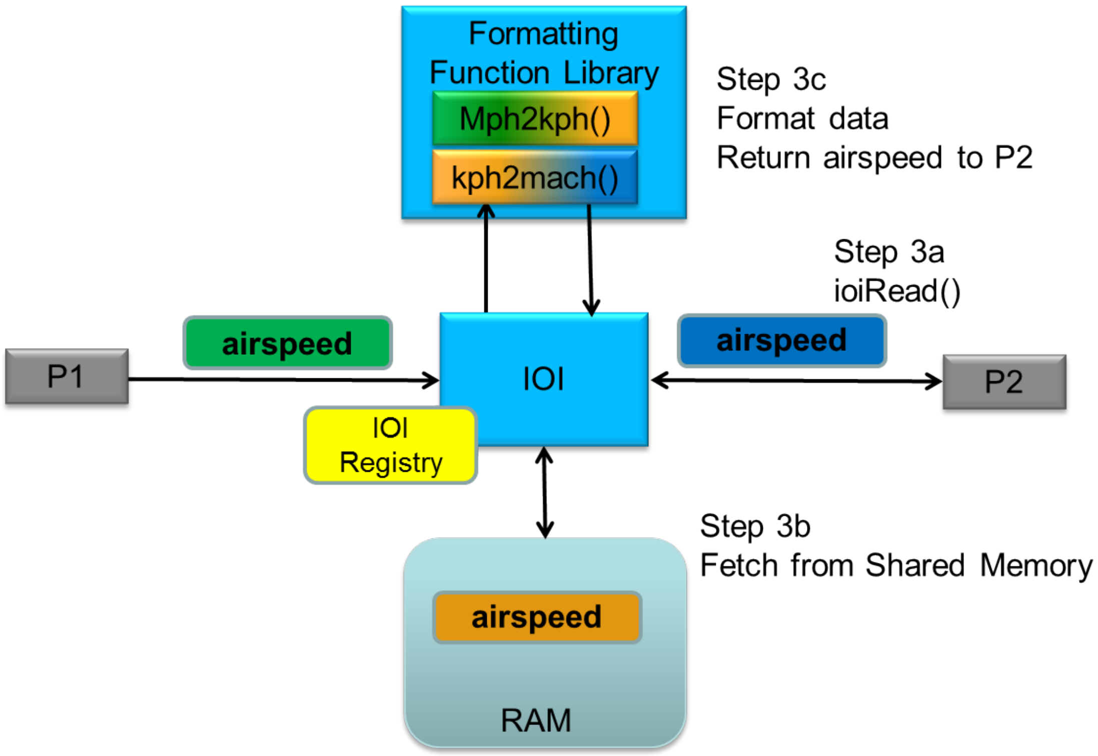

Figure 5 – Data Formatted on Read

In Step 3a, P3 calls *ioiRead( )* to read *airspeed*. Here, assume that P3 wants *airspeed* in *mach* (represented by blue).

[1] Other partitions could produce an *airspeed* value. Therefore each consumer of a value must specify the intended producer of that value. However, recall that this property is configured in the XML-based registry.

In Step 3b, IOI fetches *airspeed* from its RAM (stored in *kilometers-per-hour*), then invokes *kph2mach( )*, a user-defined formatting function that converts *kilometers-per-hour* to *mach*.

In Step 3c, the data is formatted and IOI returns *airspeed* to P3 in *mach*.

In this case, the IOI Registry specifies that P3 expects to consume the *airspeed* value produced by partition P1. It also specifies that P3 wants *airspeed* in *mach* and that IOI should use the *kph2mach( )* formatting function before returning *airspeed* to P3.

I/O Example 2 – Re-Configuring for Reuse

Recall the original configuration (from Figure 3), where P1 writes *airspeed* in *miles-per-hour* (green) and the IOI converts it to *kilometers-per-hour* (orange) before storing it in RAM. Then, when P2 reads *airspeed*, expecting it in *kilometers-per-hour*, the IOI simply returns the value stored in RAM.

Now, consider what happens to P2 from Example 1 if/when the target system within which it operates changes and its I/O changes along with it. This scenario is shown in Figure 6.

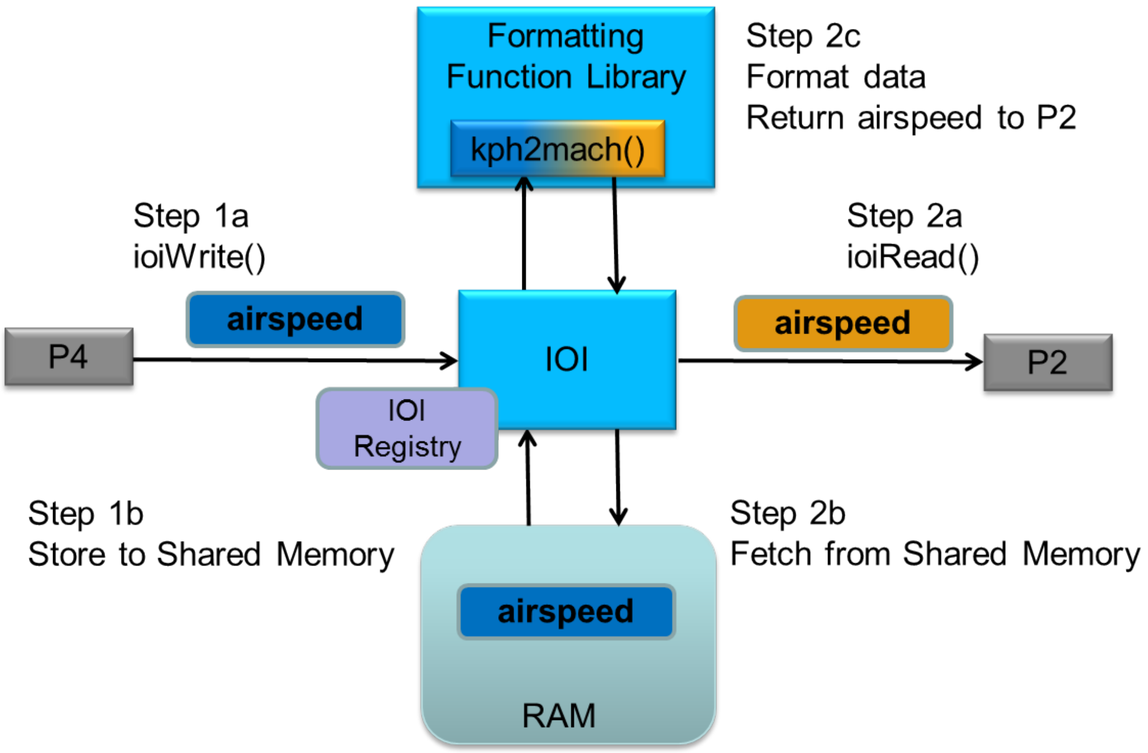

Figure 6 – Example 2, P2's I/O Reconfigured for Reuse

Here, notice that *airspeed* is produced by an application in P4, not P1, as in Example 1. Further, it's now produced in *mach* (blue) not *miles-per-hour*. Also, notice that the IOI saves *airspeed* to RAM in *mach*, with no formatting function applied at the time *airspeed* is written by P4.

Next, the application in P2 reads *airspeed*, and based on the IOI registry file, IOI knows that P2 wants *airspeed* in *kilometers-per-hour*. To accomplish this, the IOI invokes the *mach2kph( )* formatting function, which converts *mach* to *kilometers-per-hour*, and return *airspeed* as expected by P2.[2]

Now, the question is, can the P2 software remain unchanged in this new target configuration, while performing the same function it performed in the old configuration, even though its I/O is now completely different?[3]

Recall that the answer to this question has particular importance for avionics software. If P2 is modified to adapt it to the new I/O environment, then not only has the software been modified, requirements and design may change too. Further, significant rework likely will be required on the previously completed verification activities (i.e., tests, reviews and analysis). Such rework can be very costly in terms of budget and schedule. Therefore, the goal is to minimize change to P2 thereby minimizing rework, while ensuring that P2 will safely perform its intended function in the new environment.[4]

At a high level, Figure 6 illustrates how to achieve this goal. Specifically, one simply reconfigures the IOI Registry used in Example 1 as follows:

- The registry is modified to indicate that *airspeed* is now produced by P4 in *mach*, rather than P1 in *miles-per-hour* (as before).
- The registry is modified to indicate that P2 now consumes *airspeed* from P4, rather than P1 (as before).
- The registry is modified to indicate that IOI should invoke the *mach2kph( )*, formatting function that converts *mach* to *kilometers-per-hour* when P2 reads *airspeed*, rather than when *airspeed* is written (as before)

Note that the IOI Registry still indicates that P2 expects *airspeed* in *kilometers-per-hour* (as before), which is essential to isolating P2 from the changes in its I/O.

[2] Formatting functions can be associated with the writing of a value (e.g., airspeed in Example 1), with the reading of a value (e.g., airspeed in Example 2), or with both. Where formatting functions are used is a design-time decision.

[3] P2 may also consume hundreds of values, other than *airspeed*, and many may change.

[4] The same is true of the IOI library itself. Specifically, one does not want to change it either even though it's operating in a completely new I/O environment.

I/O Example 3 – Data Chaining and Formatting

Many values that are required to be passed to avionics equipment are a combination of sensor inputs that are used to calculate the required data. While the application can receive the individual values, format and calculate the value needed, this can limit the reusability of this application. In this example we will use IOI to read multiple values, correctly format them and then calculate the desired value, thus keeping the end application reusable if the format of the data changes in the future.

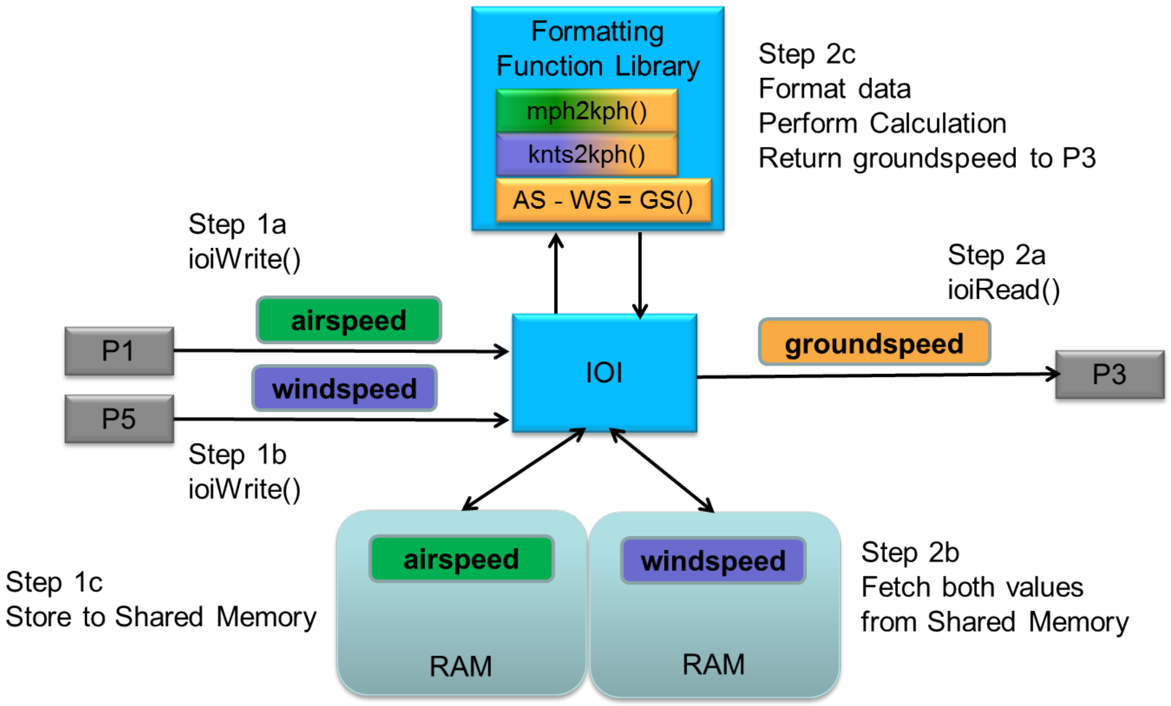

Figure 7 – Example 3, Data Chaining and Formatting

In Step 1a, P1 calls *ioiWrite( )* to write *airspeed*. Green color-coding is used to show that *airspeed*, as produced by P1, is in a given engineering unit. As before, assume that green represents *miles-per-hour*. Notice that the IOI saves *airspeed* to RAM in *miles-per-hour*, with no formatting function applied at the time *airspeed* is written by P1.

In Step 1b, P5 calls *ioiWrite( )* to write *windspeed*. Purple color-coding is used to show that *windspeed*, as produced by P5, is in *knots*. Notice that the IOI saves *windspeed* to RAM in *knots*, with no formatting function applied at the time *windspeed* is written by P5.

In Step 1c, based on instructions in the IOI registry, IOI saves the values of *airspeed* and *windspeed* in its defined shared memory regions.

In Step 2a, P3 calls *ioiRead( )* to read *groundspeed*. Orange color-coding is used to show that *groundspeed*, as required by P3, is in *kilometers-per-hour*.

In Step 2b, based on instructions in the IOI registry, IOI knows that *groundspeed* is a calculation of *airspeed* and *windspeed* and that P3 wants *groundspeed* in *kilometers-per-hour*. To accomplish this, the IOI reads *airspeed* and *windspeed* from memory.

In Step 2c, IOI invokes the *mph2kph( )* formatting function, on the *airspeed* value and invokes *knts2kph( )* on the *windspeed* value. Next, the IOI invokes the calculation *GS()* and returns *groundspeed* in *kilometers-per-hour* to P3.

It is easy to see how this could be expanded to handle many different scenarios that arise in avionics development.

I/O Example 4 – Abstract Data Referencing

For the previous examples we have been using single values produced by applications and stored in memory. In reality there are many situations where large amounts of data are sent between partitions via data structures. Although this makes it easy for applications to share data, it creates some significant problems when trying to create reusable applications.

- Both applications must understand the exact layout of the structure to get access to individual elements.
- Changes required if elements change position
- Changes required if elements are added
- Changes required if elements are removed

IOI provides a way to use the structures to store large amounts of data and still provide a way for applications to access all or individual elements without detail knowledge of the structure format.

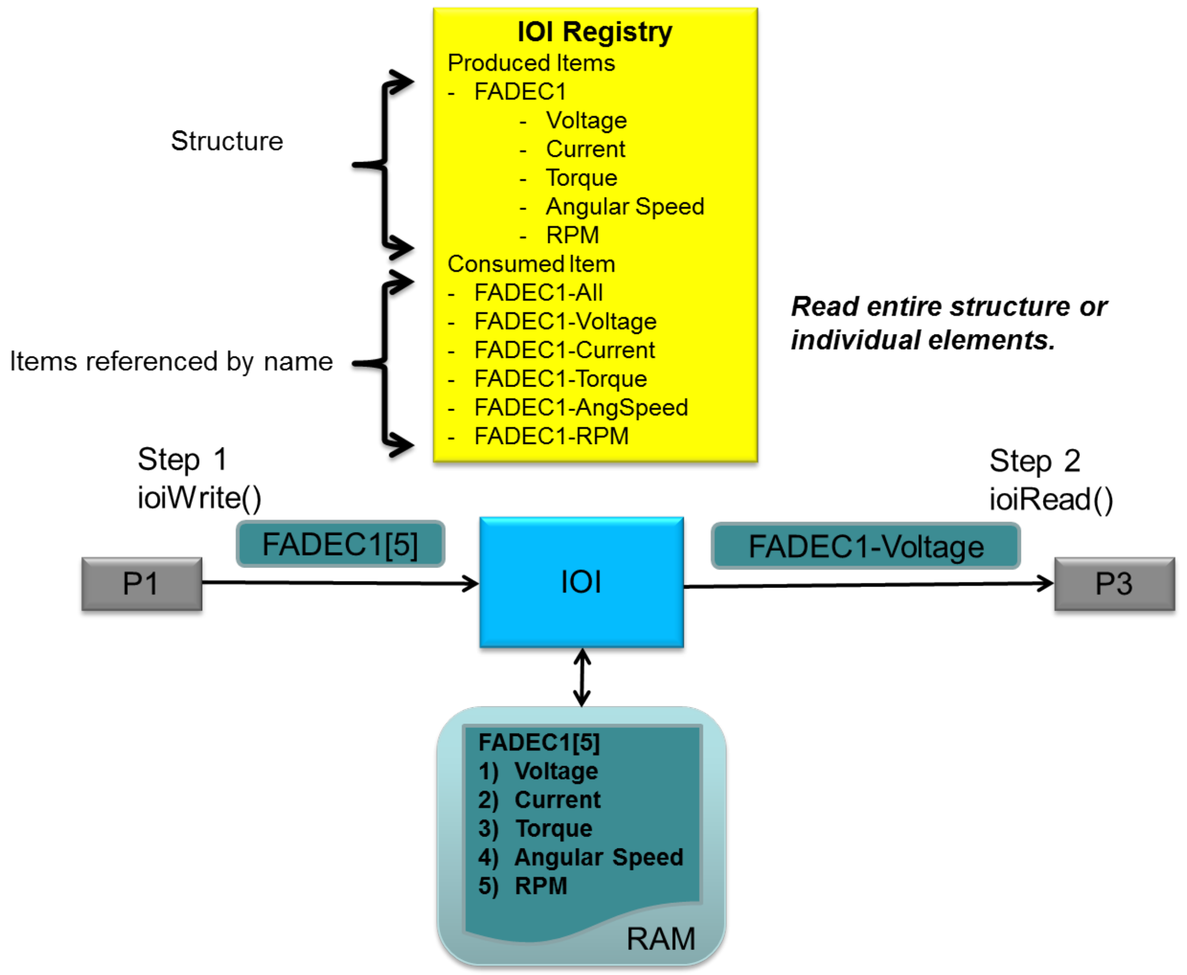

Figure 8 – Example 4, Abstract Data Referencing

In Step 1, P1 calls *ioiWrite( )* to write the entire data structure represented by FADEC1[5]. In this example there are 5 values where in real systems this could be 100's of values. Just like in previous examples each of the values of this structure could have different engineering units and could also be formatted on read or write as well.

In Step 2, P3 calls *ioiRead( )* to read *FADEC1-voltage*. In this way the individual elements of *FADEC1* are accessed by name and returned to P3.

In figure 8, we also show more detail about the contents of the IOI registry configuration file. The file shows that the *FADEC1* data is produced as a structure, but is can be consumed all at once (*FADEC1-All*) or by each individual element. By adding this abstraction detail in the IOI configuration file, P3 is not required to have knowledge of the structure that is providing the data.

I/O Example 5 – Re-Configuring for Verification Testing

Now, consider what happens to partition P2 from Example 1 (or Example 2) during verification testing. Actual target hardware is often very expensive and hard to get time on for software verification testing. Consequently, in order to keep schedules in check, one must conduct much of that testing on a reference platform test environment that is "sufficiently similar" to the target environment.[5]

Just like the reuse scenario in Example 2, if P2 must change when moving from the test environment to the target environment (e.g., changes due to differences in I/O), then significant retesting likely will be required on the target. Such rework can be very costly in terms of budget and schedule. Remember, the goal is to minimize change to P2 thereby minimizing retesting, while ensuring that P2 will safely perform its intended function in the target environment.

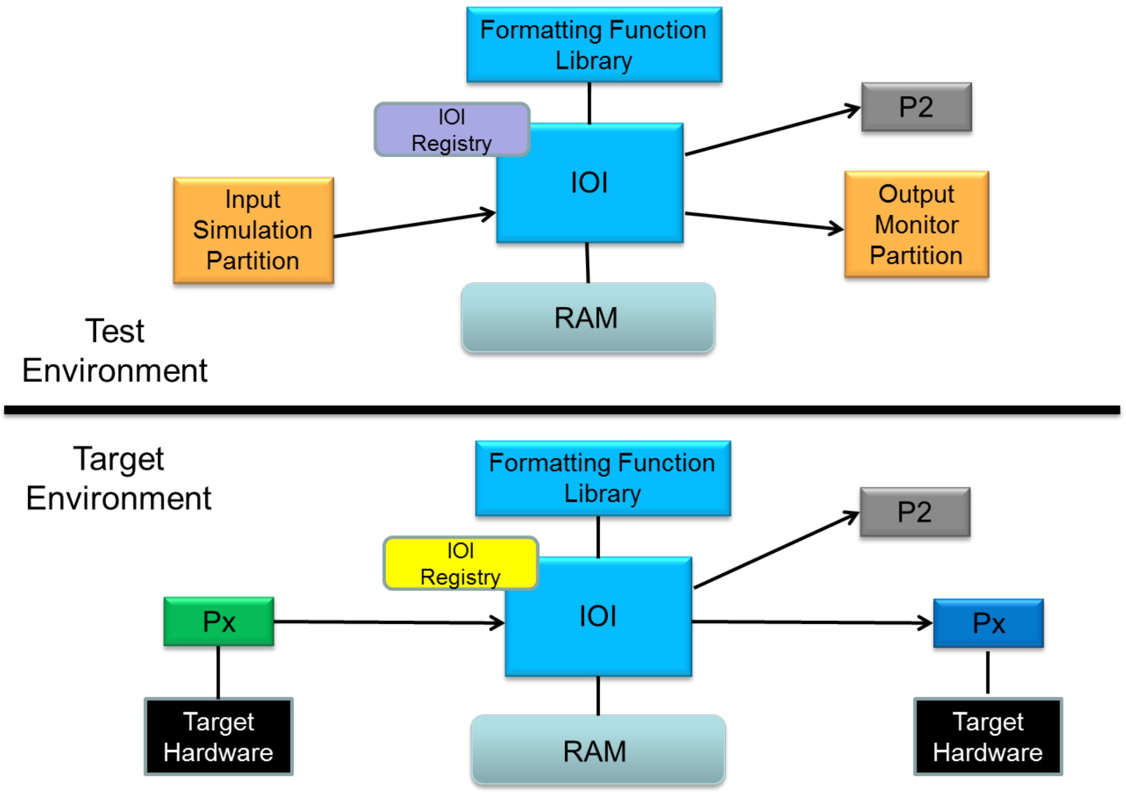

Figure 9 – Example 5, P2's I/O Reconfigured for Verification Testing

[5] Often times, even for high design assurance level software, as long as the CPU is the same in the test and target environments, they will be deemed "sufficiently similar" for preforming functional testing.

At a high level, figure 9 illustrates how to achieve this goal.[6]

Specifically, one creates an IOI Registry for testing in the test environment, then reconfigures the IOI Registry for the target environment:

- In the reference environment, the registry is configured such that:
    - P2's inputs and outputs are defined as expected in the target environment, from its perspective.
    - P2's inputs are produced by one or more input "simulators" to drive P2's test vectors. The simulators can be anything from a simple file of values up to a sophisticated set of device simulators.
    - P2's outputs are consumed by one or more output monitors so that engineers can monitor its behavior.
    - Formatting functions may be used, if desired.
- In the target environment, the registry is configured such that:
    - P2's inputs and outputs are defined as expected in the target environment, from its perspective.
    - P2's inputs are consumed from the actual sources of its data.
    - P2's outputs are consumed by the actual users of its data.
    - Appropriate formatting functions are employed.

Change Impact and Reconfiguration Costs

The main point illustrated in examples 1, 3 and 4 is the importance of keeping the data's characteristics (e.g., engineering units, data type and data rate) separate from the data in order to maximize reuse. These examples also showcase how IOI enables software engineers to accomplish this level of abstraction.

The main point illustrated in Examples 2 and 5 is that the application software in P2 remains unchanged in its new target environment, even though it's I/O is different.

Given that the P2 software is unchanged, it follows that P2's requirements, design, tests, test results, reviews and analyses haven't changed either. Consequently, a great deal of costly rework can be avoided and that can result in significant cost & schedule savings, especially in a certified, safety-critical world.[7]

Of course, IOI isn't a silver bullet. While P2 *hasn't* changed, the IOI Registry *has* been modified and DO-178C recognizes that configuration files like the IOI Registry must be

[6] In this example, for simplicity, each partition is color-coded to represent the characteristics of the data it produces and/or consumes.

[7] Of course, this statement isn't universally true. Hardware (and other) changes can be introduced that would force changes to P2 (e.g., a change in the production rate of *airspeed*). That said, IOI *can* be used to effectively manage many I/O changes, as discussed here.

treated much like software, with requirements and verification.[8] The exact nature of how configuration data is handled can vary program to program. It is up to system designers and IOI users to incorporate its use in their design and development as well as the verification processes and procedures.

Further, the test environment is typically sufficient for performing most (or all) functional testing. However, it will not accurately represent the timing of the target environment. Consequently, performance and/or time-sensitive testing must be performed in the target environment.

Finally, even though the application software may be unchanged, some retesting will be required in the new target environment to verify proper hardware/software and software/software integration. The amount and type of retesting required will depend on a number of factors, including the nature of the function being performed and the required level of design assurance.

Summary

Avionics system I/O is volatile and changes frequently from aircraft to aircraft. From the perspective of a given avionics software application, the function to be performed in one aircraft is often the same as in another aircraft and the I/O required is the same, but the I/O interfaces are different.

Managing this volatility via traditional means (i.e., software changes) can take a great deal of time and incur significant cost due to changes/rework.

The key benefit from using DDC-I's IOI is that I/O-related change impact is isolated in the IOI Registry configuration file, which typically results in simpler changes that can be re-verified more quickly and less expensively than if software changes are required.[9]

For Additional Information

[8] The same has been true under DO-178B since the early/mid 2000's.

[9] As avionics manufacturers begin to use multicore processors in their designs, capabilities like those provided by DDC-I's IOI will be invaluable, allowing designers to migrate applications between cores without software changes.