Ada Avionics Real-Time Software (AARTS) Operating Avionics architectures make use of recent advancements in microelectronics (VHSIC, MIMIC, packaging technologies, etc.) and software technology (the Ada language, fault detection/isolation techniques, etc.) to enhance the uninterrupted information and automation processes of its avionics systems. This type of architecture must address concepts such as information integration and coupling, high-speed bus transfer, a fault tolerant system architecture, resource sharing, and common modules. The real-time, multi-tasking and multiple processor configurations required for avionics systems have been examined in detail with respect to implementation in Ada to understand the problems associated with these configurations. A key element in this architecture is the software, specifically the operating system which runs as the core software element in the system. This paper will examine such a real-time operating system, the Ada Avionics Real-Time Software (AARTS) Operating System (AOS), developed by TRW for the Wright Laboratory (WL). Development concerns, integration issues, and testing results will be discussed.

DESCRIPTION

The AOS consists of a three part executive: the System Executive, the Kernel Executive, and finally, the Distributive Executive. Each is described in the following paragraphs.

System Executive

The System Executive (SE) resides in the Mission Data Processors. There are two copies, one of which is designated the primary (System Supervisor), and the other is designated the standby. The standby system operates in a 'hot backup' mode, available to take over whenever a fault occurs in the primary. The SE is responsible for monitoring the state of the system hardware and software. It is responsible for keeping a fault log for all system components and for controlling reallocation of the system resources. The SE handles two types of system resource reallocation. First, when any system component currently in use is determined to have failed, reallocation of the remaining components may be necessary in order to retain any functionality lost by the failure.

Kernel Executive

The Kernel Executive (KE) is responsible for controlling the application software assigned to a processor, providing control and communication between application tasks, controlling peripheral devices, and participating in processor level fault tolerance operations. Control of the application software is provided by a task sequence which handles the invocation and suspension of tasks within the processor. Inter-task control and communication services are provided to support the Ada (MIL-STD-1815A) tasking model which includes the concept of a rendezvous between tasks, allowing inter-task control and communication. The peripheral device control capability of the KE permits an application task to request an operation on a peripheral device at a logical level. Processor level fault tolerance operations include the receipt of processor error indicators, analysis of the conditions, and determination of the appropriate action. This appropriate action may be to pass the condition to an application task for further action by an exception handler, notify the system executive fail the processor, or any combination of these or other actions.

Distributed Executive

The Distributed Executive (DE) provides bus control interface capability, inter-processor data transfer operations, and general participation in the overall system operation. The DE communication interface provides for the exchange of information between processors. Data transfer requirements result from transactional requirements in support of application tasks and form system status monitoring and reconfiguration operations. System software and hardware configuration information is maintained by every processor in the system. The DE is responsible for the maintenance of the system state information, configuration requirements and system operation. A DE in one processor interfaces with the DE in other processors via the Mission Avionics Multiplex bus and with the Kernel Executive resident within its own processor.

AOS Operation

The AOS was developed and executed on the hardware configuration illustrated in Figure 1. A 'core' piece of the AOS exists that can be run in any V1750A CPU in the system, including the High Speed Data Bus (HSDB) Bus Interface Modules (BIMs), since these modules include a V1750A CPU. This 'core' piece of the AOS will consist of the Ada Run-Time System (RTS), service interfaces, the low-level PI-Bus I/O routines, and the Startup ROM (SUROM) software. User-specific application software perform the additional functions that are required. It is also important to note that the Ada tool set is a very important piece of this system since the Ada Run-Time is one the key building blocks of the AOS. The Tartan Tool set was selected for the AOS development.

The AOS operates in various modes and states. These modes and states provide the environment that satisfies its functional, performance, and interface requirements for the AOS. The modes

U.S. Government work not protected by U.S. Copyright.
describe the role the AOS assumes in a particular processor. The AOS has five modes of operation including: System Supervisor, Hot Backup, Cluster Controller, Specialist, and Off-line. The AOS can operate in only one mode at any specific time. The states describe the operations the AOS may execute on any processor. The AOS operates in four states including Startup/Restart, Configuration/Reconfiguration, Normal Operations, and Shutdown. The possible combinations of modes and states govern AOS behavior. For example, the AOS executing in System Supervisor mode and the Normal Operations state behaves differently than the AOS executing in Specialist mode and the Normal Operations state.

**System Communications**

Three approaches were considered on this program including: Singlecast where the sender of a message knows the exact address of the receiver(s) and broadcast from the sender to one or more receivers with status response(s) back to the sender; or broadcast from a sender to one or more receivers with no status response(s) back to the sender. The third alternative has the advantage of not caring where the receivers of the messages are located. Therefore, no extra time is consumed with status responses or maintenance of routing tables. This does require a message level control strategy in order to detect failures. Assuming this is provided, this alternative is highly conducive to a dynamic system. This is the one selected by the AARTS program because of the highly dynamic nature of the system. A message control strategy using “heartbeat” or status messages was adopted in order for the various CPUs to monitor each other and take action if necessary. Unfortunately, this alternative could not be completely tested due to PI-Bus problems with broadcast messages.

**Communication Control Strategy**

The Communication Control Strategy defined for AARTS centers around the use of logically addressed content labeled messages. In this scheme, each message is assigned to a unique, globally recognized label. The label or Message ID (MID) serves as the handle for all communications transactions. The DE assumes responsibility for ensuring that each globally addressed message is available to any node in the system.

**Message I/O**

Message I/O services provide the capability to send and receive information throughout the system. Message transaction requests are issued through the use of logical connections, either transmit or receive. Each connection represents a data flow path between a message data buffer and the local PI-Bus control data structures. Transmit connections are linked to a PI-Bus communication control block. Receive connections are linked to a PI-Bus receive label.

Both transmit and receive messages are DMA'd directly from/to CPU memory. The PI-Bus BIU can address any location in CPU memory, it can address only one buffer per label. This is to be expected. In situations within a single CPU, there are multiple receivers of the same message, the PI-Bus DMAs the data to the first receiver. When servicing the PI-Bus interrupt, the DE completes the transaction by copying the data from the first receiver to all subsequent receivers.

Message I/O uses the Ada Run Time System services to synchronize task execution with message services. In other words, task processing is suspended until the requesting transaction is completed (or times out). This provides the capability to develop a totally data driven system.

**Security**

One of the DE requirements is to provide a security mechanism to guard against unauthorized operations. The security function provided by the DE was designed following analysis of the NSA Orange Book.

---

**Figure 1 - Hardware Configuration**

**Software Mapping**

Four different alternatives were considered for mapping the AOS and applications to the VHSIC Avionics Modular Processor (VAMP) network processing resources including: a single Ada program distributed across the entire network; multiple Ada programs distributed one per CPU; multiple Ada programs within each CPU with no memory location restrictions; or multiple Ada programs within each CPU with each program limited to a 1750A address state. Data security and fault tolerant requirements (i.e. message security, isolation of applications, and the ability to reconfigure dynamically) resulted in the first three alternatives being rejected. Also, alternative one was not well supported by available tool sets. Alternative four was therefore selected. For the same reasons, it was decided to maintain the AOS as a separate program from the application software and provide an interface to the AOS via the 1750A Branch-to-Executive (BEX) instructions. This approach not only allowed full implementation of data security requirements, but also allowed the application software to be developed entirely in Ada and independently of the AOS.

**Scheduling Algorithm**

Scheduling of tasks is a critical issue in real-time systems. The AARTS program considered only two scheduling alternatives: periodic, hard deadline scheduling, and preemptive, interrupt driven scheduling. Because of the possibility of actions that require immediate attention and partially due to the decision to use a broadcast communication approach, the preemptive, interrupt driven approach was selected. This became important because changes in mission mode, or the detection of a failed CPU could be dealt with immediately instead of waiting for the next appropriate time-slice. Also factored into this decision was the requirement to support dynamic reconfiguration.
and companion documents. All Message and File I/O transactions are filtered through this function to determine if the requesting application has the proper security authority to access to access the data. All files and messages are tagged with a security code. This is done at system build time and a security file generated and stored in System Mass Memory. For Read/Receive operations the requesting application must have an equal or higher security level than the data object. Write/Transmit operations, the requesting application must have an equal security level. The security code of the requesting application must also have the same compartment(s) of the data object in order to access it. This provides a 'Need-to-Know' capability that can be implemented within a security classification.

Reconfiguration

There is a single configuration request message sent to all modules in a cluster. The message is tagged with the target module and the current configuration sequence. The configuration sequence is updated by the SE whenever a reconfiguration is initiated. If the Mass Memory (MM) detects a different configuration sequence and the message is tagged with the host module ID, the MM begins the reconfiguration process. The reconfiguration process is relatively straight forward and proceeds as follows: Loadable Program Units (LPUs) that are currently loaded, but not required are unloaded. This releases address states and data pages for use by new LPUs; LPUs that are required, but not loaded are loaded; LPUs that are loaded, but not running are started; LPUs that are loaded and already running are left alone.

If, for some reason, MM cannot load or start an LPU as directed, a non-compliance indicator is set in the report. This remains in effect until the SE provides additional direction with respect to the unimplemented LPUs. In these situations, there is no attempt by the MM to second guess the SE by reconfiguring itself. All reconfiguration activities must be initiated by the SE. When the reconfiguration process is completed, the MM generates an updated configuration report which is transmitted to the SE.

INTEGRATION/TEST

There were a number of challenges to overcome in the course of the integration of the AOS since all critical elements were essentially first generation implementations. This included the VAMP hardware/software, the Tartan Ada Tool Set, and each of the AOS components. To minimize impact of such problems, a three phased integration plan was implemented. The details of each phase are discussed in the following sections.

Module Integration

The purpose of module level integration was to integrate and test the AOS Kernel and establish a basic operational capability early in the program. The critical interfaces were the Ada Run Time System and PI-Bus I/O. Since the AARTS System control strategy depended on the proper operation of these two components, it was important to isolate any design problems early.

Integration activities were performed using two different target environments. A VAX hosted Mil-Std-1750A instruction simulator was used initially to verify the proper operation of the code generated from the Ada tool set. Integration of the software with the VAMP 1750A module was then performed. To complete this phase, the integration of the PI-Bus to the low level AOS utilities was performed. At the completion, an AOS Kernel communicating over the PI-Bus to the 1553B interface module was successfully demonstrated.

Distributed Integration

The purpose of the distributed integration phase was to verify the proper operation of the distributed components of the AOS. The critical elements in this phase included the communications interface to the Mission Avionics Bus (MAB) module, the File Services interface to the Block Transfer Bus (BTB) module, and the Module Management functions for program control. To work around a PI-Bus problem during communications testing, a broadcast capability was implemented in software by single-casting each message transmit request to all modules in the cluster. This ensured that all modules had access to all messages, but greatly impacted the message throughput.

The integration of the AOS File Services was essential to the system reconfiguration concept, since the AOS and LPU program images would be stored on the System Mass Memory. This phase involved bringing together three separate configuration items including the AOS, the Block Transfer Bus Module, and the Simulated System Mass Memory. Since the AOS-to-MAB module had already been integrated in the Mission Avionics Bus (MAB) module, the experience gained there allowed the AOS-to-BTB module to proceed smoothly. The BTB-to-SMM integration was more difficult, but completed with few problems.

System Integration

The system level test phase of the integration and test process verified the proper operation of the Distributed and System Executives and their interfaces. This included system level reconfiguration transition verification, the management of system resources, distribution of LPUs throughout the system, monitoring and reporting system status, controlling system startup/stop activities, and system fault recovery.

Most of the system level processing was initially developed and tested using the AARTS shell for VAX VMS systems. This step was initiated to take advantage of the powerful integrated development environment provided by the VAX Ada Compilation System. Using the Ada generic capability, it was not a difficult task to create virtual machines that hosted many different instantiations of the system and distributed executives. Since all system level interfaces were AOS messages, the AARTS shell provided a good intermediate step to the actual target processors. As a result of this step, many of the critical system level functions such as arbitration and state transition were functionally integrated and tested prior to VAMP integration. The actual target integration for these functions went smoothly.

DEMONSTRATIONS

The AARTS program provided three major demonstrations (with Demonstration #2 being split into 2A and 2B) to show the progress of the work being done for the AOS. Each demonstration was designed to show the increasing functionality of the AOS. A summary of the functions verified in each of the demonstrations is provided in Figure 1.

Demonstration #1

The primary purpose of this demonstration was to show that an Ada-based operating system could support real-time avionics operation. The functions included in this demonstration are shown in Figure 2.
A single program image was created using the AOS, a simple airplane model, and a representative avionics application program (navigation and guidance functions). Each application function was coded as a single Ada package using AOS message I/O for inter-program communication. The message interfaces were designed to interface with the existing cockpit controls and displays, thus providing a real-time visual display.

A single VAMP cluster was configured such that the demo program was loaded into one CPU module via the VAMP Console Interface Unit (VCIU) software into a VAMP CPU module. The MIL-STD-1553B module within the same VAMP cluster was then loaded with AOS and a Mil-Std-1553B bus controller. This configuration would demonstrate the communication thread between the PI-Bus and the Mil-Std-1553B.

Only one-way PI-Bus communication was possible due to a hardware problem which would intermittently cause a module's PI-Bus Interface Unit (BIU) to lock-up. Figure 3 shows the configuration used for Demonstration #1. Displays provided included a waypoint map and a running clock.

Demonstration #2

This demonstration was split into two parts due to delays in fixing the PI-Bus problem which affected Demonstration #1 and a delay in completion of the System Executive functions required to support Demonstration #2. This demonstration added the functionality shown in Figure 2.

For this demonstration, all of the application software as well as the AOS were built as separate Ada programs. Two VAMP clusters were used for this demonstration. One VAMP contained a CPU module, a MIL-STD-1553B module, and a High Speed Data Bus (HSDB) module connected to the Mission Avionics Bus (MAB). These modules were loaded statically via the VCIU with the AOS and interface application software (sensor, cockpit, and display generation), the AOS and the MIL-STD-1553B control software, and the AOS and the MAB HSDB control software, respectively. This VAMP served as the interface between the MIL-STD-1553B and the Integrated Test Bed (ITB) facility. The other VAMP, termed the mission cluster, consisted of a CPU module and two HSDB modules with one HSDB module connected to the MAB and the other connected to the Block Transfer Bus. The software used on this cluster included the AOS in all modules, the avionics application software (i.e., navigation and guidance) in the CPU module, and a passive boot loader which was dynamically loaded, as required, at start-up. All software on this VAMP was loaded dynamically from the Simulated System Mass Memory (SSMM) over the BTB. Figure 4 shows the configuration used for Demonstration #2. Operator I/O was provided on the laboratory control station via a Vertical Situation Display (VSD), a Horizontal Situation Display (HSD), the Mission Mode Panel (MMP), and the Integrated Multifunction Keyboard (IMFK). In addition, a VT220 terminal was used to monitor the module loads and provide system status information.

Demonstration #2B ran as expected. All displays updated smoothly, all operator inputs were handled properly, and the status display updated as expected.

Demonstration #3

The final demonstration was the same as Demonstration #2B with the addition of all remaining functions (as shown in Figure 2).

The hardware additions for this demonstration included a second HSDB module in the simulation cluster (for BTB access), a second CPU module in the mission cluster, and another full mission cluster identical to the first. The software loads were organized as they were for Demonstration #2B. A software work-around was provided to bypass the PI-Bus broadcast problem. Start-up control was identical with two exceptions. First, all modules in the simulation cluster were dynamically loaded. Second, all CPU modules completed loading of the
This demonstration went extremely well. Mission mode changes were entered by the operator via the appropriate, inter-cluster arbitration. Figure 1 shows the configuration used for Demonstration hardware resets to the mission clusters at various AARTS sizing and timing, including Lines Of Code performance. The following paragraphs contain data concerning AARTS sizing and timing, including Lines Of Code performance. As supplied from the vendor and additional code added by the AARTS program. The LOC numbers for Ada compilation units are the number of executable instructions and data storage assembly language units are the number of blank lines and comments. The LOC numbers for Ada compilation units are the number of executable instructions and data storage assembly language units are the number of blank lines and comments. The LOC numbers for Ada compilation units are the number of executable instructions and data storage assembly language units are the number of blank lines and comments.

| LOC | Cluster Mgmt package body, 708 LOC, produces 6564 words of code for a ratio of 9.27 words/LOC | - Cluster Mgmt package body, 708 LOC, produces 6564 words of code for a ratio of 9.27 words/LOC  |
| LOC | KE Support package body, 35 LOC, produces 277 words of code for a ratio of 7.91 words/LOC | - KE Support package body, 35 LOC, produces 277 words of code for a ratio of 7.91 words/LOC  |

The disparity in the ratios for the code packages above is directly attributable to the complexity of the package and its data structures. Generally, the more complex the function, the higher the ratio. Since the above is a representative cross-section of the AOS software, the average of these numbers (approximately 6.8 words/LOC) would be a valid ratio for comparison purposes.

### Sizing

There is always concern when dealing with embedded computer operating system software about size. Table 2 summarizes the goals and totals.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Goal Code</th>
<th>Goal Data</th>
<th>AOS/SSD Actual Code</th>
<th>AOS/SSD Actual Data</th>
<th>AOS/SSD Actual Code</th>
<th>AOS/SSD Actual Data</th>
<th>AOS/SSD Actual Code</th>
<th>AOS/SSD Actual Data</th>
<th>Total Code</th>
<th>Total Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOS</td>
<td>12.7</td>
<td>22.3</td>
<td>16.00</td>
<td>14.00</td>
<td>10.97</td>
<td>6.25</td>
<td>59.22</td>
<td>21.02</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: LOC, produces 6564 words of code for a ratio of 9.27 words/LOC.

As can be seen from the table, the data sizes correlate very well with the goals established for the AOS. The code sizes are higher, however, the AOS code as built contains full Ada run-time constraint checking, console I/O functions, and extensive built-in debugging software. In addition, the KE actuals only appear in the CPU modules which assume the roles of System Supervisor and Hot Backup.

### Timing

There are four timing considerations which are important in the operation of the AOS. These include the time to populate an empty VAMP, message transfer times, reconfiguration times, and system services times. All of the timing numbers discussed in this paragraph were obtained from the final version of the AOS which was used in Demonstration #3.

The time required to populate a VAMP (i.e., a BTB HSDB, a MAB HSDB, and two CPU modules) was approximately 11 seconds. This activity included the following processing:

- applying power to the VAMP via the Processor Control Panel (PCP)
- performance of the SURON code in each module (a parallel activity)
- the BTB HSDB module being loaded with the BTB Control Software from the SSM
- the MAB HSDB and CPU modules being serially loaded with the passive boot software from the SSM
- each passive boot software starting and becoming operational
- the MAB HSDB module being loaded with the MAB Control Software from the SSM in 4k word increments interleaved with the CPU
modules being loaded with the AOS from the SSMM in 4K word increments, and the MAB Control Software and AOS loads starting and becoming operational.

Message transfer times were obtained by loading test programs via the VAMP Console Interface Unit (VCIU) software, executing them, and watching the screen for text output indicating the test results. The times observed for message transfer are as follows:

- Between LPUs within a CPU: 1623 usec
- Between CPUS within a VAMP: 960 usec
- Between CPUs in different VAMPS: 2335 usec

It is important to note that each of these times include the actual execution of four separate transmit commands due to a problem encountered with the VAMP PI-Bus hardware. Once this problem is resolved, each of these times should be reduced by at least a factor of four.

Another important timing factor for real-time fault-tolerant operation is the time required to reconfigure the system due to a mission mode change or as a result of an error condition. Table 3 contains reconfiguration times for various system conditions.

An important note concerning the times shown in Table 3 is that configuration and health status reporting functions are intensive users of the message communication capability and, as such, are also subject to delays because of the VAMP PI-Bus hardware problem. Correction of this problem should decrease the latency of determining that a reconfiguration must be done, thereby reducing the overall reconfiguration times.

Finally, the timing information for the 1750A version of the AOS system services is provided in Table 4. The service timing information was obtained in a similar manner to the message transfer times. A test LPU was loaded via the VCIU software, executed, and the times for each service displayed as text on the VCIU screen.

**Table 3 - Reconfiguration Times**

<table>
<thead>
<tr>
<th>Function</th>
<th>1 VAMP 4 CPUS (AOS/SSM)</th>
<th>1 VAMP 2 CPUS (AOS/SSM)</th>
<th>1 VAMP 1 CPU (AOS/SSM)</th>
<th>1 VAMP 1 CPU (AOS/SSM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Mode Change</td>
<td>245 x 42</td>
<td>45 x 18</td>
<td>31 x 27</td>
<td>21 x 24</td>
</tr>
<tr>
<td>CPU Failure Recovery</td>
<td>N/A</td>
<td>N/A</td>
<td>21 x 23</td>
<td>21 x 23</td>
</tr>
<tr>
<td>VAMP Failure Recovery</td>
<td>N/A</td>
<td>N/A</td>
<td>21 x 23</td>
<td>21 x 23</td>
</tr>
<tr>
<td>Supervisor Failure Recovery</td>
<td>N/A</td>
<td>N/A</td>
<td>21 x 23</td>
<td>21 x 23</td>
</tr>
<tr>
<td>Fail Safe Boot Failure Recovery</td>
<td>N/A</td>
<td>N/A</td>
<td>21 x 23</td>
<td>21 x 23</td>
</tr>
</tbody>
</table>

**Table 4 - AOS System Services Timing**

**REFERENCES**


**CONCLUSIONS**

The AARTS Program met its program objectives. The success of demonstration #3 proved that Ada could indeed support fault-tolerant operation of an Operational Flight Program (OFP) on a distributed network of VHSC processors. In addition, the AOS performed extensive history logging (for debug purposes), console I/O, and ran with the benefit of full Ada run-time constraint checking. The modular design and the isolation of machine dependent functions resulted in code that was, and still is, highly reusable. The same demonstration software will be used for integrating and testing a Motorola 68020 CPU module in the VAMP cluster. The only required change will be to recompile the code for the appropriate target.