A FAULT-TOLERANT SYSTEM INCORPORATING AN ADA* EXECUTIVE AND 1750A PROCESSORS

David Butler

Sanders Associates
Nashua, New Hampshire

INTRODUCTION

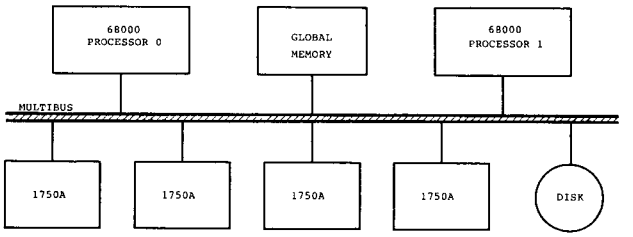
Current embedded military systems, and to a greater extent, systems of the future, must meet very stringent timing and reliability requirements. Uniprocessor systems with general purpose operating systems are unable to meet these requirements. In order to satisfy throughput requirements, systems consisting of multiple processors with specialized functions must be developed for use in embedded systems. In order for a multiprocessor system to be reliable, it must be able to maintain its required throughput even when processors fail. A reconfigurable system must be able to assign resources (including processors) so that the resources are used in the most effective way to accomplish the mission.

The DoD has mandated that the programming language Ada be used in future embedded systems. In addition, many Air Force systems are required to use MIL-STD-1750A processors.

The Sanders Executive testbed was created in order to perform research in the areas of multiprocessing and fault tolerance and to analyze Ada's feasibility in embedded systems. The testbed was required to incorporate MIL-STD-1750A processors.

HARDWARE OVERVIEW

The hardware layout of the system is shown in Figure 1. The system is built on an Intellimac Ada workstation consisting of two 68000 processors and global memory on a Multibus. Four Sanders 1750A processors have been added to the system, making a total of six processors. Each 68000 has 2MB of local memory and runs at .8 MIPS. Global memory is 4MB.



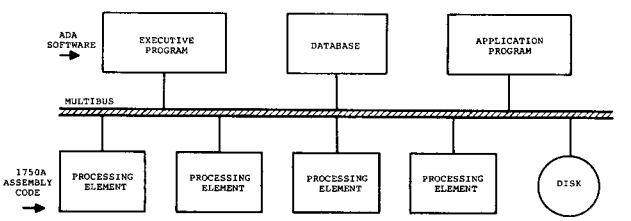
SYSTEM ARCHITECTURE
FIGURE 1

The 1750A processors have 64K program/data space. The 1750A processors, built by Sanders, run at 4 MIPS (non floating point). Each 1750A has a one-word mailbox associated with it which is located in Multibus I/O space. When its mailbox is written to, the 1750A is interrupted.

The 1750A's are housed in a cabinet with control panel switches so that individual machines can be turned on and off ("killed"). These switches allow the operator to kill processors easily in order to test system reconfiguration capability.

SOFTWARE OVERVIEW

The software functions in the system are shown in Figure 2.



SW DISTRIBUTION
FIGURE 2

Ada® is a registered trademark of the U.S. Government, Ada Joint Program Office.

The Ada programs in the 68000's run on the Telesoft ROS operating system. The compiler used was Telesoft version 1.3.

The Executive program is written in Ada. Its functions are to make an initial hardware configuration check, monitor the health of the Processing Elements and initiate reconfiguration when a Processing Element dies. A "heartbeat" mechanism has been implemented to monitor the health of the Processing Elements. Each Processing Element is assigned a "heartbeat word" in global memory which it must increment periodically. The Executive examines each heartbeat word periodically. If any heartbeat word has not been incremented since the last check, the Executive concludes that the corresponding Processing Element has died, and alerts the Application Program to the updated status of the Processing Elements.

Using Ada's data hiding techniques, the Executive was implemented so that it is completely generic. The Executive does not care how many Processing Elements are to be monitored, as the Executive deals with PROGRAM_ID'first ... PROGRAM_ID'last. In the future, two more 1750A's will be added to the system, but only the DATABASE package (which contains the changed definition of type PROGRAM_ID) will have to be changed, not the Executive program itself.

The Application Program, written in Ada, is responsible for reading data from disk and scheduling the Processing Elements for work. In addition, the Application Program maintains a status display which shows the status of each Processing Element.

Presently, the Processing Element software consists of bubble sort routines coded in 1750A assembly code.

The default state of a Processing Element is the idle state, in which it sends heartbeats to global memory when interrupted by the internal clock. A Processing Element is interrupted when a PROCESS_COMMAND is written to its mailbox by the Ada Application Program. The Processing Element then fetches the address of a packet of unsorted arrays from a table in global memory, sorts the arrays, and writes the results back to global memory. The Processing Element then returns to the idle mode until the next PROCESS_COMMAND is received.

The next phase of the program is to design and code electronic countermeasure (ECM) algorithms in Ada, translate them to 1750A assembler code (using an Ada to 1750A code generator), and download the code to the 1750A's, replacing the current sort routines.

IMPLEMENTATION OF GLOBAL DATABASE

In order to implement the system with a global data base, it was necessary for programs written in different languages (Ada and 1750A assembler) and running on different processors to be able to access the data stored in global memory.

Once it was determined that all programs could map to the same area in global memory, it was a matter of placing data structures into global memory in such a way that all programs could access them.

Each Ada program must "with" package DATABASE. This package contains all of the data structures which must be shared. Each data structure is declared and is then placed at an explicit address by using the Ada address specification. For example, once we've declared the object SYSTEM_CONFIGURATION_TABLE, it is placed at location 7A7000(hex) with the following statement:

```
for SYSTEM_CONFIGURATION_TABLE use at
16#7A7000#;
```

Now, when an Ada Program accesses an element of SYSTEM_CONFIGURATION_TABLE, the program is reading or writing a location in global memory.

Since the software running in the 1750A processors is written in assembly language, the addresses of the tables in global memory must be hard-coded into the 1750A programs.

INTER-PROCESS COMMUNICATION

A standard interface concept was developed for Ada programs. Using this standard interface, an Ada program can communicate with any other program in the system.

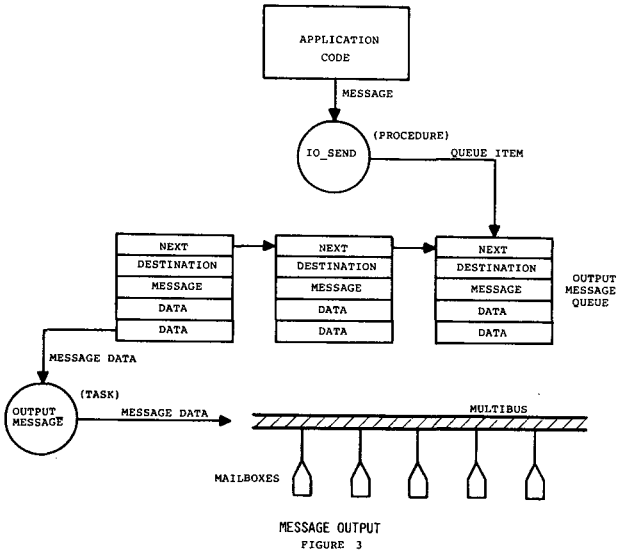
An Ada program communicates with other programs in the system by using the IO_SEND/IO_RECEIVE mechanism. IO_SEND and IO_RECEIVE are procedures which are defined in the IO_UTILITIES package. This package contains all of the system-specific knowledge needed to route messages between processes. An Ada program wishing to send a message to another program in the system calls IO_SEND, passing message type, associated data, and process name of the destination. In order for an Ada program to receive a message from another process, it must define a task which attempts a rendezvous with task IO_CHECK, which is also defined in package IO_UTILITIES. When IO_CHECK detects that a message has been queued for the calling program, IO_CHECK completes the rendezvous, returning the code of the incoming message to the calling program (the application code). The application code can then call IO_RECEIVE to get the data associated with the message. When the message has been processed, the application code then attempts another rendezvous with IO_CHECK, waiting for the next message. Note that the task which is attempting the rendezvous is suspended while the rendezvous is not completed. Therefore, this task is not using CPU time while it is waiting. In the meantime, other concurrent tasks in the Ada program are performing their assigned actions.

The advantage of this interface implementation is that if the interface mechanism changes, only package IO_UTILITIES has to be modified; the application code knows nothing about the

underlying communication mechanism, and so does not have to change if the interface mechanism changes. So, for example, if the inter-process communication mechanism was changed from polling to interrupt-driven, the application code would not have to be changed in any way.

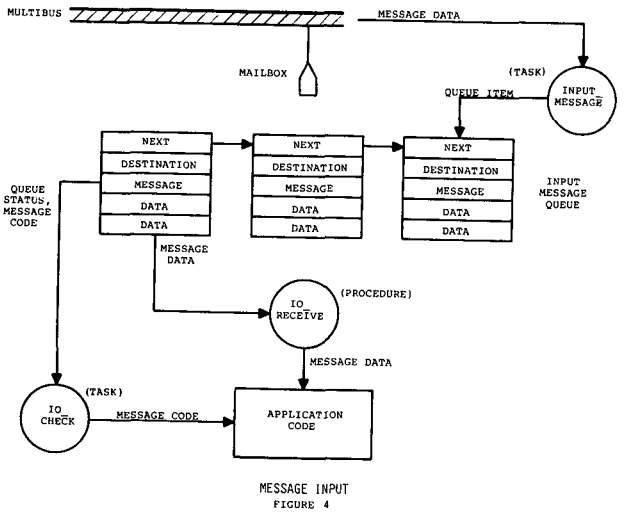
The inter-process communication mechanism was implemented using the concept of mailboxes and concurrent Ada tasks. Each 1750A program has a mailbox defined in Multibus I/O space. Each Ada program has a mailbox defined in global memory. Using mailboxes, any program in the system can communicate with any other program in the system.

Figure 3 describes the mechanism for sending a message. The Ada program calls procedure IO_SEND with the message type and associated data. IO_SEND creates a queue item containing the information and appends it to the output queue for that Ada program. Task OUTPUT_MESSAGE polls the output queue; when the queue is not empty, the next item in the queue is dequeued and the message and data are sent to the specified destination's mailbox.



MESSAGE OUTPUT
FIGURE 3

Figure 4 describes the mechanism for receiving a message. Task INPUT_MESSAGE polls the mailbox of the Ada program and appends an item to the input message queue when a message is received in the mailbox. Each Ada program defines a task which is responsible for checking for incoming messages. This check is accomplished by attempting a rendezvous with task IO_CHECK. IO_CHECK has a guarded select statement so that it will complete the rendezvous only if there is an item in the input message queue. As long as the input message queue is empty, the task which is attempting to rendezvous with IO_CHECK will be suspended. When an item is appended to the input message queue by task INPUT_MESSAGE, IO_CHECK will complete the rendezvous and the calling task will be provided with the message code of the incoming message. If the message has data associated with it, procedure IO_RECEIVE is called to get the data. The message is then processed by the application code.



MESSAGE INPUT
FIGURE 4

FAULT DETECTION/RECONFIGURATION

When the Processing Elements are started up, they immediately begin sending heartbeats to global memory. The Executive is started and uses the heartbeats to determine which Processing Elements are initially up and running.

The operator is told how many Processing Elements are available and is requested to specify which are to be online and which are to be spares. By specifying how many Processing Elements are to be online, the operator has told the Executive the required system throughput; the Executive now is responsible for ensuring that that number of Processing Elements is online at all times.

If an online Processing Element fails, the Executive is responsible for detecting it and bringing up a spare to take over.

The Executive maintains the System Configuration Table in global memory, which contains the status of each Processing Element. There are three statuses: online, spare, and dead.

Once the initial configuration is determined, the Application Program is commanded by the Executive to start. After reading raw data from the disk, the Application Program schedules each online Processing Element for work (sorting

arrays) by using IO_SEND to send a PROCESS_COMMAND to each. When the PROCESS_COMMAND is received by the Processing Element, it accesses a table in global memory which points to a packet of raw data. A packet consists of a number of unsorted arrays. The Processing Element reads each array, sorts it, writes it back to a designated area in global memory, and increments an array count in global memory. When the end of the packet is reached, the Processing Element sends a PROCESS_COMPLETE message back to the Application Program, which zeroes the array count for that Processing Element and then schedules the Processing Element for another packet.

The Executive and the Application Program run in separate 68000's. So while the Application Program and the Processing Elements are doing their application-specific work, the Executive can concurrently monitor the health of the Processing Elements without slowing down system throughput.

The Executive is responsible for checking the Heartbeat Counters in Global Memory in order to monitor the health of each Processing Element. The Executive health monitor consists of three "concurrent" Ada tasks: one task to monitor Processing Elements which are online, one task for spares, and one task for "dead" Processing Elements. (A Processing Element whose state is "dead" is checked on a regular basis to see if it has begun sending heartbeats again, in which case its status becomes "spare." This situation occurs if the operator switches a 1750A from "off" to "on" from the control panel.)

By using Ada tasks and the Ada DELAY statement, we can set the frequency of heartbeat checks in the three tasks so that the health of Processing Elements in the more critical states is checked most often. Another feature of the DELAY statement is that between checks, a task is "asleep" and thus is not using any CPU time.

If the Executive detects that an online Processing Element has stopped sending heartbeats, it tries to find an available spare to take over processing. If a spare is found, the Executive uses IO_SEND to send a RECONFIGURE message to the Application Program. This message includes the names of the Processing Element which died and the one that is to take over. If a spare is not available, the Executive puts an item on a queue, and when a spare comes available (that is, a Processing Element changes from dead to spare) a RECONFIGURE message is sent to the Application Program.

The Application Program uses the array count in global memory to determine the number of arrays which were completed by the dead Processing Element. The Processing Element which has just been brought online is given the address of the next array in the packet and completes the packet. Using this simple checkpoint scheme, as little work as possible is re-done as a result of reconfiguration.

LESSONS LEARNED

1. "Data-hiding" was implemented by using packages DATABASE and IO_UTILITIES to insulate the Ada application code (Executive and Application Program) from changes to the system configuration and changes in the method of passing messages. When the number of Processing Elements is increased, the enumeration type PROGRAM_ID in package DATABASE will be modified to reflect the added Processing Elements, but the application code will not have to be changed. The fact that the Executive is monitoring six Processing Elements instead of four will require no change to the Executive.

We found that when the Application Program uses IO_SEND to send PROCESS_COMMAND to a Processing Element, the queue management and tasking mechanism of IO_UTILITIES incurs undesirable overhead. IO_UTILITIES was changed so that when the message to be sent is PROCESS_COMMAND, the message is written immediately to the Processing Element's mailbox, avoiding the usual queueing/tasking mechanism. The Application Program's call to IO_SEND to send PROCESS_COMMAND was not changed; but unknown to the Application Program, the message-passing mechanism was changed in package IO_UTILITIES.

Therefore, using utility packages to hide implementation details from the application code means that changes can be made to enhance the system and perform tradeoff studies while not requiring changes to the application code.

2. Ada tasks can be used to great advantage when a number of separate processes are required within a program. The use of tasks for health monitoring and message sending has been described. There are also independent tasks which update the status display and check for system shutdown. In the future, when Ada programs are executing in the 1750As, tasks will be used to send heartbeats as well as monitor heartbeats. Task priorities and the DELAY statement will be used to fine-tune the interaction of the tasks so that fault detection time is minimized.

3. Along with the advantages of tasking come some problems. Once a software system begins using tasks, the behavior of the system is dependent on the implementation of the task scheduler. For instance, in our system, tasks run until blocked (that is, until a rendezvous or I/O statement occurs). We have no control over how the runtime system schedules tasks. We have no way to try a round-robin scheme, for instance. If a task which contains an infinite loop (such as a polling loop) begins executing and does not rendezvous or perform I/O, no other task will be able to start running, due to the non-preemptive task scheduler. This problem was dealt with by inserting a rendezvous with a dummy task in the loop, so that the task is suspended each time through the loop.

In order to understand and predict the

behavior of a system which incorporates tasks, it is imperative that the task scheduling mechanism be understood by the user. That is, it should be well documented by the provider of the compiler/runtime system (RTS).

The task scheduler portion of the RTS should also be easily tailorable so that it can be fine-tuned for a given application. The user should be able to define a task scheduling strategy and fine-tune it for his application. The RTS source as well as a user's guide should be provided.

4. Ada is not portable. But this problem can be minimized in some cases by using packages to insulate implementation-dependent information. An example is the DATABASE package. The current compiler has address specifications implemented so that data structures can be placed at explicit addresses easily. The current compiler will be replaced by a new compiler which does not have address specifications implemented. This means that package DATABASE will not compile on the new compiler. However, since DATABASE is the only package which contains address specifications, it is the only package which will have to have address specifications removed. (Address specifications will be replaced by access variables so that data structures will still be placed in global memory at explicit locations; again, the application code which references the data structures will not have to be changed).

Another example of lack of portability is task priorities. Our current compiler does not have task priorities, but if task priorities were available, pragma PRIORITY would be sprinkled throughout the Ada programs. A compiler change could mean changing from a priority range of 0 .. 15 to a range of 0 .. 7. In this case, the implementation dependency would be a problem because all of the pragma PRIORITY statements would have to be found and checked, and if the system had become dependent on the larger range of priorities, it may not run as well with the smaller range.

Again, making the range of priorities tailorable by the user would make the compiler/RTS more flexible and practical.

5. System throughput in the testbed is limited by an obvious bottleneck: there are up to four high-speed Processing Elements interfacing to one Ada Application Program through the single Application Program mailbox. When the jobs to be performed by the Processing Elements are small and more than one Processing Element is online, the Processing Elements spend a lot of time waiting for the Application Program mailbox to be free in order to send the PROCESS_COMPLETE message and receive more work. (Bus traffic is negligible.)

A proposed solution to this problem is to define four tasks in the Application Program, one for each Processing Element, and each having a mailbox. If this approach were taken and all four tasks resided in the single 68000, there would still be a bottleneck, only it would be the 68000 CPU instead of the single mailbox. The reason for this is that the four tasks would not be running in parallel: they all would be time-slicing on the same CPU. A real system throughput improvement would be realized only if the tasks were distributed so that two are in each 68000. We hope to implement this approach in the near future and measure its performance.

The point is that Ada tasking does offer a means to attack a problem involving multiple processes, but it can only be efficient (i.e. appropriate for realtime systems) if the capability exists to distribute the tasks of an Ada program across CPU's and thereby truly achieve parallel processing.

FUTURE WORK

1. We are awaiting upgrades to the Ada workstation which will replace the Telesoft subset compiler and ROS operating system with a Verdix compiler and UNIX system V. The full Ada compiler will allow us to improve the software with elements such as task delays, task priorities, and task types. (Currently, task delays are simulated.)

2. The current sorting application will be replaced by ECM applications. We will procure a VAX-hosted Ada compiler with a 1750A back end and use it to translate ECM algorithms to 1750A code which will replace the current assembly language sort routines. The Ada Application Program will be replaced by a new program to drive the new Processing Elements. The Executive program will require a change because it is likely that the new ECM applications will incorporate "hot backups" as well as autonomous spares.

3. Two more 1750A processors will be added to the system. This will enable us to implement multiple processes in the Processing Elements with hot backups and autonomous spares. We will also port the Executive and Application Programs to 1750As.

4. The Ada Application Program which schedules the Processing Elements will be enhanced to consider application-specific scheduling requirements including degraded modes and prioritized scheduling based on missions.

SUMMARY

The Executive Testbed has been built to model the embedded military system of the present and future. The testbed has shown that a multiprocessing, fault-tolerant system can be built using Ada to implement the system executive and job scheduler.

Features of Ada which were used to implement the system include address specification, data hiding techniques, and multitasking. Positive and negative aspects of the Ada tasking capability have been found.