Spaceborne Application Multiprocessor Operating System

A Paper Presented To The

5th Annual AIAA/Utah State University Conference
On Small Satellites
August 26-29, 1991

Gary S Grisbeck, Control Data Corporation, EMS Division
Minneapolis, Minnesota

Wesley D. Webber, Control Data Corporation, EMS Division
Minneapolis, Minnesota

Spaceborne Application Multiprocessor Operating System

Gary S Grisbeck
Control Data Corporation, Minneapolis, Minnesota 55425

and

Wesley D. Webber
Control Data Corporation, Minneapolis, Minnesota 55425

Introduction

Control Data Corporation, as part of a joint development program with the Boeing Electronics Company, built a nine node, three active module multiprocessor called SPA-1 or Spaceborne Processor Array-1. (SPA was the processing system concept proposed as the heart of a multi-mission autonomous surveillance satellite in a paper [1] presented at the 4th Annual AIAA/Utah State Small Satellite Conference in 1990).

Since then, an operating system called the Operational Kernel, or OK, has been completed, tested, and functionally validated in a demonstration using the SPA-1. This demo featured **fully autonomous** on-board control of data movement, fault detection, fault isolation, hardware reconfiguration, application re-start, and load balancing/redistribution. The demo application consisted of ephemeris calculations being performed for two satellites, in each of three independent processors at different nodes on the SPA-1; this data (and SPA-1 health status) was sent via a serial I/O (input/output) channel to a host machine for display. With the demo software running in the three active processing nodes, observers were invited to cause random nodal interconnect or processing hardware elements to fail by selection of switches on a fault injection panel. SPA-1, under the aegis of the OK, detected that a failure had occurred, isolated it, reconfigured around it, redistributed the processing load (to the two or one remaining active processors) and continued with the application processing, all without operator intervention of any sort.

The OK is written in Ada. Support of the execution of Ada programs is provided for by the Ada Run Time System (RTS), provided by the Ada compilation system. The RTS provides very basic services such as tasking and memory management, and exception handling. The OK consists of Ada packages that are run on top of the RTS. This collection of packages include lower level services that involve message buffering, interrupt handling, and individual configuration commands. Operational high level services include: a block I/O facility that uses protocols to ensure the integrity of data transfers between modules at different nodes on the SPA network; a configuration facility that provides a high level set of operations to configure the network; and a health check facility to support application controlled detection and isolation of failed SPA elements.

The SPA hardware, with processing elements operating essentially asynchronously at each node on the network, supports many concurrent activities. The OK handles this with Ada tasks. The number of tasks is application dependent. Approximately sixty tasks were employed in the SPA-1 demo.

SPA Overview

The SPA is a modular collection of processing and external I/O resources. The modules are interconnected by the MIN (modular interconnect network), which is the heart of the SPA system. The MIN has high bandwidth, is reconfigurable under software control, and is fault tolerant. The SPA allows a heterogeneous mix of module types, achieving greater optimization than could be obtained with the use of universal modules. Every module contains processing resources, instruction and data memory, and an interface to the MIN.

The MIN is the heart of the SPA system; it distributes both power and data. The MIN uses software configurable data paths to form logical data path structures on the interconnect that is a two dimensional physical mesh. The modules form a plane with each processor having connections only to its four nearest neighbors, thus forming a Manhattan geometry (orthogonal interconnect). The physical implementation of this scheme allows the plane of processors to be topologically equivalent to a toroid. This minimizes the longest data path, which maximizes overall system operating speed and overhead.

The MIN is a fully distributed system. The element of the MIN residing in each node is the configurable network unit (CNU). The CNUs in each node are identical. Each CNU has two data path (edge) connections to each of its four nearest neighbors, for a

total of eight connections. Each of these eight connections is called an edge connector. Each edge connector can be programmed to be an input or an output, but not both simultaneously. Within the CNU there are four internal data paths between the edge connectors, and a system of multiplexers. The multiplexers allow any edge connector to provide an input to any of the internal data paths. Also, any edge connector that is programmed to be an output can take its data from any (or all) of the four internal paths. The CNU connectivity is illustrated in Figure 1.

Two of the four internal data paths have address recognition logic. These are called network interface units (NIUs) and will extract any word of data that is addressed to them, sending it to the processor associated with that node on the MIN. The NIUs (or ports) also can place data in any unused data slot in the data path. When a port removes a data word, it can also place (output) a new data word into the same slot for transmission down the MIN.

The other two internal data paths merely pass data through the CNU and to the next node in the MIN. These internal "passthrough" data paths are are called vias. A port can act as a via by putting the same data word back on the data path and sending it downstream to the next mode.

Present MIN hardware has information transferred at a rate of up to 8 million data words per second. The 52 bit data words are comprised of 32 bits of data, an 8 bit destination code, and other control information. Each data word has its own destination address; this allows data from different sources going to different destinations to be interleaved on any given physical data path on a word by word basis. Thus a data path does not have to be reserved for exclusive use by any given module, which greatly reduces latency. Furthermore, each data word carrying its own destination code means that no channel request protocol is required to send data. Data is queued for transfer and goes onto the data path whenever room is available.

Generic fault avoidance, fault detection, and fault tolerance techniques implemented at the module and SPA system level are discussed in great detail in the paper referred to at the beginning of this article.

SPA-1 Hardware

SPA-1, a three by three, nine node SPA system, was fabricated, tested, and delivered as part of the joint development program between Control Data Corporation and Boeing Electronics. The SPA-1 interconnect is graphically represented in Figure 2. A physical implementation of the whole SPA-1 system is shown in Figure 3.

The SPA-1 was connected to a portable test set (PTS) with a MicroVAX host that was used to develop and debug software. During the demo, all of the application and fault isolation, reconfiguration, and load shifting were accomplished by software running at nodes on the SPA-1; the PTS was used for status, command, and graphics display purposes only.

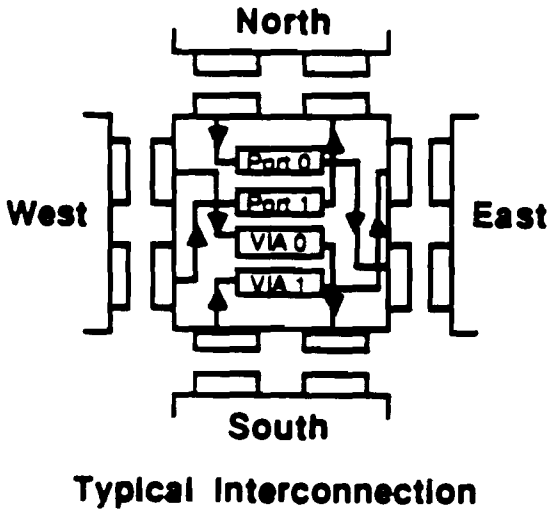
Only three of the nine nodes in SPA-1 had processors, and these were in locations 1, 3, and 8 (refer to Figure 2). Two of these processors, at locations 1 and 3, were modules with serial I/O hardware (called PIMs) and were attached to the PTS; and the other was a basic processor module or PM. The processors were Control Data's 444R[2], a (currently space qualified) MIL-STD-1750A ISA processor. The other six nodes on the MIN were "stubs" that electronically terminated the MIN elements to allow the network to function with no processor attached to the CNUs at any of these nodes.

SPA-1 Operating System Overview: The Operational Kernel (OK)

The OK, or Operational Kernel, is written in Ada. Support of the execution of Ada programs is provided for by the Ada Run Time System (RTS). The RTS provides very basic services such as tasking and memory management, and exception handling. The RTS is included in every active processor module in the SPA. The services of the RTS are not directly invoked by calls to the RTS, but are accessed through the use of standard Ada language statements. For this reason, a detailed treatment of the RTS is not included in this paper.

The OK consists of Ada packages that are run on top of the RTS, and which provide services not available from the RTS. This collection of packages include lower level services that involve message buffering, interrupt handling, and individual configuration commands. It also includes higher level services such as:

- MINIO - a block I/O facility for the MIN using protocols to ensure the integrity of data transfers between modules



Configurable Network Unit

- **1 Per Module**
- **Provides 4 Independent Data Paths Between 8 Connectors**
- **2 Paths Interface With Module**
- **Electrically Reconfigurable**

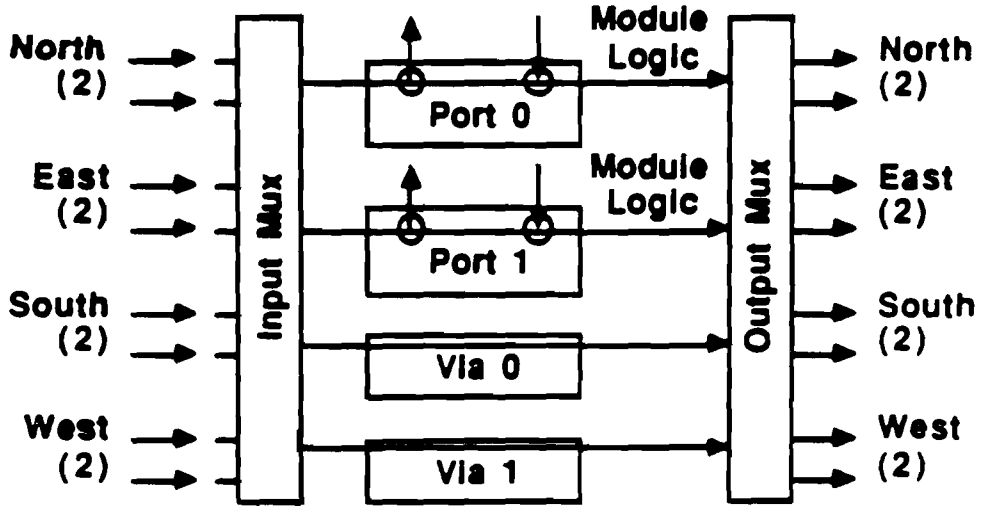
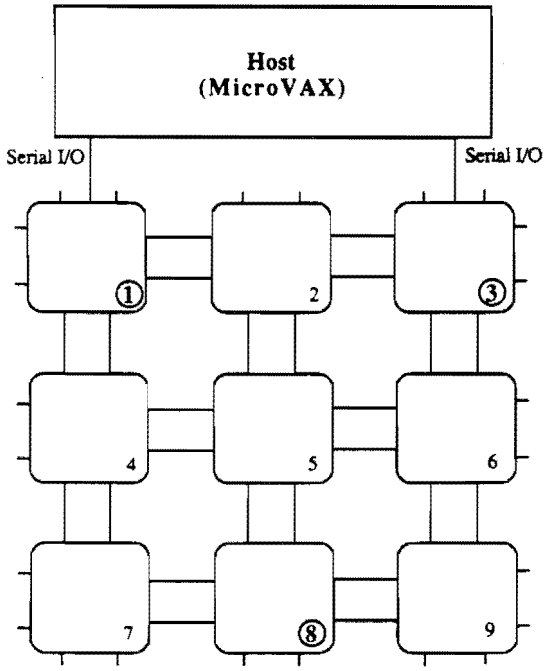


Figure 1. CNU Connectivity



Nodes 1,3, and 8 have 1750A Processor Modules
Nodes 1 and 3 have Serial IO Connected to the Host MicroVAX

Figure 2. SPA-1 Interconnect

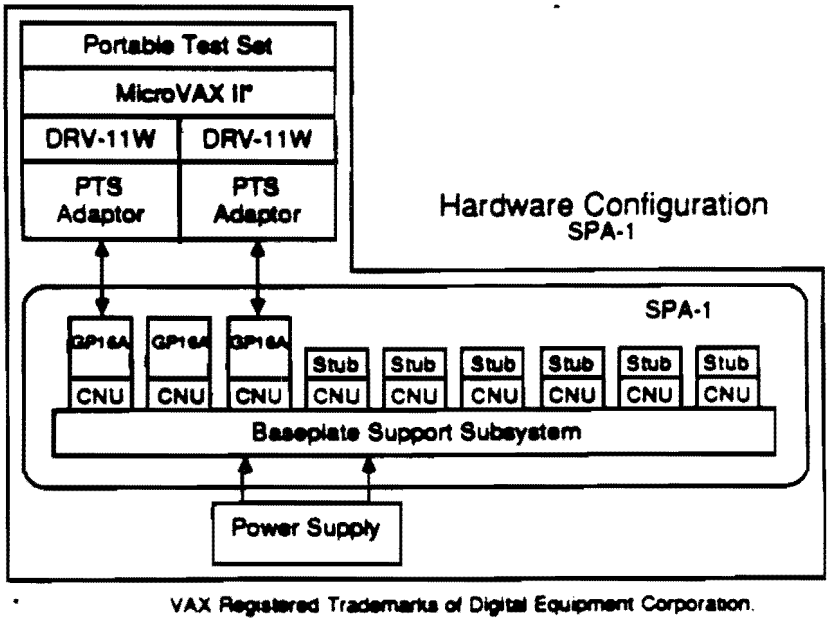


Figure 3. SPA-1 Configuration

- HEARTBEAT - a healthcheck facility to support application controlled fault detection and isolation on the MIN
- CONFIGURATION - a configuration facility that provides a high level set of operations to configure and re-configure the MIN

All, some, or none of these higher level OK services can reside in each processor.

OK - Theory of Operation

The OK supports applications in both a development environment and an operational environment (in the case of SPA-1, the demo). These environments differ basically only in their reliance on a host computer. The operational environment communicates with a host (a MicroVAX, in the case of SPA-1) over a serial I/O (in the case of SPA-1), or over a downlink in the case of a spaceborne application.

All communication in SPA-1 consists of software in one module passing messages and data to software in a second module. The code involved need not be concerned with the physical aspects of communications; the interface between the communicating elements remains the same. The logical to physical mapping can be updated (via modification of a table) to change the physical path of any data channel, but the application code need not change.

Communications between a SPA module's application code and the OK occurs through application calls to OK procedures. These calls reference communication paths logically. The logical to physical mapping defines a unique physical hardware path from the source module to the destination. For output requests, the OK packages the output items and transmits them across the appropriate hardware paths as specified by the logical I/O definition. For input requests, the OK processes the interrupts from the I/O devices, matching them to outstanding requests.

When an I/O activity passes some hardware recognized event, control is transferred to an OK task. The OK task determines the nature of the event by decoding hardware status registers. This OK task communicates such events to the OK procedure, thereby notifying the application software.

All communication with the SPA consists of application code in a send module passing messages to the application code in a receive module. Figure 4 illustrates the software configuration at this level of interaction. Note that the OK controls access to the I/O devices (serial I/O units or SIOUs and the NIUs or ports).

Note also that the RTS is included in the diagram but has no direct connection to the OK or the application tasks. The application tasks interact with the OK through procedure calls. The called entry points are dictated by the OK service requested by the application. Most of the services provided can result in the suspension of the caller; this is to allow the hardware to complete the service requested. To avoid blocking the progress of the application, it is typical that the application will define tasks to service such I/O calls to the OK. This is the situation illustrated in Figure 4. As with any task, the control and synchronization of these application defined tasks are taken care of by the RTS. (The request for RTS services occurs at the Ada level as task elaborations and processing. They come from the Ada ACCEPT and ENTRY-CALL statements, which, when paired, make up the Ada rendezvous providing for synchronizing and passing data between tasks).

A logical I/O Channel is defined by a starting point, an ending point, and the data path between the two points. The starting point refers to a module and its NIU to be used for the communication. The ending point refers to a module's destination code (the CNU port id of the destined module). The "data path" refers to the physical path traversed by the data from the starting module to the destined module. For NIU communication the path involves the MIN configuration and the modules and CNUs between the starting processor module and the destined one.

Obviously, the definition of the logical I/O Channels and the MIN configuration are closely coupled. Each logical I/O Channel must be pre-defined by the application for use by the OK in order for the communication to be successful. OK services are then obtained by calls to OK procedures associated with the function to be performed. The logical I/O Channel is provided as one of each OK procedure's input parameters; specific channel information is indicated by a Channel Index, that is passed with each OK service request. Logical I/O Channels operate in a logically independent manner, in that I/O on one logical I/O Channel is not blocked by I/O on another logical I/O Channel. This important fact means that multiple logical I/O channels can share a single physical data path.

Logical I/O channels are used for all types of trans

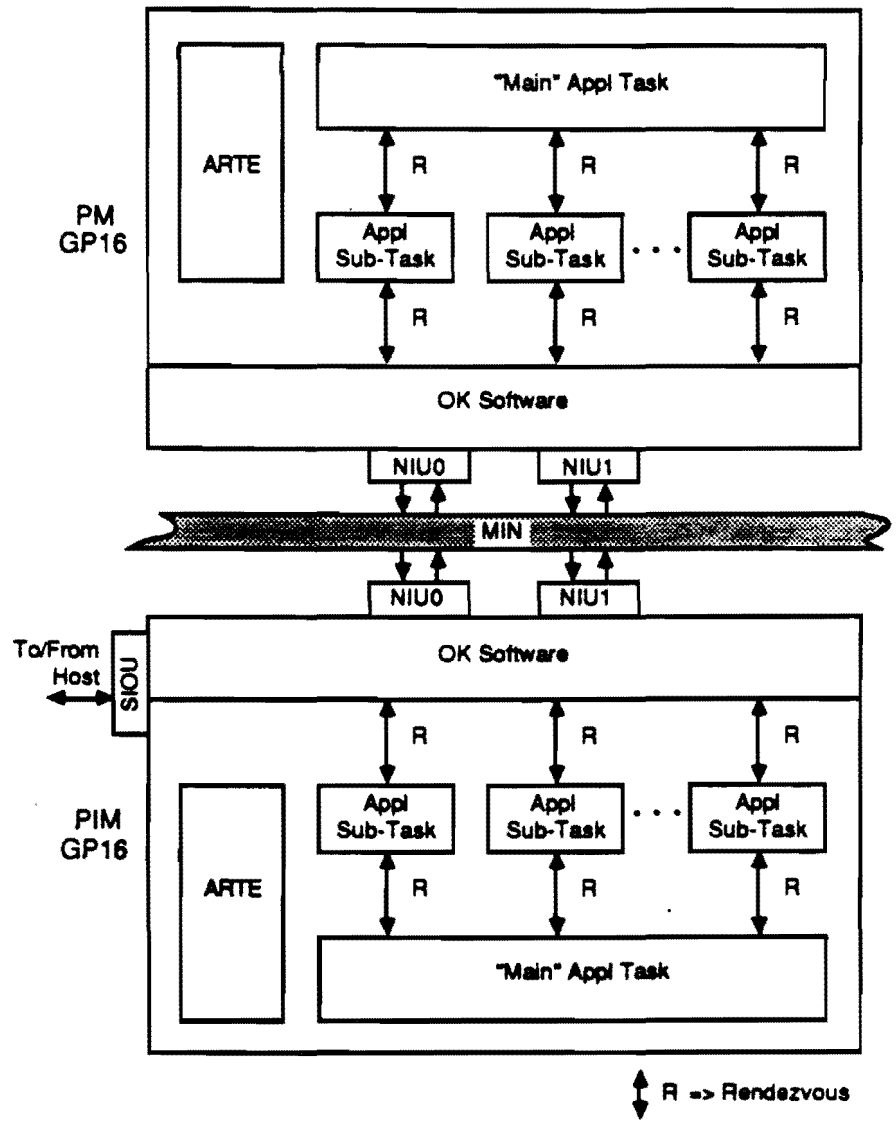


Figure 4. OK/Application Multi-Module Tasking Environment

fers: messages; data; external I/O commands; and CNU commands. The Channel Index is used to index the I/O Channel Table to determine the specific physical path for the transfer at hand.

For message transfers, the I/O Channel Table is used by the sending modules to determine which NIU, MIN destination code, and destination module's channel index to be used for the transfer. The destination module's channel index is part of the message packet. When received by the destination module, this channel index is used to route the message data to the appropriate application task. The I/O Channel Tables in the two modules can point to each other (that is, its destination code of one corresponds to the the other's channel index), giving the effect of a bidirectional logical I/O communication path for message transfers.

Unlike message transfers, block transfers pass only data to the destination module. Without wrapping a "message transfer communication protocol" around a block transfer (which is what MINIO does), block transfers are strictly unidirectional. There is no inherent information to tie two communicating logical I/O Channels together.

Figure 5 illustrates the interfaces to the OK from the application. The I/O Channel Table is the data structure of the I/O Channels used by the OK and its applications. The OK Driver provides asynchronous, unchecked I/O. MINIO provides handshake synchronized and verified I/O. Configuration provides hardware initial configuration and reconfiguration support. Heartbeat provides a SPA health monitoring support.

The functions provided by OK services are accessed by a high level interface (procedure calls). These functions are summarized as follows:

I/O Channel Table

This data structure is made up of I/O Channel records. The I/O Channel record structure is illustrated in Figure 5. Each record contains the information necessary for point to point communication using the OK Driver services. The application coder builds this array of records based on the requirements of an application.

OK Driver

The OK Driver performs the low level hardware manipulations required to send message, data, and hardware command packets (XIO and CNU commands) between processor modules within a SPA and between a processor module and the host. The OK Driver function also implements the logical I/O channel concept. The OK Driver provides low level access to SIOU and NIU devices, and to the MIN (CNU) hardware, both in the executing module and in other modules of the SPA. The OK Driver is required in all modules of a SPA. Figure 5 lists the procedures to the OK Driver service.

MINIO

This service provides verified I/O transfer functions.It implements a data block transfer protocol between two SPA modules. The protocol includes verifying that the transferred data arrives at the receiver correctly and returns a status to the application indicating whether the data transfer was successful or not. Figure 5 lists the procedures to the MINIO service.

Heartbeat

This service provides a health monitoring function for the application program. Once initiated, the heartbeat function creates MIN traffic by sending heartbeat messages at user defined time intervals and/or monitors incoming heartbeat messages verifying that received messages occur within user defined time limits. Heartbeat returns status to the application program when a missing heartbeat message is detected. Figure 5 lists the procedures to the Heartbeat service.

Configuration

This service supports the application in its efforts to initially configure or to reconfigure the SPA. Figure 5 lists the procedures to the Configuration service.

In a spaceborne environment, the need to suspend application software operation, reconfigure the array, and resume application software operation may arise. One method by which the need to reconfigure is made known to the application software is the failed heartbeat return function provided by the OK Heartbeat service. Typically, a "pecking order" is established in the application processors that determine which processor is to assume control for the reconfiguration activity. This decision can be based on the

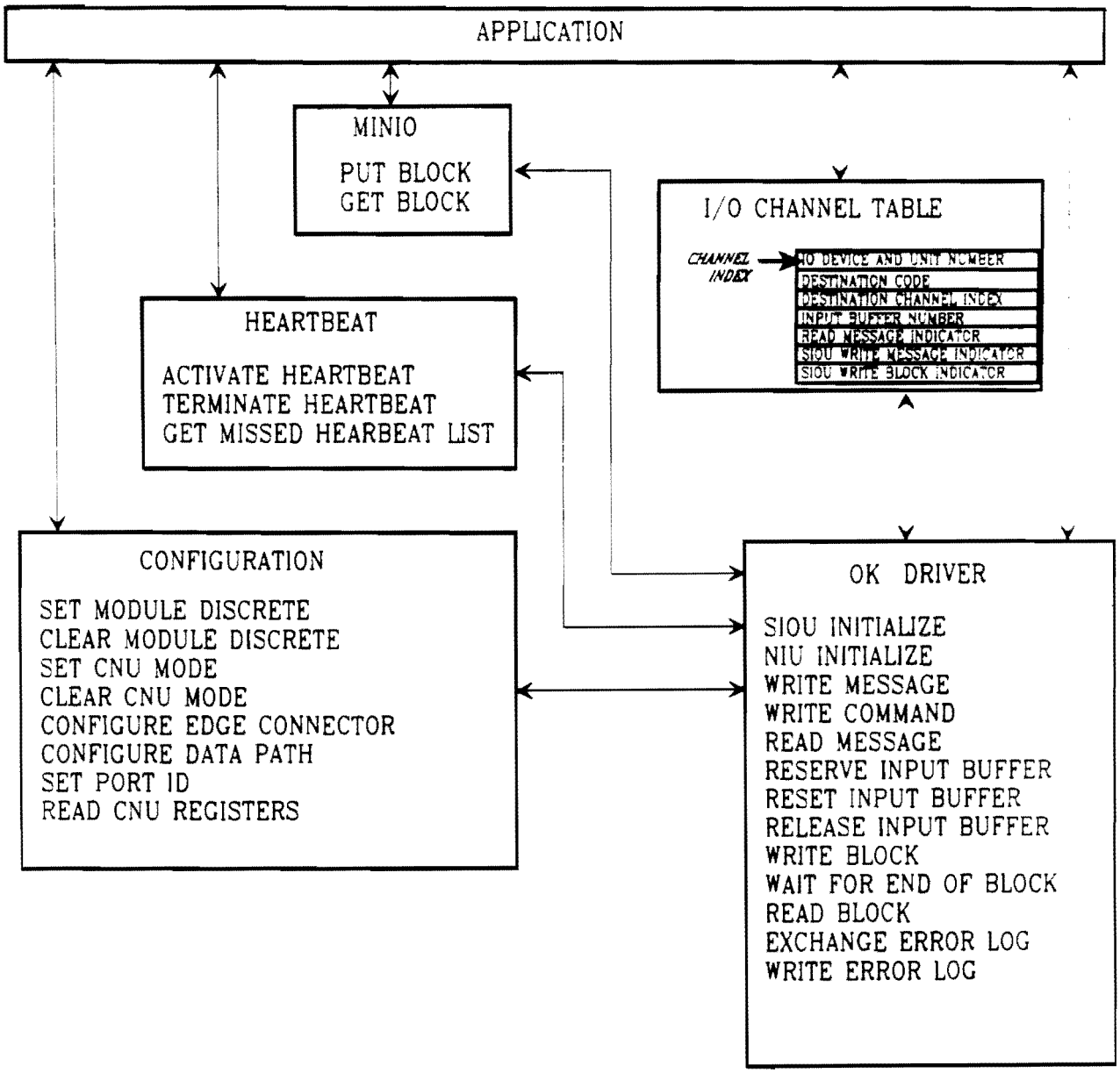


Figure 5. OK/Application Interfaces

data provided by the Heartbeat service. The controlling processor uses the OK Configuration services to repair the array and resume operation. Repair can be effected by powering on an additional processor, changing MIN paths, or both. The approach taken with the SPA-1 demo was to have this entire activity be the responsibility of the application programmer, by using the services of the OK.

SPA Concept Validation Demo

The purpose of the demo was to demonstrate the functionality of the main features of the OK software in the SPA-1 multiprocessor environment. An orbital ephemeris application written in Ada showed how the RTS handles tasking, memory management, and interrupt processing. MINIO was used to transfer the results of the ephemeris calculations to other processors on the SPA-1 system. Heartbeat was responsible for finding faults injected by the user during the demo, and the Configuration package was used to configure the CNU at all involved nodes to build a new path around the failed node.

The demo application consisted of ephemeris calculations being performed for two satellites, in each of three independent processors at different nodes on the SPA-1; this data (and SPA-1 health status) was sent around the path (ring) connecting the three active processing modules and ultimately out of the SPA-1 via a serial I/O (input/output) channel to a host machine (MicroVAX) for display. With the demo software running in the three active processing nodes, observers were invited to cause random nodal interconnect or processing hardware elements to fail by selection of switches on a fault injection panel. SPA-1, under the aegis of the OK, detected that a failure had occurred, isolated it, reconfigured around it, redistributed the processing load (to the two or one remaining active processors) and continued with the application processing, all without operator intervention of any sort.

The main features of the SPA concept validated by the SPA-1 demo were:

On-Board Control -

All data movement, fault detection, isolation and recovery was performed on the SPA-1. The host was used for display of status and of graphics showing the ephemeris calculations.

Data Movement -

Data generated by the ephemeris calculations in the active processor modules on the SPA-1 was moved around the data path to the master processor module (one of the two with serial I/O hardware connected two the host) and then sent to the host. All such data movement was done using OK facilities.

Fault Detection -

Faults were detected by means of the heartbeat software. Every active processor module periodically (twice per second) sent messages to other processors. Both of the processors connected to the host monitored the heartbeat messages, and when four in a row were missing, a fault condition was declared.

Fault Isolation -

From the pattern of missing heartbeats, the failed module or CNU was determined. This required that the monitor processor modules know the path configuration between the modules being used in the demo. The configuration was passed from the master monitor to the backup as part of the re-start after a reconfiguration.

Fault Recovery -

Fault recovery involved data path reconfiguration, application re-start, and load balancing. The rings were reconfigured by the master node following a fault, unless the fault was injected so as to remove the master processor (or its CNU), in which case the secondary/backup node began the reconfiguration, after waiting for a certain time period for the master to reconfigure. The backup then became the master, and all data communications were switched to the new master.

As part of the reconfiguration, all nodes except the master were reset (and thus put in the reset configuration shown at the top of Figure 1.) The heartbeats and application programs were then restarted in other nodes with a series of synchronizing messages.

When the demo began, each of three active processor modules was computing the positions of two satellites. After the first injected failure that involved the loss of any one of these processing modules, the failed processor's load was reassigned and shared by the other two remaining processors, each of which determined the ephemeris for three satellites. After the injection of an error which took out one of the

two remaining satellites, the entire load of six orbit calculations was given to the remaining processor module. At this point, any failure which removed this single remaining processor (or all paths to it) ended the demo.

It should be reiterated that the host or MicroVAX was used only to start the demo and to display health status/injected fault, and the orbits being calculated by the processor modules on the SPA-1.

Validation Software Implementation

The major software elements and their location are shown in Figure 6. The software in the SPA-1 drove the demo, while that in the MicroVAX kept its displays updated by repeatedly requesting satellite position update information from the SPA-1, and displaying the responses.

The SPA-1 software was distributed in nodes 1,3, and 8. Node 1 was initially the master; node 3 was the backup master (the only other node that had serial I/O hardware that could communicate with the host), and had a copy of the same software as the processor in node 1. Both nodes 1 and 3 kept checking that the ring was intact by using the heartbeat facility. In the event of a failure, node 3 would not reconfigure the SPA unless node 1 failed to do so in a predetermined time. Node 8 was always a slave node; it had no responsibility for fault detection/isolation (except in sending heartbeat messages as the specified intervals).

The master processor (whether it resided in node 1 or node 3) had three demo specific functions: to respond to the host's requests for satellite orbit data; to monitor heartbeat responses for ring or processor failures; and, to reconfigure the SPA-1 after a failure was detected and isolated. The slave module at node 8 had only a subset of the software in nodes 1 and 3.

(Re)Configuration

Once the failed module or MIN element was determined, reconfiguration was determined by a table look-up scheme. Table 1 shows the sequence of configurations used, the order of which is dependent upon the order of the failure occurrences. Configuration 0 is the initial configuration, and is shown in Figure 7. Other configuration examples are shown in Figures 8 through 10. Based on which element fails, the next configuration to be implemented is read from the table. If there is no entry, or if the reconfiguration is unsuccessful, then the "fall-back" configuration will be attempted.

Summary

An operating system called the Operational Kernel, or OK, has been completed, tested, and functionally validated in a demonstration using the SPA-1. This demo featured **fully autonomous** on-board control of data movement, fault detection, fault isolation, hardware reconfiguration, application re-start, and load balancing/redistribution.

With the demo software running in the three active processing nodes, observers were invited to cause random nodal interconnect or processing hardware elements to fail by selection of switches on a fault injection panel. SPA-1, under the aegis of the OK, detected that a failure had occurred, isolated it, reconfigured around it, redistributed the processing load (to the two or one remaining active processors) and continued with the application processing, all without operator intervention of any sort.

It should be pointed out that the orbital ephemeris application code was the only SPA based software written specifically for this demonstration. The other components were standard OK packages and supporting elements: heartbeat; MINIO; configuration (and package Ring_Builder); and OK Driver, used by all of the above to do message and data transfers, and to manage buffers. This demonstration effected autonomy and fault tolerance using an approach whereby the application was very closely coupled with the systems support; indeed, the structure of the application for this demo was built around use of the standard OK elements as a basic part of the application itself.

References

1 Grisbeck, G. S, Doleman, J.G., and Reed, R.G., "Autonomous Surveillance Satellite", 4th Annual AIAA/Utah State University Conference on Small Satellites, Logan, Utah, August 1990.

Details of the Operational Kernel and the SPA-1 Demonstration Software may be found in the following:

- "Software User's Guide for the Operational Kernel", #11920588, Control Data Corporation, Minneapolis, Minnesota, Feb. 1991.
- "SPA-1 Demonstration Software User's

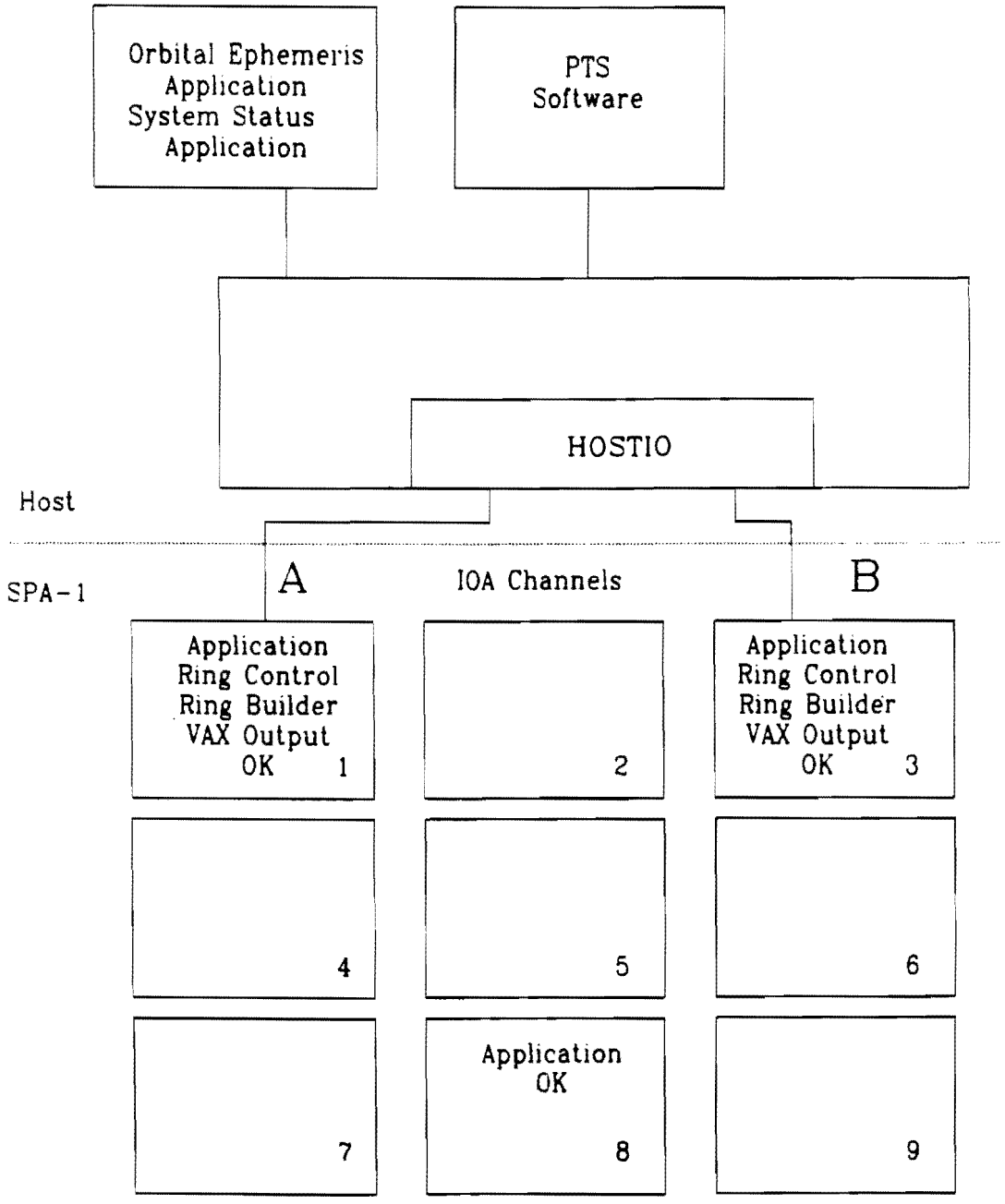


Figure 6. Major Software Packages Residency

Table 1. Configuration Sequence

No.	Config.	Next Config Based on Module Failure 1	2	3	4	5	6	7	8	9	Fallback
0	C12346789	1	2	3	4	0	6	4	8	6	C13789
1	C23689	0	0	0	0	0	10	0	0	0	C2389
2	C1346789	1	0	3	7	0	7	0	5	0	C13789
3	C12478	0	0	0	9	0	0	0	0	0	C1278
4	C123689	1	7	9	0	0	7	0	8	0	C123
5	C1346	0	0	0	11	0	11	0	0	0	C1379
6	C123478	10	7	9	7	0	0	0	8	0	C13789
7	C13789	10	0	9	0	0	0	0	11	0	C1379
8	C123	0	11	0	0	0	0	0	0	0	C1346
9	C1278	0	0	0	0	0	0	0	0	0	C1278
10	C2389	0	0	0	0	0	0	0	0	0	C2389
11	C1379	0	0	0	0	0	0	5	0	5	C123

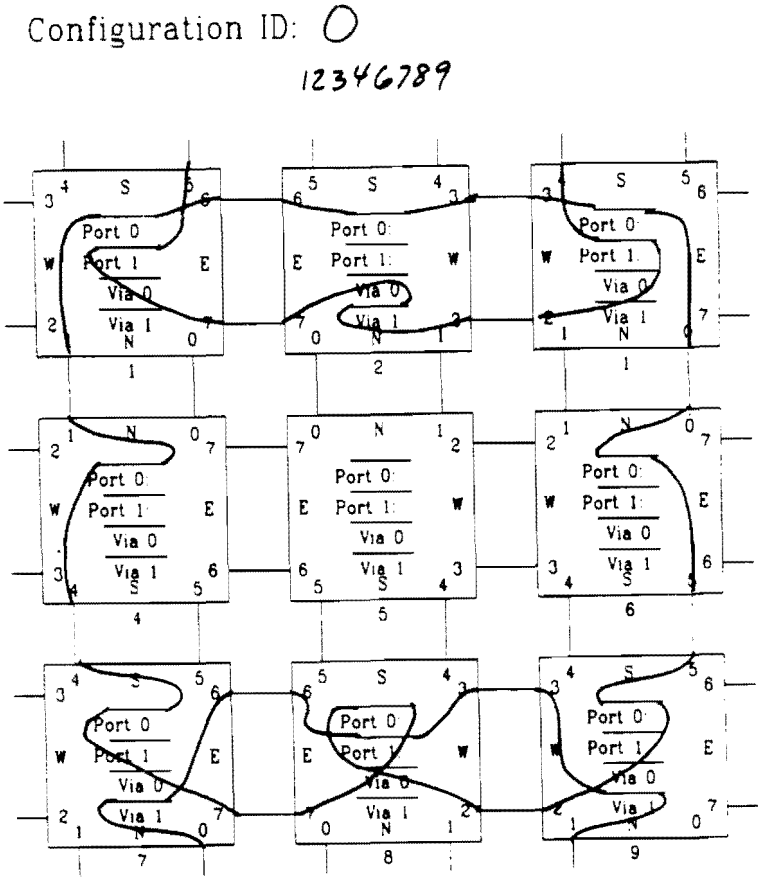


Figure 7. Configuration Example

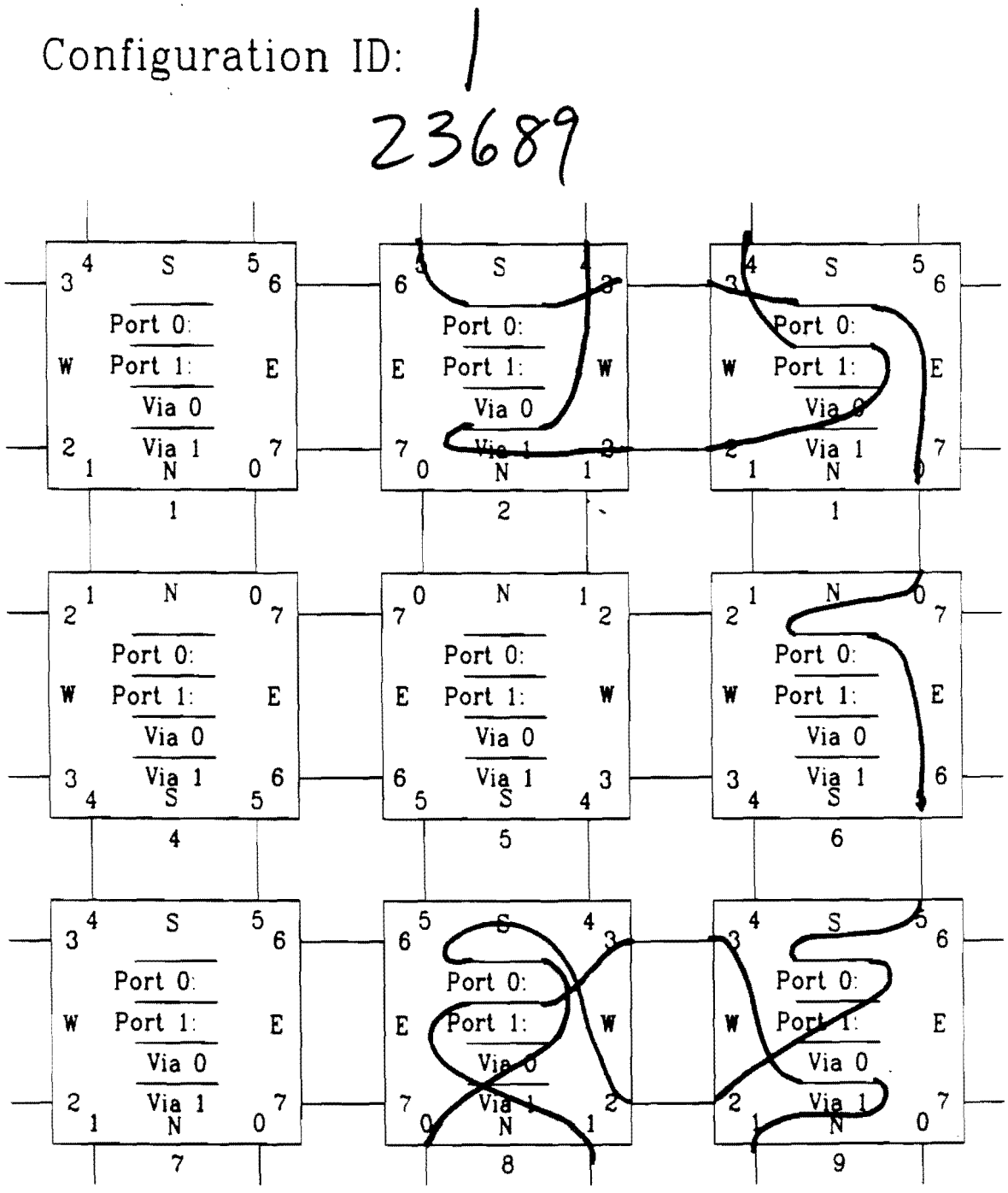


Figure 8. Configuration Example

Configuration ID: 2

1346789

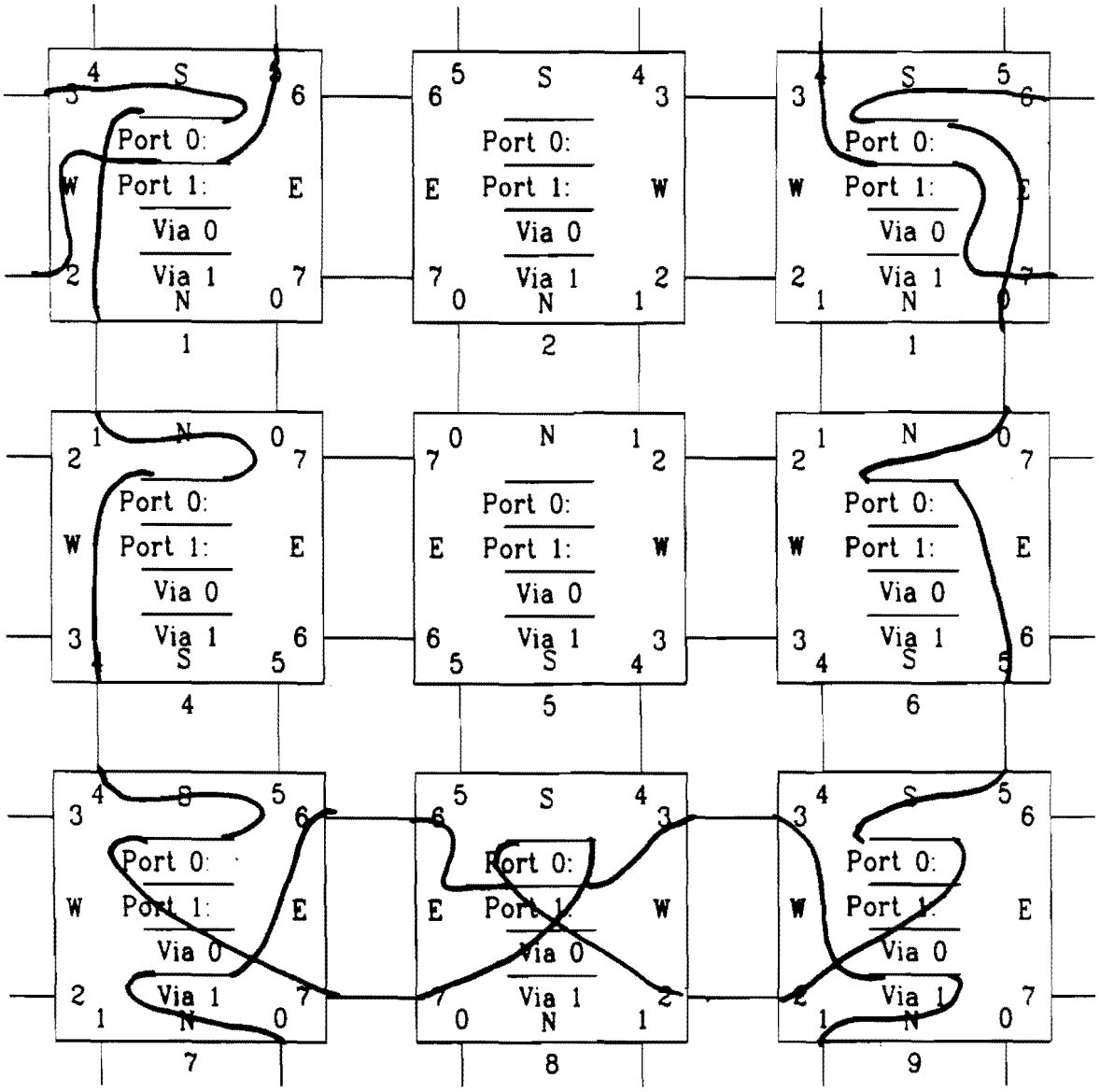


Figure 9. Configuration Example

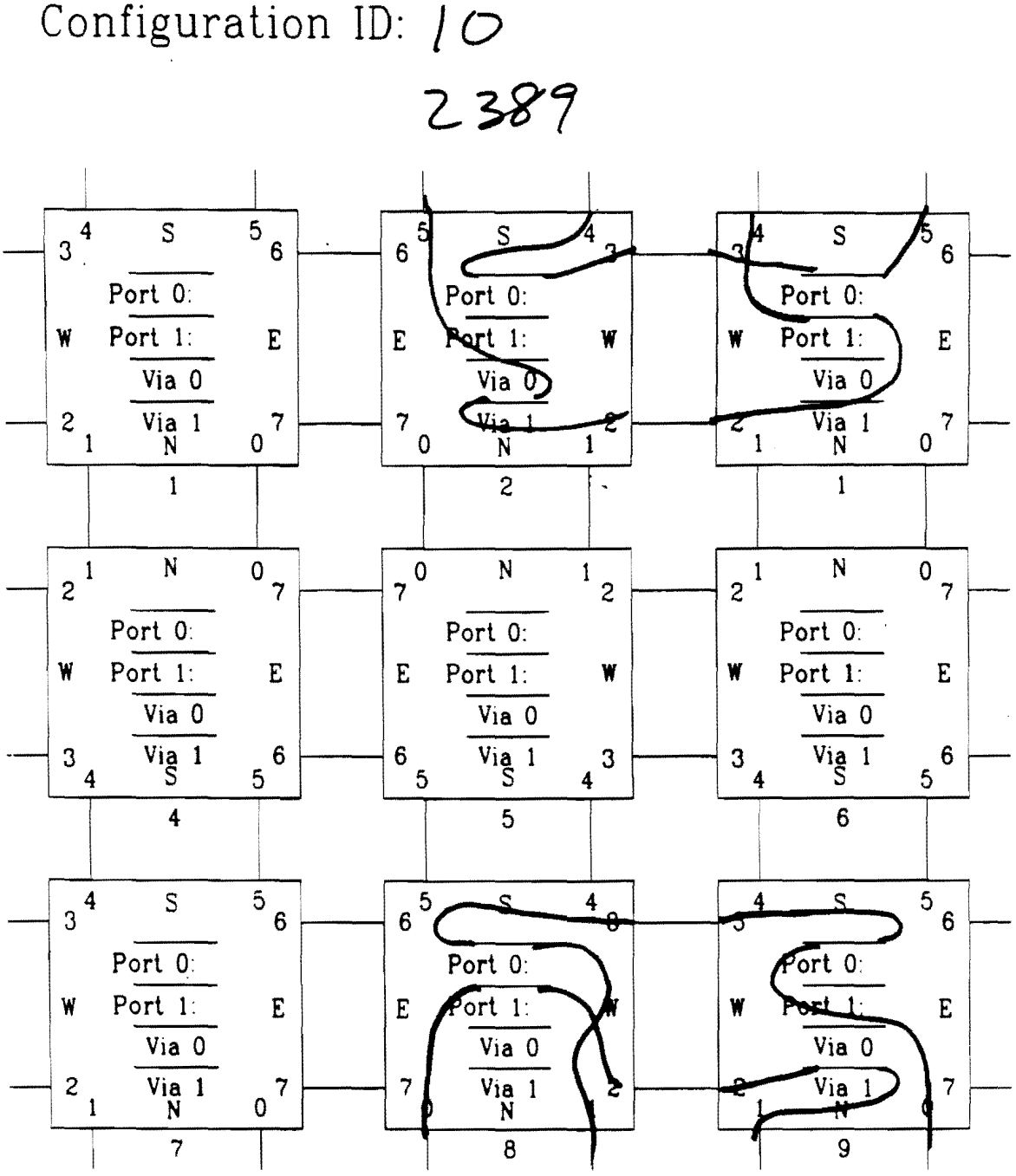


Figure 10. Configuration Example

Guide", #SPS-00482, Control Data Corporation, Minneapolis, Minnesota, Feb.1991.

Brief Overview of Ada Terminology

One of the goals of the Ada language was to make it possible to write huge programs effectively, programs so large that they could not be easily understood in their entirety. Another goal of Ada was to be able to write general software elements that could be used in several different programs. Thus, independently developed parts of Ada programs could be used as building blocks. The two most important kind of building blocks are subprograms and *packages*.

Subprograms are comprised of *procedures* and functions, and are very similar to subprograms in languages like FORTRAN and Pascal. A procedure call statement can specify parameters whose values are used, and may change variable values; a function, which also may have parameters, only returns a value when it completes its execution. This value is then substituted in the expression containing the function call.

A *package* is a collection of items (such as subprograms, variables, exception declarations, etc.) that are usable by other elements in an Ada program. Thus the desires for the ability to reuse software components and the ability to divide a large, complicated program into manageable elements are both met with Ada packages. Packages typically can be understood in isolation from the rest of a large Ada program.

Programs are often executed sequentially, i.e., the program's statements are executed in order. Ada programs can be written in this fashion. However, it is possible to have two or more sequences of actions, called *tasks*, to be performed concurrently. Often, a computer running a multitask program spends a little time running one task, then switches to another task, and eventually switches back to where it left off at some earlier task. This is called interleaved concurrency. In some environments, like SPA-1 where there are several processors, different processing modules may be executing different tasks at the same time. This is called overlapped concurrency, or just *concurrency* from here on.

Multitasking programs can greatly increase the speed at which programs finish their activities. Straight procedure calls are sequential, and their execution is suspended until necessary data or hardware activity is available. If tasking is used to effect overlapped concurrency, overall computations can be finished much quicker by allowing parts not dependent upon one another to execute simultaneously on other processors. Decomposition into different tasks allows work to proceed in certain areas while work (tasks) in other areas are waiting for some external event to occur. I/O operations are a good example of this. Ada programs have the additional benefit of being *reentrant*. Simply put, this means several tasks may execute the very same sequence of Ada statements at the same time.

In an environment like SPA-1, where no assumptions can be made about the relative progress of one task relative to another, the tasks are called *asynchronous*. From time to time, however, it is necessary for tasks to cooperate with one another or to synchronize their activities.

Tasks synchronize and communicate through a process called a *rendezvous*. A task communicates with another by either accepting a call to one of its own entries or by calling another task's entry. When one task calls the entry of another and the second task accepts that call, a rendezvous has taken place.

Concurrent multitask programming in a multiprocessor environment is much more difficult than sequential processing. One risk is timing, whereby a program depends implicitly upon the relative execution rates of several tasks. Another danger is called *deadlock*. Deadlock occurs when no task can proceed because each task is awaiting the result of, or some action by, another waiting task. The Ada program used in the SPA-1 demo software used approximately 60 tasks, and was a fairly subtle and significant programming effort.

Glossary

Ada	DOD programming language based on MIL-STD-1815A
CNU	Configurable Network Unit
DOD	Department of Defense
I/O	Input/Output
ISA	Instruction Set Architecture

MIN	Module Interconnect Network
MINIO	MIN I/O
OK	Operational Kernel
PTS	Portable Test Set
RTS	Run Time System
SIOU	Serial I/O Unit
SPA	Spaceborne Processor Array
SPA-1	SPA Version 1
1750A	ISA based on MIL-STD-1750A
444RR	444 Rugged/Reliable, a space-qualified 1750A processor designed and built by Control Data Corporation