A Data-Driven Operating System for Data-Driven Architectures of Real-Time Systems

BY

Richard M. Wallace
James E. McDonald
Duane O. Hague

System Avionics / Electronic Technology Divisions

Air Force Wright Aeronautical Laboratories
Wright-Patterson AFB, Ohio 45433

A Data-Driven Operating system for Data-Driven Architectures of Real-Time Systems

Abstract

In an experiment with the AFWAL/AAA AVSAIL DECsystem 10 and PDP 11 flight simulation equipment, redundant data bus transmissions were blocked to demonstrate that a large reduction in data bus transmissions do not degrade the effectiveness of real-time programs. The programs became "driven" by their input data rather than being "driven" by the synchronous, time-based interrupts. The experiment produced three major results: The first was that currently optimized real-time software does not have impaired performance with a hardware-enforced 80 to 90 percent reduction in its data flow. The second was that currently optimized executive and application software generates considerablly more data than is optimal for the execution of the real-time software. The third was that the optimal software structure using AVSAIL generation hardware and software has reached its technological limit. The next generation or rule-based, causality software is emerging from the previous generation of deterministic, synchronous software in the need for Data-Driven operating systems and architectures. A description is given for construction of a Data-Driven Operating System to take advantage of reduced data bus transmission rates required by the real-time software in order for the software to be "driven" by the production of data rather than synchronous, time-based, interrupt schemes currently in use. A design combination using Data-Driven computer architecture and a Data-Driven Operating System is explored providing 80 to 90 percent reduction of data bus loads with an estimated 50 to 60 percent reduction in central processor processing loads.

1.0 Scope Of Paper

This paper is written to introduce the networking system concepts, hardware architecture, and software concepts for fabrication of a data-driven system network. As the network has not been prototyped, only the functional attributes of data-driving are discussed. When specific hardware, processor timing, and system capacities are mentioned in this paper these data are extracted from the component manufacturer's data handbooks.

2.0 Introduction

Recent system performance measurements indicate that nearly 90 percent of real-time system avionics computer processing power is wasted. Future computer architecture designs will probably realize an order of magnitude improvement in processing speed by using parallel processing schemes, but these architectures still do not approach the wasted computer power problem now evident in real-time systems. The data-driven architecture described in this paper will allow current and future real-time computer systems to operate nearly an order of magnitude faster with features that promote revolutionary real-time system concepts.

There are conceptually three layers to the data-driven system (Figure 1). The outermost layer is the system functionality; how the complete data-driven system operates to complete a mission. The middle layer is the dissection of the system into its Local Area Network (LAN) configuration, while the lowest level of the system is the data-driven node itself. Further dissection is possible internal to the data-driven node and will be covered by this paper.

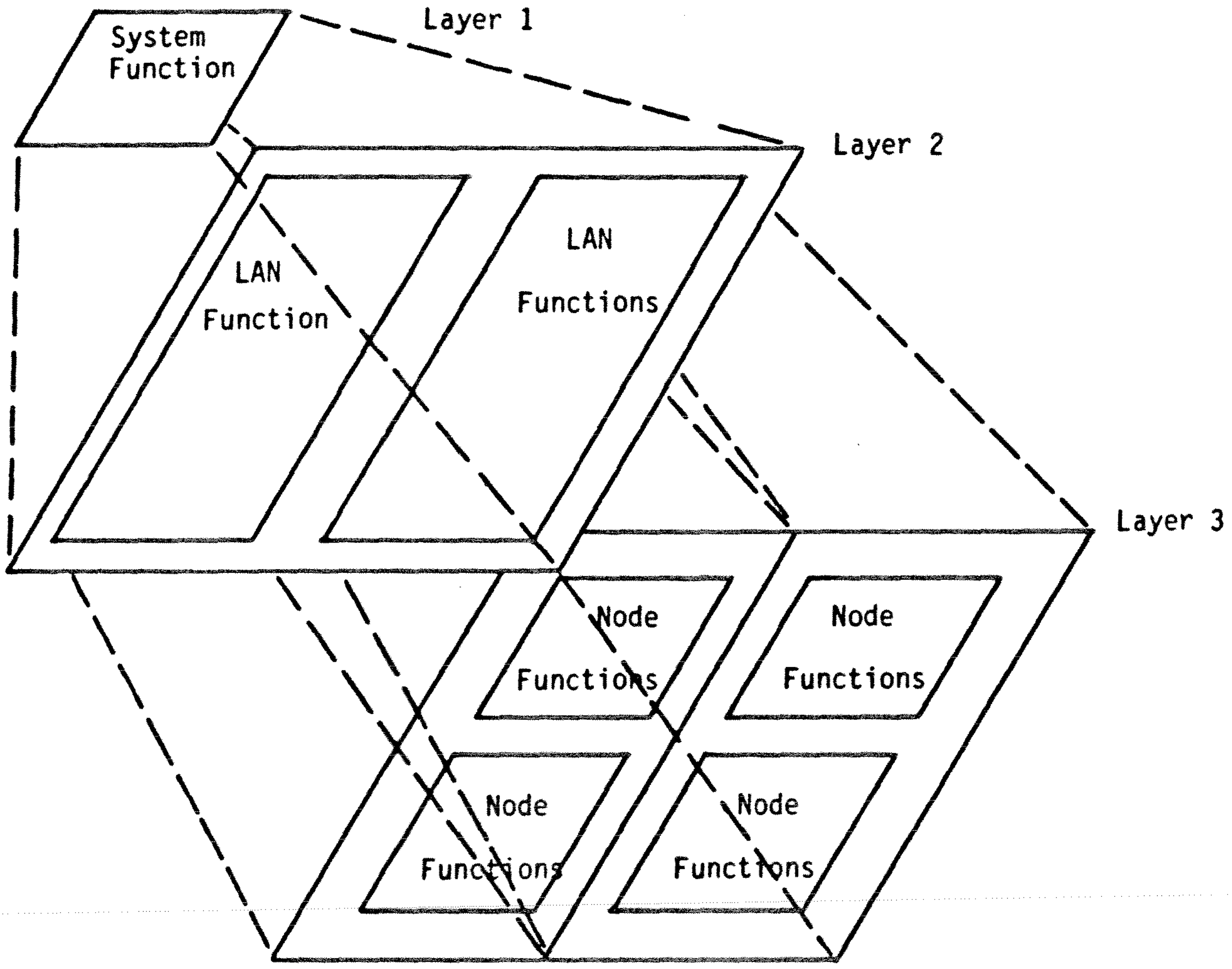


Figure 1. Layers of a Data-Driven System.

The data-driven terminology used in this paper implies that appropriate software is executed on its processor when data inputs to the software computation change their value by significant amounts. The term "data-driven" is used to denote the control method for the exchange and transmission of data on an inter- and intra-processor level for control of an entire computer system through the exchange of new/relevant data between computational processes. In the context of this paper the term "data flow" should not be associated with the term "data-driven." Data flow is the method of partitioning the actual computational algorithms across two or more physical processors and does not control passage of data between processes, or contribute to system control.

3.0 Data-Driven System Network

3.1 Data-Driven System Requirements And Design Sketch

After observing current network capabilities and program operation deficiencies in the real-time arena, the requirements and design of the data-driven information network for multiple real-time computer systems were formalized from an optimized blend of real-time applications of available network components. The requirements are:

1. Shared memory emulation. To the host computer operation software, the network must appear as a shared memory unit which implies that the network is nearly transparent. This permits autonomous or near-autonomous program execution. Reads and writes to the shared memories shall require no more host CPU time than conventional shared memories (0.5 to 1.0 microsecond). To increase the throughput of the distributed shared memory, a copy of the shared memory is to be located at each networked computer. Shared memory reads would involve only the computer's locally owned copy of the distributed shared memory while shared memory writes would update all copies of the distributed shared memory.

2. Serial linking. To avoid bulky and limited distance shared memory cables, high speed fiber-optics shall be used. The shared memory transfer times are accommodated and the distance between computers can be up to 10 kilometers.

3. Information-only transfer. To increase the efficiency of the network, the only items transmitted on the network are data variables that have been updated, or written to with new values. These new values are termed

"information." A data variable can only acquire a new value as a result of a write action by a CPU, or direct memory access, to the particular data variable. Therefore only write actions instigating new data variable values can instigate a network transmission. The information-only transfer gives birth to the data-driven concepts for the entire system architecture. It should be remembered that in conventional real-time systems, the majority of write actions to shared data variables' do not modify the variable value but still demand a network transmission.

4. Data addresses through vectored interrupts. When a computer receives information from the network, an interrupt shall be generated by the receiving node hardware directing the receiving computer to the software needed to process the information. This technique is employed to increase efficiency and to minimize the response time of the reacting software program.

With these requirements in mind, the initial block diagram design of the serially linked, shared memory ring network is depicted in Figure 2.

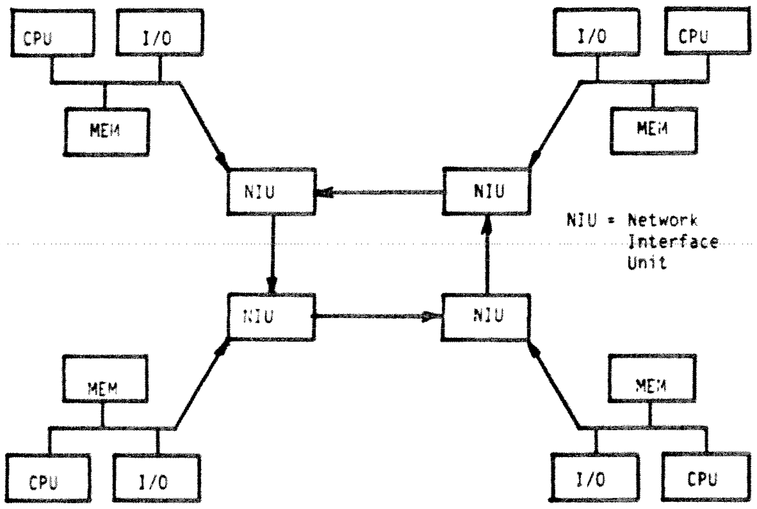


Figure 2. Serially Linked, Shared Memory Ring Network.

This block diagram illustrates the major component configurations:

1. Serial Communication Scheme

2. Serial/Parallel Transmission and Reception

3. Distributed Shared Memory

4. Information Detection

5. Data Addressing Through Vectored Interrupts

4.0 Data-Driven Hardware Architecture

At first glance, the data-driven architecture looks very similar to a standard distributed multi-microprocessor system with a LAN interconnection scheme. However, there are two primary innovations in the data-driven hardware architecture. The first innovation is the hardware and data structure of the data-driven LAN (DDLAN), and the second innovation is that the hardware structure of each data-driven processing node is designed to provide optimum support to the Data-Driven Operating System software. The details of the data-driven processing node structure are described in the following sub-sections. The general form of the data-driven architecture is a ring network with nominally sixteen nodes. While the network protocol could be modified for more nodes, it is not likely that more than sixteen nodes will be used in an avionics application.

4.1 General Form Of The DDLAN.

The data-driven network uses a single line asynchronous slotted-ring manchester protocol (nominally 25 megabits/second) with distributed network control. The network is fiber-optic based where each node has a fiber-optic bypass switch so that off-line nodes do not interrupt the ring network. The network protocol is designed to support dynamic network reconfiguration with a 5 millisecond data stoppage and automatic recovery.

4.2 The Data-Driven Processing Node.

The Data-Driven Processing Node (DDPN) is actually a dual processor configuration consisting of a Data-Driver Processor (DDP) and a Main Processor. For the purposes of initialization the data-driver is a slave processor. However, in normal operation, the two processors are asynchronous with communication through FIFO process queues and dual-port data table memory. All process variables are stored in the data table common memory along with each variable's process control descriptors. While logically a single common memory, the data table is physically distributed between different portions of the node.

4.2.1 Data Table Monitor.

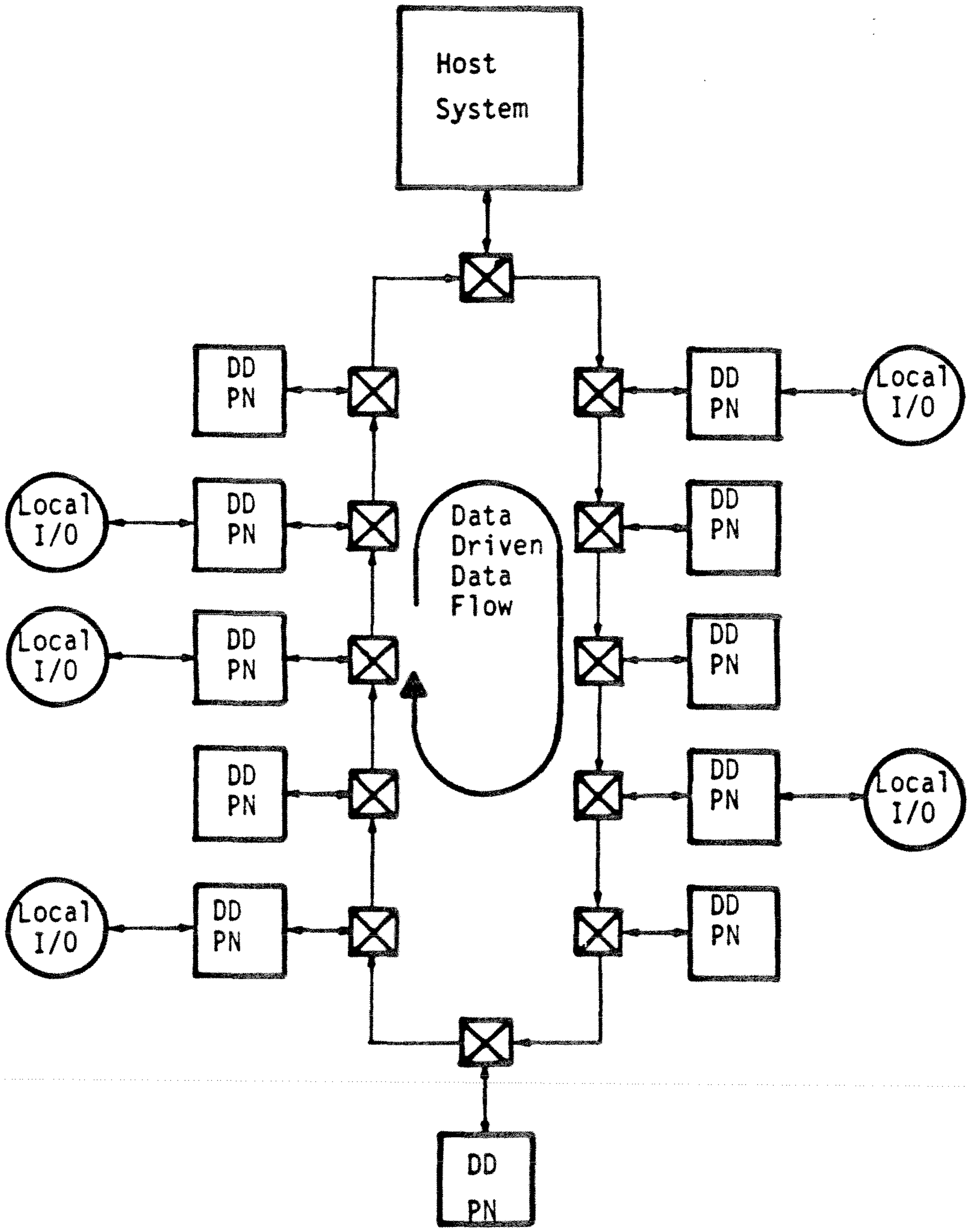
Two embedded hardware functions that operate independently of the system software are the key to the data-driven node. The first embedded function is the Data Table Monitor (DTM) which is invisible to the Main Processor. Its function is to trap any Main Processor write to a global variable's address where the new value is different from the old value. This results in the new value and that value's address being loaded into the input FIFO queue of the Vector Driver, which is the second embedded function.

4.2.2 Vector Driver.

The Vector Driver input queue is always loaded with address/data pairs received from the DDLAN along with the node's internal address/data pairs. The Vector Driver uses the address of the address/data pair, plus an offset, as the address of a look-up table entry to obtain the vector of the Data Driver routine that is to handle the associated global data variable. This new vector/data value is then loaded onto the Data Driver processor's input FIFO queue. As a vector/data pair come to the top of the input queue, the pair is dequeued and used in the routine that jumps to the appropriate Data Driver routine for that data type. The process continues until the queue is empty at which time the routine stays in a busy-wait with interrupts enabled. The handling of the Data Driver and Main Processor FIFO queues is "ready" status driven with provision made for program interrupt-on-ready or a "watch-dog" service time-out on "ready" interrupt.

4.2.3 Data-Driven Local Area Network.

An example of the data-driven architecture is shown in Figure 3. This example could be applicable to a real-time distributed avionics system or a distributed real-time simulation system. Individual DDPNs might have local input/output functions to a low order servomechanism.



⊠ Fiber Optic Bypass Switch

DDPN = Data - Driven Processing Node

Figure 3. An Example of a Data-Driven Local Area Network.

4.2.4 DDPN Processing Speed.

The basic requirement of the DDPN is that the DDP must process the incoming data flow faster than the Main Processor. The DDP must also be able to handle all data types used in the distributed system. While these requirements could be met by a raw processing speed difference, the DDP routines are designed to be very short with simple arithmetic functions compared to the Main Processor's computational routines. Thus where the DDP and the Main Processor have equal throughput, the difference between average and worst-case DDP/Main Processor service routines can be handled by adequate queue lengths with queue-overflow fault handling routines.

4.3 Specific DDPN Hardware Components.

The data-driven architecture is very compatible with future high speed processors. A prototype DDPN could be implemented with any of the latest commercial single-card microprocessors with full arithmetic capabilities (i.e. fixed and floating point data types) and with any system bus structure. Hovever the practical "form-fit" aspects of implementing a double system bus within each DDPN limit the reasonable choice of a node bus to the VME(A24D16) or DEC Q22BUS type buses. The memory management structure of a particular microprocessor has a synergistic effect on the implementation of data-driven software. Based on current availability, the best CPU for a data-driven prototype would be the National Semiconductor 16032 Microporcessor (with floating point data types and memory management) using the VME bus. The second choice of equipment for prototype fabrication is the DEC KDJ11-A (LSI 11/73) Microprocessor using the Q22BUS.

For the purpose of simplicity, futher discussion of the data-driven prototype will assume use of the KDJ11-A and Q22BUS. A block diagram of the DDPN is shown in Figure 4. The KDJ11-A is used as the CPU for both the Main Processor and the DDP. The Main Processor has significantly more system resources than the DDP to handle the complex computational algorithms.

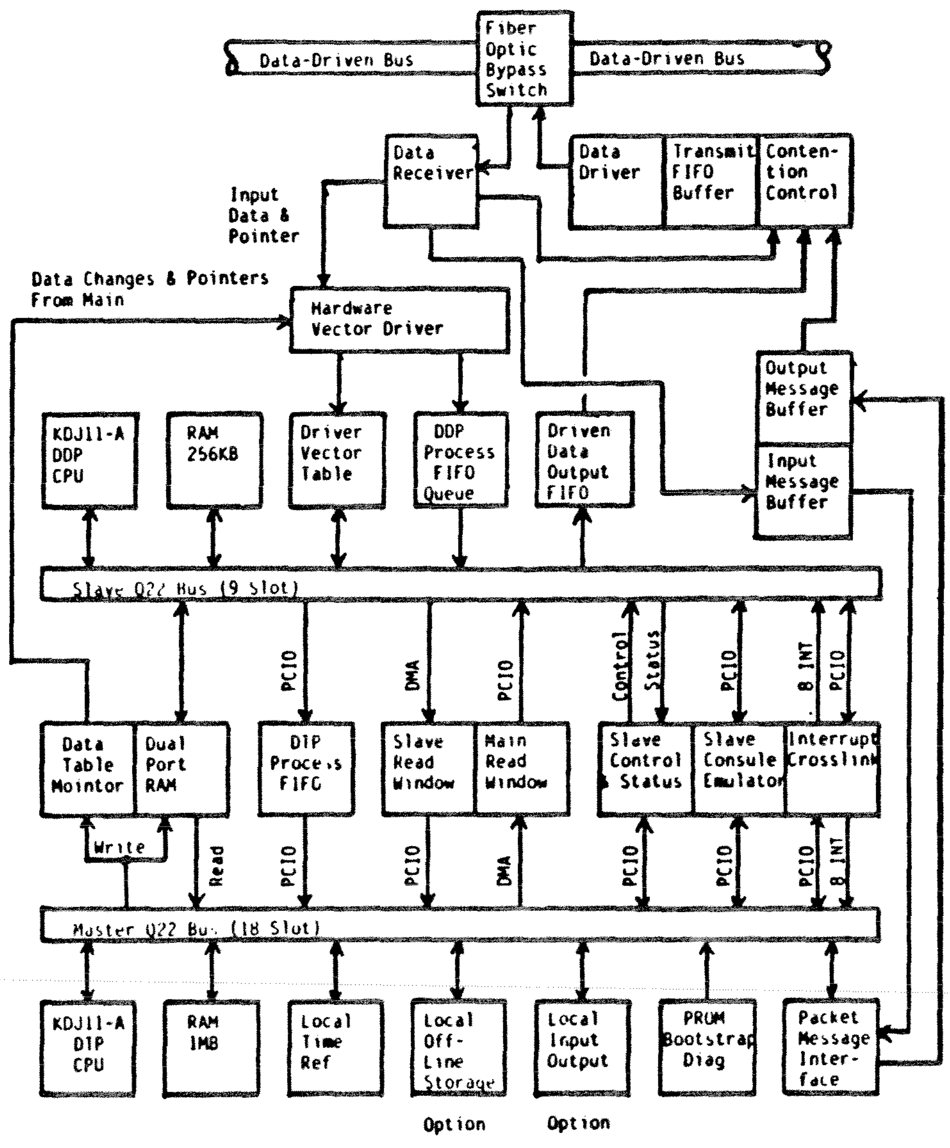


Figure 4. A Block Diagram of the Data-Driven Processing Node.

4.3.1 DDP And Main Processor Interconnections.

There are three levels of interconnection between the DDP and the Main Processor. The first level is for initialization and basic control. A design feature of the KDJ11-A is that it contains microcode support for a console terminal that can modify memory and start programs. Using the first interconnection level, the DDP can act as "system operator" of the Main Processor as well as providing for eight crosslinked, maskable program interrupts for each CPU. These interrupts can be set via software in the DDP CPU.

The second level of interconnection is via "read-windows." Each of the CPUs can read the memory contents of the other CPU via a register-mapped window. This allows each of the node's CPUs to check the status of the flags in the other CPU as well as read data blocks from the other CPU.

The third level of interconnection is at the data-driven level. At this level, communications are a function of the Data Table Common Memory and the FIFO process queues. The software of each CPU is normally "queue-ready-status" driven, but both process queues have provisions for program interrupt on either queue-ready or service-time-out interrupt on the queue-ready.

4.3.2 Data Table Common Memory Requirements.

The size of the data table in each DDPN must be identical or larger than the total number of defined variables in the entire DDLAN. If a particular node does not use a variable, then the pointer that is defined in the Vector Driver look-up table must be null so that the software can discard that data reference and not waste time and queue space by loading the vector/data pair onto the DDP input FIFO queue. For prototype purposes a maximum of 4096 entries in the Vector Driver look-up table (a 12 bit address) is sufficient; although the hardware can be adapted for another table limit.

4.4 Time Input To The DDPN.

An important aspect of data-driven operation is that time is an explicit variable in all operations. This is radically different from traditional synchronous software where time is implicit in interrupt processing. Time is handled by the Local Time Reference Module connected to the Main Processor. For example, if there are six time intervals of interest to the computational algorithms where each interval is a multiple of clock ticks, six data table entries are defined as time variables where intial values define interval relative phasing and the significance threshold determines the period. On each clock tick, the Local Time Reference Module writes current time to the six time variables by direct memory access. This results in the sigificant time variables being processed by the DDP and then the Main Processor as time becomes relevant to the computational algorithms in the Main Processor. All this occurs without interrupt to the Main Processor.

4.5 DDLAN Packet Switching Emulation.

A final point on the DDPN, indirectly related to the data-driven operation, is the Main Processor's additional interface allowing the DDLAN to emulate a message packet-switching network for node-to-node block transfers of data as a background task to the normal data-driven transfers. This function allows the downloading of the system and application software from an external mass memory.

4.6 The Data-Driven Local Area Network.

The basic structure of the DDLAN is a unidirectional serial ring topology where a configuration bypass switch is provided for each node in the ring. A serial ring topology is ideal for data-driven operation since all new data transmitted on the DDLAN must be received by all nodes. Selection of the data to be processed within that node is accomplished by the internal data-driver software. Conceptually the DDLAN may be viewed as a continously circulating series of data slots (Figure 5). Each node receives slots from its predecessor node and transmits them to the successor node. In a simple sense, all slots are either empty or full. When a node receives an empty slot, the node has the option of filling the slot with an address/data pair before retransmission of that slot. When the filled slot has circulated around the ring back to the source node, the slot is converted to empty and is passed on to a successor node. When a node receives a full slot from any other node, the address/data packet is copied to the Data Driver FIFO before slot retransmission. Because of the transmission of an individual address (pointer) with each data value, a DDLAN is only about 50 percent efficient for any given network bandwidth. However, experimental testing on a synchronous, distributed, multiprocessor system (AFWAL AVSAIL) has shown that, conservatively, 80 percent of the data flow packet switching information is redundant (i.e. has no information value). Thus a DDLAN would only require about 40 percent of the transmission bandwidth required by the equivalent data flow packet switching network. For a given network bandwidth, elimination of transmitting redundant data provides about 250 percent more informational data transfers than a data flow packet switching network. This improvement in efficiency easily justifies the increase in hardware complexity required to support data-driven networks.

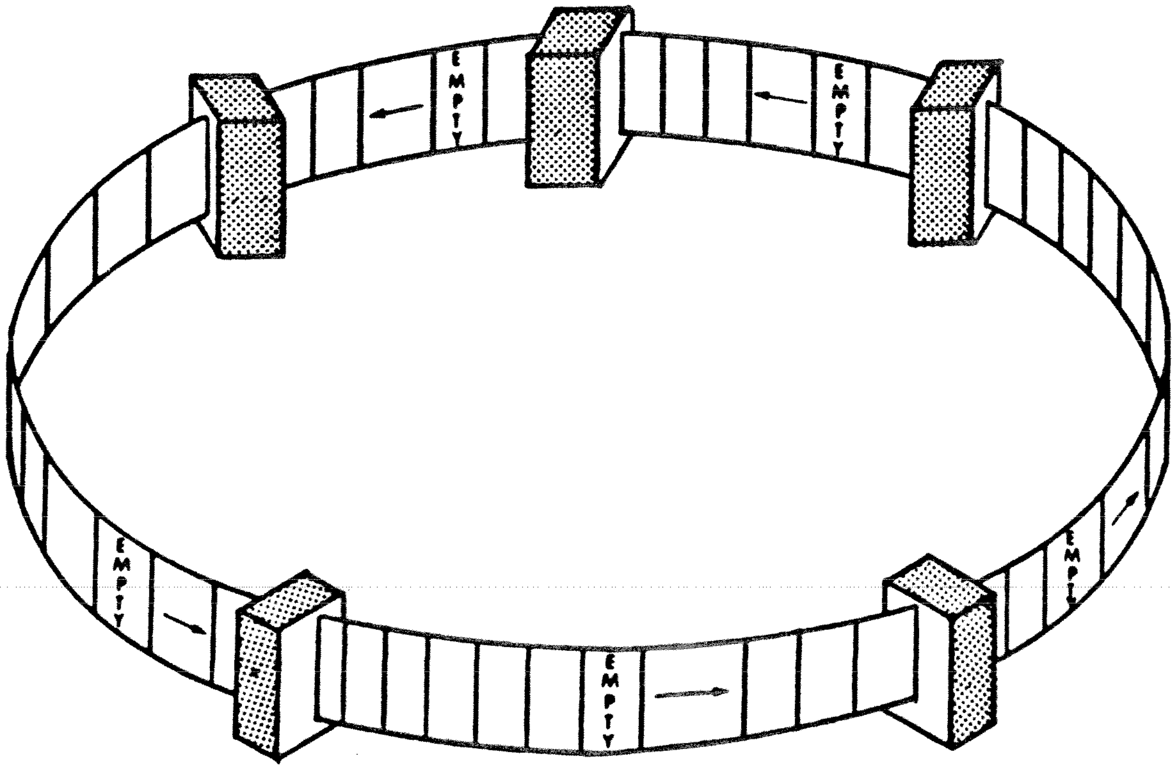


Figure 5. DDLAN as a Circulating Series of Data Slots.

The DDLAN ring approach has major advantages over alternative network architectures. The advantages are:

1. Distributed control with optional node by-passing for fault tolerent operation.

2. Highly efficient use of the available network transmission bandwidth.

3. Improvement in the ease of expanding a network's node count.

4. Automatic equal distribution of the available bandwidth among all network nodes contending for DDLAN access with hardware contention resolution.

5. All nodes "see" common data which eases partitioning and distribution of system and application software. This allows all real-time evaluation data acquisition functions to be centralized in a single network node.

6. The DDLAN can perform a hardware emulation of a node-to-node packet switching network as a background task to data block transfers.

7. The DDLAN is ideal for fiber optic implementation which provides a higher bandwidth network while minimizing physical distance and routing problems of a distributed network.

4.6.1 <u>The DDLAN Packet Format.</u>

DDLAN is best explained by a discussion of an implementation prototype. The basic data slot structure is a 36-bit, manchester-encoded time slot where all slots are separated by a minimum of two-bit transmission times for a 1.5-bit time time-out as explained below (Figure 6). Each DDLAN receiving node must view slot reception as an asynchronous operation. Due to the small size of information packets, the DDLAN can not afford the long synchronization preambles used in serial packet switching networks.

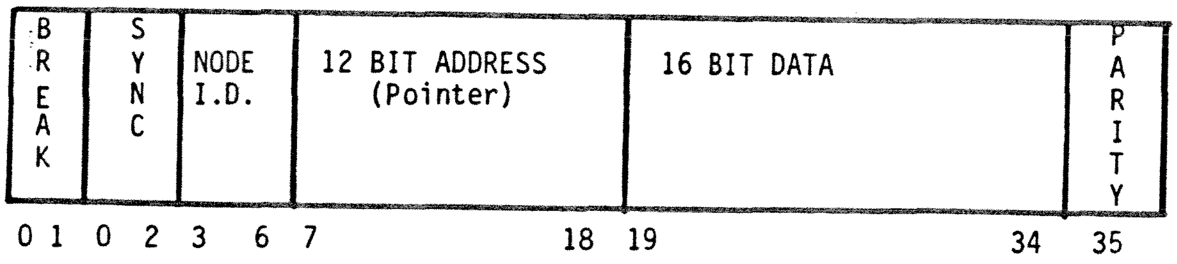


Figure 6. DDLAN Data Slot Structure.

The basic DDLAN slot consists of a 3-bit synchronization pattern that identifies "start-of-slot" and "type-of-slot", 32-bits of information, and a parity bit. The synchronization pattern is one of three readily detected invalid manchester patterns which identify a slot as either an empty slot, a full slot, or a message slot. For all slots the parity bit is based on the 32-bit field with all subfields being transmitted most significant bit first.

The information field for a full slot would be a 4-bit source node identification, followed by a 12-bit address (pointer), followed by a 16-bit data word. Execept for the fault condition described later in the node protocol definition, an empty slot has the same format as the full slot with the pointer and data as zero.

The message slot has an information field of a 4-bit source node identification, followed by a 4-bit target node identification, followed by a "start-of-message" flag, followed by an "end-of-message"
flag, followed by 2-bits reserved for future use, followed by 16-bits for a "count-down-byte-of-message" counter, followed by 8-bits of data.

No commercially available network systems are suitable for a data-driven prototype. A prototype has yet to be built and should have the highest practical bandwidth available. Commerical asynchronous manchester decoders currently only support data rates up to 2.5 megabits/second. A design by Mr. Hague has been developed based on newly available components (normally used in radar) that would allow asynchronous manchester decoding with 80 percent confidence of acheiving 25 megabits/second and 99 percent confidence in acheiving 15 to 20 megabits/second transmission. The design is based on use of nanosecond programmable delay lines and two-level enabled edge synchronized 50 megahertz (25 megabit data) Schottky Square Wave Generator Modules (part number: MDSWGM-TTL-50 from Engineered Components Company). These components are being obtained to verify this design. The design operates by using the manchester waveform to reconstruct the manchester phase-clock waveform synchonized to the manchester waveform to $\pm$ 5 percent instantaneous accuracy at 25 megabits. Lower data rates of course give better accuracy. This same design also provides recognition of manchester activity time-out in 1.5 bit transmission time which is the reason for the 2-bit minimum gap between slots. This gap also prevents decoding errors from propagating into succeeding slots. The reconstructed phase-clock allows the manchester waveform to be converted into a sequence of edge transitions for "data-one", "data-zero", "phase-one", "phase-zero" and/or a missing transition (either of: data/phase/one/zero).

This technique allows for detection of the synchronization waveform types, reconstuction of the information, and simultaneous checking for correct manchester format. Besides format testing, the slot information field is also tested for parity and for having the correct data bit count before the slot gap is detected. Prior use of this technique indicates that it provides hardware detection of all transmission errors to a confidence level of at least 99.99999 percent (the "7-nines Confidence Level").

4.6.2 Implementation Of The Fiber-Optic Node Bypass Switch.

The choices in the fiber-optic field are still somewhat limited due to the low number of manufacturers, however Frequency Control Product, Incorporated produces an ideal optical bypass switch in their Model SW-T2; which comes in two versions.

Model SW-T2 is electrically activated with spring-return--to-bypass. This version allows node attachment control by the node with automatic power-off line bypass. Model SW-T2F is a bistable latching version where state switching is accomplished by steered current pulses. This version allows the on-line/off-line condition of one node to be controlled by other nodes. For prototype purposes the selection is for the SW-T2 model. It should be noted that the SW-T2 switches states in 5 milliseconds and that no optical information will pass during the switching transient. This characteristic as well as "optical-switch-bounce" can be handled automatically by the DDLAN protocol.

4.7 DDLAN Protocol.

The DDLAN protocol is identical in all nodes. There are two parameters effecting node operation that are programmable within each node. These parameters are "slot-wait-count-for-echo" and "error-retry-counter." While it is feasible to have hardware detection of network activity within each node, it is recommended that polling be done for all possible nodes through data-driven system software or through the DDLAN packet switching mode. The following are the DDLAN protocol rules in descending order of importance. Each slot contains the node identification of the last node that created/used that DDLAN slot.

- o If no input is detected for 32 slot times, transmit empty slots with an all ones address (pointer) until input is received. Proceed to the next protocol.
- o If no input is detected for one slot time, transmit one empty slot.
- o If an empty slot with all ones address (pointer) is received, abort data transmissions in progress, retransmit received empty slot, and wait for normal empty slot (i.e. all zeros).
- o If a normal empty slot is received and the transmission queue is more that half full, destroy the empty slot.
- o If a slot is received with error, destroy the slot. Note that failure to recongize the slot synchronization waveform will have the same effect as this protocol.
- o If more than 32 slots are received while waiting for echo of a node's last transmitted full slot or message slot, decrement the retry counter. If the counter underflows, notify the Data Driver processor of a network fault, else restore the information field for a transmission retry.
- o If a full/message slot is received that matches the "stored-last-node" transmission, destroy that slot and transmit an empty slot.
- o If a full slot is received with a different node identification, copy the information field to the Vector Driver and retransmit the full slot.
- o If a message slot is received with a matching target node identification, copy the information field to the Main Processor message interface and retransmit the message slot.

- o If an empty slot is received while the node is waiting for a full slot or message slot echoing from a prior transmission, the empty slot is retransmitted.

- o If an empty slot is received when a data-driven address/data output is ready or a message field is ready, destroy the empty slot and transmit the appropriate full slot or message slot. Then inhibit further use of empty slots until the echo matches or time-out occurs. The data-driven output has priority over the message output.

5.0 Software Concepts

5.1 The Logical Structure Of The Data Driven Node.

In Figure 7 the logical structure of the DDPN shows that the correspondence between the physical components comprising the node have almost a literal mapping to the software control entities comprising the Data Driven Operating System (DDOS). The Vector Processor (Vector Processor Operating System -- VPOS), the Main Processor (Main Processor Operating System -- MPOS), and Data Driver (Data Driver Executive System -- DDES), are control entities that contain system software operating at increasing levels of complexity.

Logically the Vector Processor consists of an input FIFO queue, a vector table and VPOS primatives which reside in RAM, and an output FIFO. These components are manipulated by the DDPN's most rudimentary system software; the VPOS. Its limited functions provide intelligent facilities for one-to-one mapping of global data addresses to Data Driver primatives.

The Main Processor's logical composition is defined as an input FIFO from the Node Bus, a data table read/write window for the two port data table RAM, and the local memory RAM containing the computational processes and the MPOS primatives. The Main Processor has the next level of complexity in operating systems (indeed a quantum level increase) by its primatives being able to control a multiprogrammed environment.

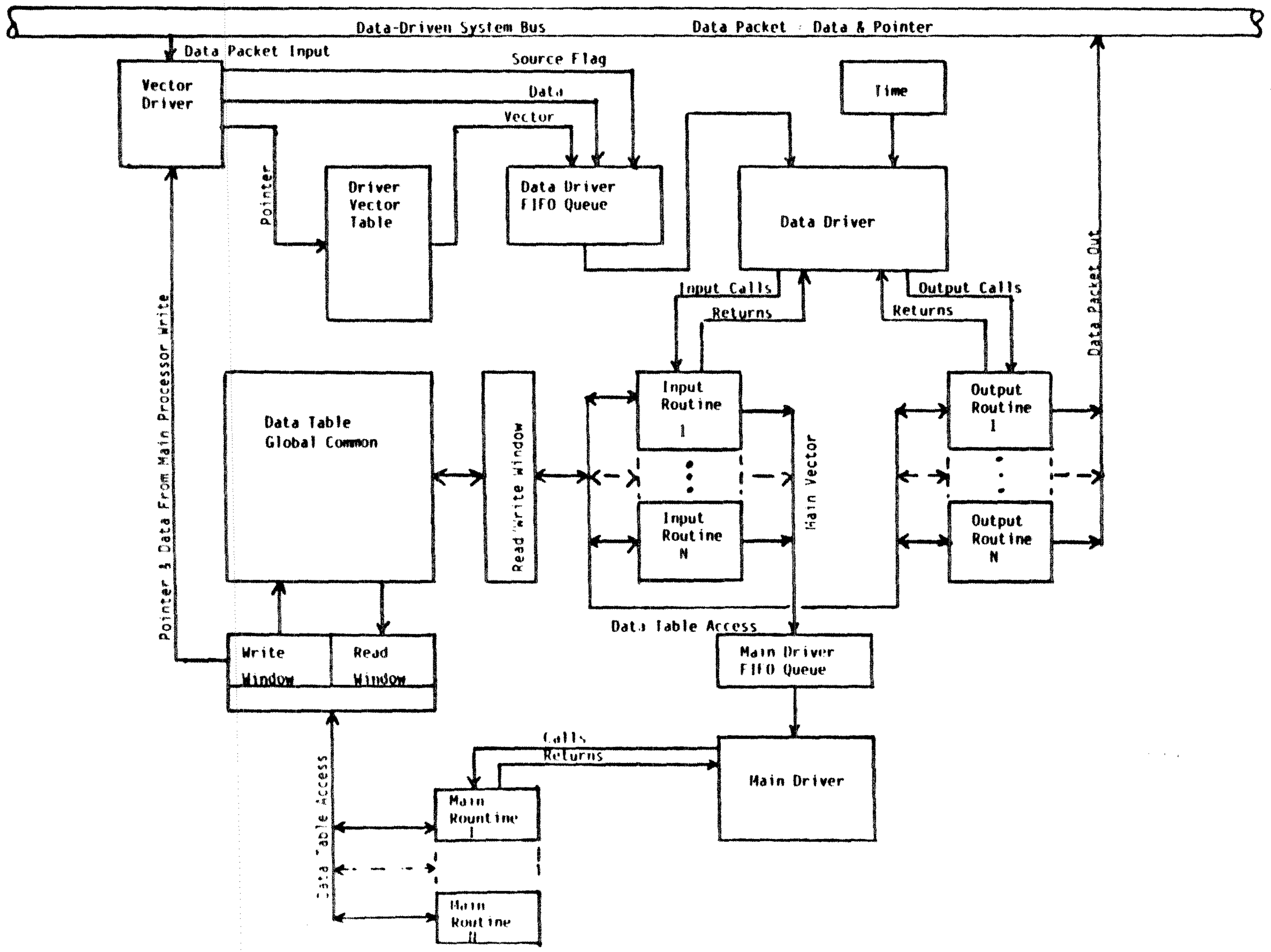


Figure 7. Data-Driven Processing Node Logical Structure.

The logical composition of the Data Driver is similar to the Main Processor. The Data Driver has an input FIFO from the Vector Processor, a data table read/write window for the two port data table RAM, a FIFO buffer to the System Bus, and a local RAM which contains the logical control primatives for the Main Processor and the DDES primatives. The most important control software of the DDPN is the DDES. This system software is simple in its organization to allow high throughput. The system is based on a prioritized, batch-queue model in which the greatest amount of throughput is the design goal. It is in this executive that the software compliments the hardware. By placing _all_ logical program decisions for the particular node's mission software into a _separate_ processor, the computational processes executing in the Main Processor can continue without interrupt. This is the premise upon which the DDPN system software is built.

The necessity of interprocessor communication, and the facility to conduct autonomous processing per DDPN, is provided by the Node bus. Bus usage follows DEC Q22BUS protocol. It is important to note here that there is a "short-circuit" in the usage of the Node Bus. The Data Table Monitor allows the Vector Processor to receive DMA interrupts from the Main Processor without any usage of the Node Bus. This is a simple and effective means of keeping the logical control of the DDPN at real time speeds. System software for this design feature is provided in the Vector Processor. This software allows "tagging" of the data as being internally or externally generated.

5.2 The Data Driven Concept Of Information Management.

In the data-driven system there are two kinds of data. The first is called _Information_ because it types the data as having relevant importance to a computational process. The second is called _Redundant_ because it types the data as being of no value to a computational process. The importance of information to a computational process is expressed as time, summation, or delta critical. Time critical information is data which must reach the computational process at fixed delta times. Summation critical information is data that is summed to form a value that triggers an event. Delta critical information is data that must be significantly different from its previous value in order to trigger an event.

In the data-driven system a computational process' cycle of execution is based on its sensitivity to the generation of information data. To control the redundant data from causing needless cycling of the computational processes, a method has been developed that filters data both external and internal to the DDPN.

5.2.1 Transmission Of Information Data & Blockage Of Redundant Data.

There are two stages in filtering data for information data control. The Vector Processor filters data in a pre-comparison state where the data is not checked for information value and the Data Driver filters data in a post-comparison state where the information content of the data is known.

The primatives of the Vector Processor operate on a data packet coming from the System Bus or Main Processor filtering it before it is placed onto the Data Driver's input FIFO. To determine if the recipient node has need of the data in the data packet, the primatives use the address (pointer) in the data packet plus a fixed offset as the address of a vector table entry. If the entry is non-null (i.e. not all zeros) the vector table's entry -- a 32-bit address of the Data Driver data comparison primative, the data packet 12-bit address (pointer)/ 16-bit data pair, and 4-bit packet indicator -- are loaded onto the 32-bit wide FIFO buffer as two long words (Figure 8). This process is the same for a Main Processor DMA detection except that the 12-bit address field is set to all zeros and the packet indicator is different.

The Data Driver's data comparison primatives are dispatched by a tight-loop entry primative that uses the vector table data (the 32-bit address of the Data Driver data comparison primative) to jump to the appropriate comparison primative. The primative is specific for that particular data's machine representation data type and output destination. The data table record entry is updated (as prescribed by the data table record's entry indicator bits). If the data is informational data, the comparitor primative signals that the new data is to be placed on either the Main Processor input FIFO queue or the DDPN output FIFO queue for transmission to all nodes on the DDLAN (based on the data comparitor primative called). Control is then returned to the DDPN's entry point tight-loop primative. If the data is redundant data, the data table record entry is updated (as prescribed by the data table record's entry indicator bits) and control is returned to the tight-loop primative.

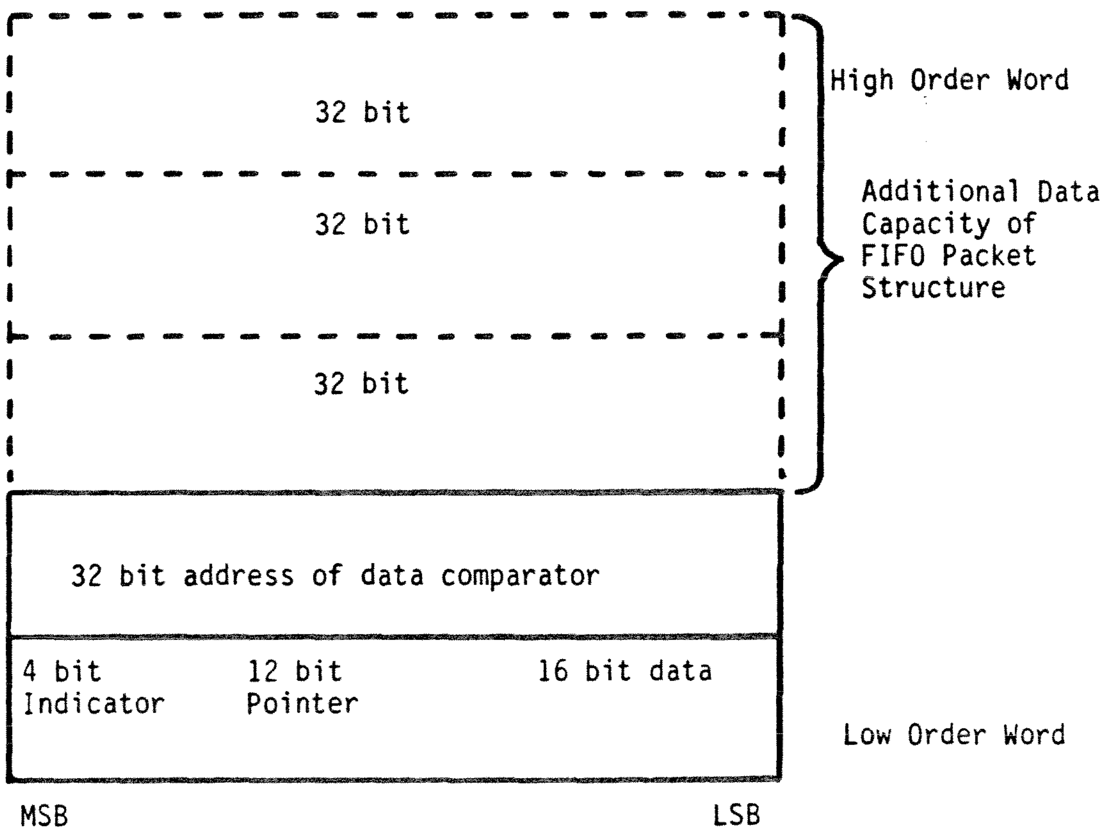


Figure 8. Data-Driver Input FIFO Structure.

A reduction in the number of primatives needed by the Data Driver is accomplished by using the 12-bit address (pointer) of the global data's data comparison primative as a qualifier for selection of the current data table entry. Selection of the appropriate entry is done through a subtable of data table entries based on the 12-bit pointer internal to the routine. Therefore the amount of primatives needed by the Data Driver for comparing data for the Main Processor's
computational processes will be based on the total amount of different machine data types in use by the application software's global variables in the Data Driven System.

5.2.2 Separation Of Process Logic And Computational Logic.

The difference between process logic and computational logic in conventional programming is bounded by a thin fluid definition. The following definition is used for distinguishing between these two logics in data-driven software.

> Process logic is programming logic which determines the execution of single function code modules. A single function code module has no external calls and branches only within its own local block.
>
> Example:

```
     begin PROCESS_LOGIC

          case CALLING_FUNCTIONS is

               when TEST => TESTING(TESTING_DATA);

               when MISSION => MISSION_START;

               when others => raise MISSION_ABORT;

          end case;

     end PROCESS_LOGIC;
```

> Computational logic is the programming logic that determines the method of data computation based on no other information other than the value of the data being computed. Computational logic can cause branches within single function code blocks.
>
> Example:

```
     begin COMPUTATIONAL_LOGIC

          for KOUNTER in 1 to 20 loop

               X := X + 1;
               Y := SQRT(X);

          end loop;

     end COMPUTATIONAL_LOGIC;
```

A Data Driver's data comparison primatives control the process logic of the Main Processor by only scheduling a process for execution when that process has information data. Control is exercised through updating the data table record entries through accepting data from the Data Driver input FIFO. The Main Processor never initiates a process on its own, but it has the primative operations to reschedule processes that it currently has. The Main Processor's function is to execute computational processes given it by the DDES without concern by the application code as to whether the process should be scheduled at that time or not (decision logic removal).

An analogy to this type of process control would be the nerve-terminal, ganglia, and cerebellum relationship in vertebrates. If the nerve-terminal receives a stimulus and the ganglion relates the stimulus to a condition know as "pain" to the vertibrate, then the
reaction of the muscles to contract is not controlled by a cerebral action but an a'priori <u>re</u>action of the ganglion. The relation in the DDPN would have the nerve-terminal as the Vector Driver, the ganglion as the Data Driver and the cerebellum as the Main Processor.

5.2.3 <u>Data Table Use And Format.</u>

Data-driven control of the Main Processor is accomplished through the use of the data table. As previously described, the two 32-bit long words are queued by the Vector Processor onto the Data Driver's input FIFO and are dequeued by the entry point tight-loop primative. From the 4-bit indicator the data is determined to be either internally or externally generated data. If the data is internally generated then the value of the data is not passed on the FIFO queue, but its address is passed through to the Data Driver When the data is external, as specified by the 4-bit indicator in the low order word placed on the FIFO queue, then the 16-bit portion(s) of the data is passed on the FIFO queue (Figure 9).

0	0	1	1	1 word internal data	from local DDPN
0	1	0	1	2 word internal data	
1	0	0	1	3 word internal data	
1	1	0	1	4 word internal data	
0	0	1	0	1 word external data	external to DDPN
0	1	0	0	2 word external data	
1	0	0	0	3 word external data	
1	1	0	0	4 word external data	

Figure 9. Decode Logic for DDP Input FIFO 4-Bit Indicator.

5.2.3.1 Format Of The Data Table.

The order of the data table in which the system global data entries occur is very sensitive due to the use of the relational position of a particular datum being an explicit offset to another particular datum. Even though the data table has variable machine representation data and variable length records, the relational position of the fixed length data is used to "walk" the table as described below. The data table entry format is in the order given. The 12-bit address (pointer) used by the Vector Processor is passed to the data comparison primative to differentiate between data table entries for data of the same machine representation.

1. Call Address -- The address of the beginning of the computational algorithm for the data in the Main Processor (32-bit).

2. Table Address -- The address of the data table data value. This points to the New Value entry (32-bit).

3. True/False -- The indicator that the data has become significant (1-bit).

4. Summation/Delta -- Indicates if the input data is summed to form the significant value and make True/False go True or whether the absolute difference of the Old Value and New Delta are to cause True/False to go True (1-bit).

5. Less than Bottom -- Lower window value trigger bit.

6. Less than Top -- Upper window value trigger bit.

7. Equal to Bottom -- Lower window value trigger bit.

8. Equal to Top -- Upper window value trigger bit.

9. Bottom Value -- The absolute value of the window's lower value.

10. Top Value -- The absolute value of the window's upper value.

11. Summation -- Contains the summed values of incremental data values. This entry is zeroed when the process is scheduled for execution by the DDES primatives (8 to 64-bit).

12. Delta from old value -- This entry tells the signed difference between the last data value compared and the new

value. This entry is set to zero when the process is scheduled for execution (8 to 64-bit).

13. Old Value -- The value prior to update (8 to 64-bits).

14. New Value -- The value after update. This is the location of the data for the Main Processor when it has become significant (8 to 64-bit).

15. Paired Values -- The Boolean List and Data Link Address are paired values in the data table. The values are fixed length to allow the DDES system primative TREE_WALK to access all data in the data table that is needed to schedule a process.

1. Boolean List -- tells what boolean operation is to be performed on the True/False bit on the following link address (3-bit). The list tells what operation as well as necessary data connectivity is needed when a computational process needs several data items becoming relevant at once.

    1. "0 0 0" : AND

    2. "0 0 1" : OR

    3. "0 1 0" : XOR

    4. "0 1 1" : NOT

    5. "1 1 1" : END OF LINK (this is the indicator of the last address to be compared in a data link list.)

2. Data Link Address -- The address of another data table entry containing a value used to determine if the information data for the process is ready for the process to run (32-bit).

5.2.3.2 Information Data Control -- The Process Lists.

Control of the process through information data has been used in the above sections as an axiom. The key for controlling process creation is the method by which the data table is used. The number of data comparitor primatives is less than the total number of unique data table entries. Therefore the comparitor primative must first ascertain what to do with the data. In the following discussion the concept of data being relevant can be tuned by judicious choice of the "window"
boundries Top and Bottom. These values keep the redundant data at the "background noise" level causing the DDPN to be less sensitive to the generation of data.

When the input data enters the comparison primative the primative accesses the data table to find the True/False flag to see if any previous access has made the data relevant. If True/False is true the data linked list is "walked" for that entry (an explanation of "walking" follows in the next paragraph). If True/False is false the Summation/Delta flag is checked to determine the computation on the incoming data. Knowing this, the four boolean trigger bits (Less than Bottom to Equal to Top inclusive) are checked to determine the boolean tests to be made on the computation (Figure 10) in relation to the Top Value and Bottom Value.

.LT. B	.LT. T	.EQ. B	.EQ. T	DATA IS RELEVANT WHEN IT IS:
1	0	0	0	.LT. BOTTOM VALUE
0	1	0	0	.LT. TOP VALUE
0	0	1	0	.EQ. BOTTOM VALUE
0	0	0	1	.EQ. TOP VALUE
1	0	1	0	.LE. BOTTOM VALUE
0	1	0	1	.LE. TOP VALUE
0	1	1	0	.GT. BOTTOM VALUE
1	0	0	1	.GT. TOP VALUE
1	1	1	0	.GE. BOTTOM VALUE
0	1	1	1	.GE. TOP VALUE

Figure 10. Decode Logic for the Data Table Boolean Trigger Bits.

If the Summation/Delta flag is true then the data is added to the Summation field value and the boolean operation is done. If the Summation/Delta flag is false the data is added to the Old Value to get the New Value and the boolean test is now done using New Value with the Top and Bottom values in the table. Note that Top and Bottom can be either absolute values or delta values while keeping the parameterization for the data comparison primatives generic. For subtraction, the data is negative. Correct sign is a responsibility of the application program generating the data. If the test is false then Summation or Delta is modified and True/False is set to False. If the test is true then True/False is set to true and the Boolean List from the Data Linked List is now "walked".

"Walking" the Data Linked List is the responsibility of the DDES primative TREE_WALK which uses the Boolean List (BOOL) and the Data Link Address (DLA) to use fixed offset addressing to access other data table entries that effect the scheduling of a process. A prioritization of DLAs is attained by ordering the BOOL/DLA pairs in the priority order in which the causality data for a process scheduling event must be checked and scheduled. Using the first DLA, in the BOOL/DLA list, vector to the data table entry and operate on the True/False flag with the BOOL value. This process continues until a True value is attained by operation of all the vectored to True/False and BOOL values. If true is the result, and knowing the priority of the BOOL/DLA pairs, DATA_COMPARE, through TREE_WALK passes all the Call Addresses in the data table entries that are referenced by the BOOL/DLA pairs to the CREATE_PROCESS primative for dispatch to the Main Processor. If the result of the TREE_WALK is false, then the True/False flags in all the data table entries retain their state and the next incoming data is processed.

The above discussion of the Data Link List describes only half of the power of the data table. The other half is the table's ability to control the transmission of data onto the System Bus. If the primative called was for transmission of the data from the node, then then data comparison primative would go through the same steps described above with a final call by DATA_COMPARE, if indeed there was a BOOL/DLA pair (the Boolean List may have had "111" as the first entry), to GET_BUFFER and WRITE_BUFFER primatives. If the Boolean List did have "111" as the first entry then the data comparison primative would have called GET_BUFFER and WRITE_BUFFER.

5.3 Control Systems Of A Data Driven Node.

In the previous portions of this paper, control of the data table has been explicitly and implicitly accomplished solely through system primatives that are independent of the application software running in the Main Processor. In the next three sections the specific primatives that make-up the system software for each processor in the DDPN are assembled. The functional descriptions are given to indicate the relationships that exist between the primatives.

5.3.1 Vector Processor Operating System Primatives.

The choice of primatives for the VPOS is limited to the most basic functions for processing data through the vector table. The primatives listed are used in the above discussion in the manner described in the text following each primative.

```
GET_BUFFER(FREE_BUFFER_NUMBER : out BUFFER_NUMBER);
```

Return an input buffer for READ_BUFFER when an interrupt occurs which indicates that a data arrival event has occured.

```
READ_BUFFER(BUFFER : out BITS_36; BUFFER_TYPE : out BITS_2);
```

Return the buffer contents and the internal/external indicator. The POINTER and COMPDATA contents are then extracted from BUFFER.

```
TABLE_OFFSET(POINTER              : in  BITS_12;
             DRIVE_PRIMATIVE_ADDR : out BITS_32);
```

Uses the 12-bit pointer plus offset to return the data comparison primative address

```
PUT_BUFFER(INDICATOR            : in BITS_4;
           POINTER              : in BITS_12;
           COMPDATA             : in BITS_16;
           DRIVE_PRIMATIVE_ADDR : in BITS_32;
           STATUS               : out BITS_2);
```

Put data onto the Data Driver's FIFO queue. The status returned is an indication of how full the FIFO is becoming.

```
SYSTEM_EXCEPTION(EXCEPTION : in EXCEPTION_TYPE);
```

There are three exceptions defined for the VPOS; BAD_ADDRESS, OUT_BUFFER_FULL, IN_BUFFER_ERROR

RETRANSMIT_BUFFER(NODE_ID : in NODE_TYPE);

Sends message out onto the System Bus for retransmission of a data packet.

5.3.2 VPOS States And Transitions.

The following state and transition diagram shows the primative states that the Vector Processor transitions through. This diagram gives the states that exist at a process level in the system software. As data comes into the Vector Processor the VPOS is taken from the NULL state to the INTERRUPT state in which the data packet is collected. Next the VPOS enters the RUNNING state until the data packet, table look-up, and FIFO operations are complete. If there is no pending interrupt, the VPOS goes back to the NULL state. If there is a pending interrupt the VPOS immediately transitions through the NULL state to the INTERRUPT state (Figure 11).

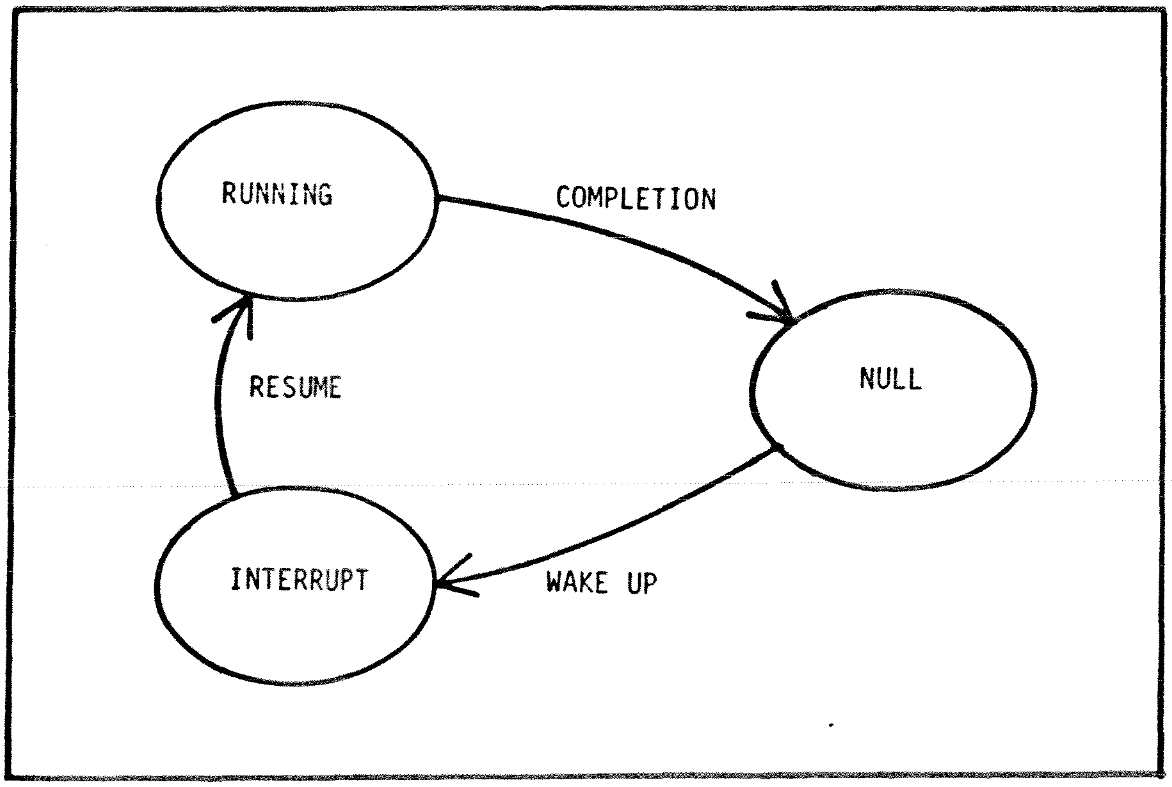


Figure 11. VPOS States & Transitions.

5.3.3 Data Driver Executive System Primatives.

These primatives provide the essential methods for controlling the processes in the Data Driver. These primatives are used in controlling the data table for scheduling processes in the Main Processor. The primatives listed are used in the above description in the manner described in the text following each primative.

```
READ_FLAGS(FLAGS : out FLAG_WORD);
```

To read the other CPU's status flags.

```
SET_INTERRUPT(MASK   : in MASK_WORD;
              LEVEL  : in INTERRUPT_LEVEL;
              STATUS : out BITS_4);
```

To set the eight maskable crosslinked interrupts.

```
CREATE_PROCESS(PCB_DATA : in PCB_RECORD);
```

When TREE_WALK results in a True value, this primative is called to create a process control block for the Main Processor.

```
DATA_COMPARE(INDICATOR             : in BITS_4;
             POINTER               : in BITS_12;
             COMPDATA              : in BITS_16;
             DRIVE_PRIMATIVE_ADDR  : in BITS_32);
```

This primative vectors to the appropriate data comparison algorithm which uses POINTER to distinguish which data type specific routine is called. This primative contains its own SCHED_PROCS.

```
GET_PID(PID : out INTEGER);
```

Return the process ID for a process in the process table.

```
GET_SCHED(PID : in  INTEGER; PRIORITY : out INTEGER);
```

Return the scheduling priority of a process in the process table.

```
ABORT(PROCESSOR : in PROCESSOR_ID; PID : in INTEGER);
```

Stop a process in the Data Driver or Main Processor. This is used when aborting a process is desired over normal process termination.

```
SYSTEM_EXCEPTION(EXCEPTION : in EXCEPTION_TYPE);
```

The exceptions are PROCESS_ABORT, SYSTEM_ABORT, NODE_ABORT.

```
PORT_COUNT(PORT_ID : in INTEGER; QUEUE : out INTEGER);
```

Return the wait count for a resource at a port.

```
PORT_RECEIVE(PORT_ID : in INTEGER; QUEUE_TOP : out INTEGER);
```

Take the first message waiting at the port. Decrements QUEUE_TOP.

```
PORT_RESET(PORT_ID : in INTEGER);
```

Reset the port, signal all waiting procedures. This causes a cascade of procedures internal to the DDES. This is used to break-out of an INCREMENTAL_READY state.

```
PORT_SEND(PORT_ID, PORT_MESSAGE : in INTEGER);
```

Send a message to a port. Increments QUEUE.

```
PORT_CLEAR(PORT_ID : in INTEGER);
```

Like PORT_RESET, but does not signal the waiting processes. This is used with PORT_RECEIVE to get a priority message.

```
SCHED_PROCS(PID, PRIORITY : in INTEGER);
```

Schedule a process in the process table with a priority. The pocess control block is DMA written to the Main Processor´s process queue.

```
SCHED_LIST_PROCS( PID      : in PID_LIST;
                  PRIORITY : in PRIORITY_LIST);
```

Schedule a list of processes in the process table with their priorities. The process control blocks are DMA written to the Main Processor´s process queue. This primative is used with TREE_WALK when there is a list of processes to be scheduled.

```
TREE_WALK(RESULT : out BITS_1);
```

Uses BOOL_DLA to walk the Data Link list maintained in the data table.

```
BOOL_DLA( out BOOLEAN_LIST : BITS_3;
          out DRIVE_PRIMATIVE_ADDR : BITS_32);
```

Return the Data Link information in the data table.

```
RESUME_PROC(PID   in INTEGER);
```

Resume a DDES suspended process.

```
PROC_SUSPEND(PID : in INTEGER);
```

Suspend a DDES process.

```
SEMA_COUNT(SEMA_NAME : in SEMAPHORE;
           COUNT     : out INTEGER);
```

Return the count associated with a semaphore.

```
SEMA_CREATE(SEMA_NAME : in SEMAPHORE);
```

Create a semaphore.

```
SEMA_DELETE(SEMA_NAME : in SEMAPHORE);
```

Delete a semaphore.

```
SIGNAL(SEMA_NAME : in SEMAPHORE);
```

Signal a semaphore to unblock a process.

```
PROC_SLEEP(PID, DELTA_TIME : in INTEGER);
```

Put a process to sleep for so many "ticks" of the processor clock.

```
SEMA_RESET(SEMA_NAME : in SEMAPHORE);
```

Reset a semaphore count to zero.

```
WAIT(SEMA_NAME : in SEMAPHORE);
```

Wait on a semaphore. Process is blocked.

```
FREE_BUFFER(BUFFER_NUMBER : in BUFFER_NUMBER);
```

Free a buffer in the "pool."

```
GET_BUFFER(FREE_BUFFER_NUMBER : out BUFFER_NUMBER);
```

Return an input buffer for READ_BUFFER when an interrupt occurs which indicates that a data arrival event has occured.

```
READ_BUFFER(INDICATOR            : out BITS_4;
            POINTER              : out BITS_12;
            COMPDATA             : out BITS_16;
            DRIVE_PRIMATIVE_ADDR : out BITS_32);
```

Reads in the FIFO data for DATA_COMPARE.

```
PUT_BUFFER(BUFFER      : in BITS_36;
           BUFFER_TYPE : in BITS_2;
           STATUS      : out INTEGER);
```

Writes the buffer contents using the System Bus or Main Processor based on the BUFFER_TYPE. The POINTER and COMPDATA contents are included in BUFFER.

5.3.4 DDES States And Transitions.

The following state and transition diagram shows the primative states that the Data Driver transitions through. This diagram gives the states that exist at a process level in the system software. As data comes into the Data Driver, the DDES is in the BLOCKED state. A reschedule transition occurs to the READY state where DATA_COMPARE is dispatched to RUNNING. As the data table is accessed the DDES process goes through BLOCKED, INCREMENTAL_READY, and READY states. After the data table has been updated, processes scheduled for the Main Processor or System Bus transmission completed, the DDES transitions through interrupt to the BLOCKED state, waiting for a resume transition which is signaled by input to the DDES input FIFO (Figure 12).

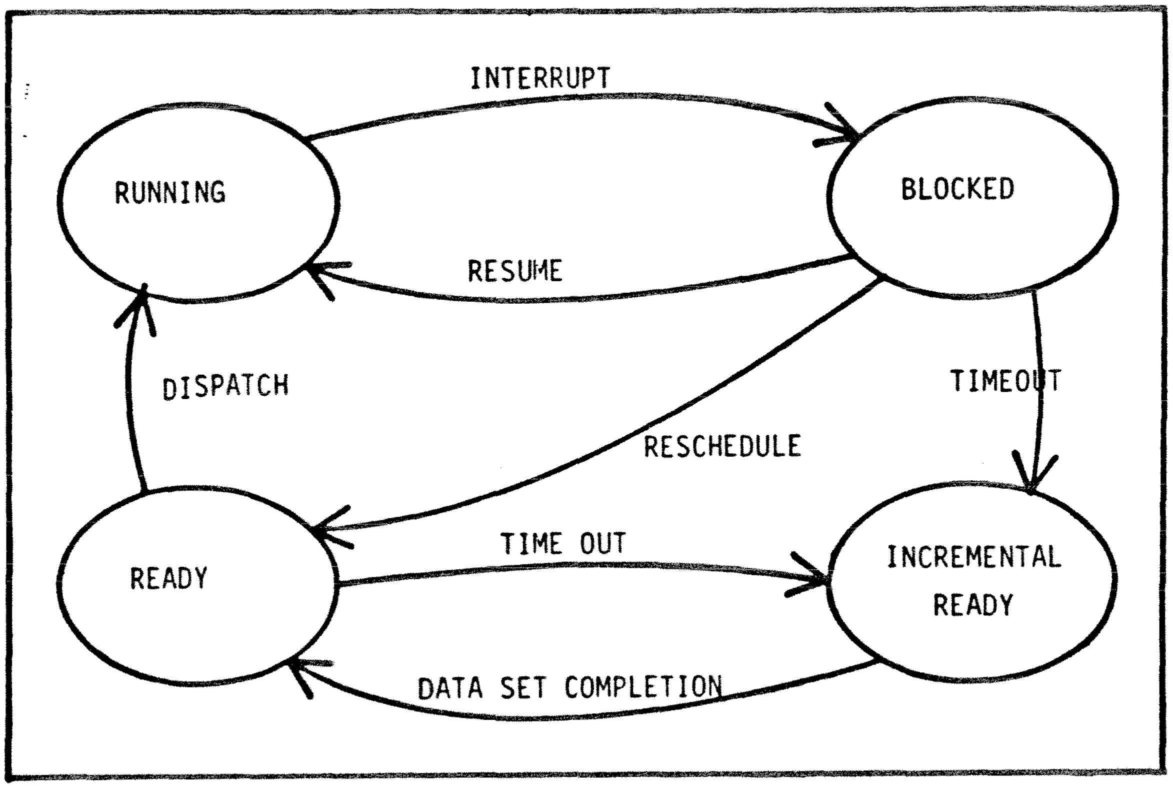


Figure 12. DDES States & Transitions.

5.3.5 Main Processor Operating System Primatives.

These primatives provide the essential methods for controlling the processes in the Main Processor. As control of process selection comes from the DDES, the MPOS is only concerned in maintaining the current process. Interrupts only occur in the Main Processor when the process list is updated. The primatives listed are used in the above discussion in the manner described in the text following each primative.

```
DEVICE_OPEN(DEVICE        : in DEVICE_TYPE;
            DEVICE_NUMBER : in INTEGER);
```

Open a channel to a physical device.

```
DEVICE_CLOSE(DEVICE_NUMBER : in INTEGER);
```

Close a channel to a physical device.

```
DEVICE_CONTROL(DEVICE  : in DEVICE_TYPE;
               COMMAND : in DEVICE_COMMAND;
               STATUS  : out DEVICE_TYPE_STATUS);
```

Generic control of a device.

```
GET_PID(PID : out INTEGER);
```

Return the process ID for a process in the process table.

```
GET_SCHED(PID : in  INTEGER; PRIORITY : out INTEGER);
```

Return the scheduling priority of a process in the process table.

```
ABORT(PID : in INTEGER);
```

Stop a process in the Main Processor. This is used when aborting a process is desired over normal process termination.

```
SYSTEM_EXCEPTION(EXCEPTION : in EXCEPTION_TYPE);
```

The exceptions are PROCESS_ABORT, SYSTEM_ABORT, NODE_ABORT.

```
PORT_CREATE(PORT_ID : in INTEGER);
```

Allows MPOS to provide process communication ports for the processes´ application code.

```
PORT_DELETE(PORT_ID : in INTEGER);
```

Allows MPOS to get rid of unneeded ports.

```
PORT_COUNT(PORT_ID : in INTEGER; QUEUE : out INTEGER);
```

Return the wait count for a resource at a port.

```
PORT_RECEIVE(PORT_ID : in INTEGER; QUEUE_TOP : out INTEGER);
```

Take the first message waiting at the port. Decrements QUEUE_TOP.

```
PORT_RESET(PORT_ID : in INTEGER);
```

Reset the port, signal all waiting procedures. This causes a cascade of procedures internal to the MPOS. This used to break-out of a BLOCKED MPOS state.

```
PORT_SEND(PORT_ID, PORT_MESSAGE : in INTEGER);
```

Send a message to a port. Increments QUEUE.

```
PORT_CLEAR(PORT_ID : in INTEGER);
```

Like PORT_RESET, but does not signal the waiting processes. This is used to get a priority message.

```
RESCHED_PROCS(PID, PRIORITY : in INTEGER);
```

Reschedule a process in the process table with a priority. The process control block is modifiable after a process list update interrupt occurs from the DDES.

```
RESCHED_LIST_PROCS( PID      : in PID_LIST;
                    PRIORITY : in PRIORITY_LIST);
```

Reschedule a list of processes in the process table with their priorities.

```
RESUME_PROC(PID : in INTEGER);
```

Resume a MPOS suspended process.

```
PROC_SUSPEND(PID : in INTEGER);
```

Suspend a MPOS process.

```
SEMA_COUNT(SEMA_NAME : in SEMAPHORE;
           COUNT     : out INTEGER);
```

Return the count associated with a semaphore.

```
SEMA_CREATE(SEMA_NAME : in SEMAPHORE);
```

Create a semaphore.

```
SEMA_DELETE(SEMA_NAME : in SEMAPHORE);
```

Delete a semaphore.

```
SIGNAL(SEMA_NAME : in SEMAPHORE);
```

Signal a semaphore to unblock a process.

PROC_SLEEP(PID, DELTA_TIME : in INTEGER);

Put a process to sleep for so many "ticks" of the processor clock.

SEMA_RESET(SEMA_NAME : in SEMAPHORE);

Reset a semaphore count to zero.

WAIT(SEMA_NAME : in SEMAPHORE);

Wait on a semaphore. Process is blocked.

FREE_BUFFER(BUFFER : in BUFFER_NUMBER);

Free a buffer in the "pool."

GET_BUFFER(FREE_BUFFER_NUMBER : out BUFFER_NUMBER);

Return an input buffer for READ_BUFFER when an interrupt occurs which indicates that a data arrival event has occurred.

READ_BUFFER(BUFFER : out BUFFER_CLASS);

Read a buffer from the class of buffers available for application programs.

PUT_BUFFER(BUFFER : in BUFFER_CLASS);

Write a buffer from the class of buffers available for application programs.

5.3.6 <u>MPOS States And Transitions.</u>

The following state and transition diagram shows the primative states that the Main Processor transitions through. This diagram gives the states that exist at a process level in the system software. When an interrupt is received from the DDES, the MPOS is in the RUNNING or READY states. While the DMA is occurring the MPOS transitions to the BLOCKED state if it is currently RUNNING, or if in READY,it maintains its READY state. When DMA posting is finished by the DDES, the MPOS is signaled if blocked to re-examine its ready-list and dispatch the highest priority process to the RUNNING state (Figure 13).

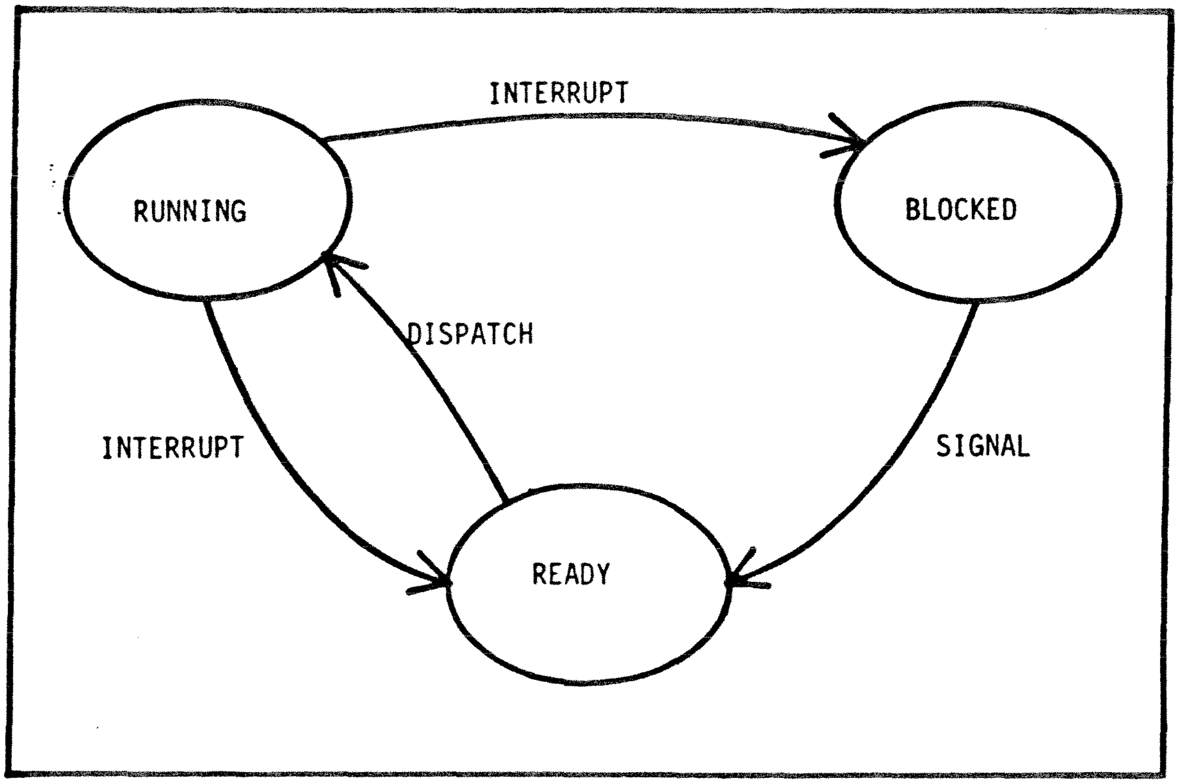


Figure 13. MPOS States & Transitions.

5.4 <u>The Support Environment Requirements For Programming Data Driven Nodes.</u>

The DDPN will need a new type of compiler and linker to produce code for the data-driven system. In particular the code required must come from compilers that are specifically multiprocessor system compilers and linkers. Synthetic execution of compiled and linked programs is necessary in order to test the data-driven system prior to hosting on the multiprocessor system. In this way a single CPU host can optimize the application by "fine tuning" the code development process with a set of predetermined execution parameters that can demonstrate how close the code design meets the requirements of the system. Such a topic is outside the scope of this paper and is mentioned here only to include a development perspective to the DDOS description given in this paper.

6.0 Conclusions & Recommendations

From our investigation into the concept of data-driven architectures and software, the premise of information data controling computational processes is found to be feasible. There are minor technical problems in software development to overcome in using a data-driven architecture due to the current assemblers and high order language compilers available being designed for single CPU systems. Current compilers/assemblers can be used in data-driven software development by manual resource resolution manipulation. We feel that the anticipated benefits of the system far exceed any minor problems in software development that can be solved by compiler pragmas for resource allocation.

Current "form-fit" factors allow the data-driven hardware to be inserted into existing systems that have provision for networking of the airframe's computational resources. The choice of fiber-optic transmission lines remove the close proximity requirements of current networked processors thus allowing for increased usage of airframe electronics storage space.

The utility of a data-driven system is best demonstrated at a flight simulator level prior to testing with an airframe's networked computational resources. The simulation level has more computational algorithms than mission software due to the simulator having to create the environment for the aircraft rather than fly through it. Thus with more demand on the data-driven system, the system's concept has a practical demonstration of its abilities.

Biblography

1. Alexandridis, N.A., Microprocessor System Design Concepts, Computer Science Press, Rockville, Maryland, 1984.

2. Comer, D., Operating System Design: The XINU Approach, Prentice-Hall 1984.

3. Glass, R.L., et al., Real-Time Software, Prentice-Hall 1983.

4. Lillevek, S.L., and Easterday, J.L., "A multiprocessor with replicated shared memory," Proceedings of the National Computer Conference, Seattle, WA, 1983.

5. McDonald, J.E., "An Architecture for Event-Driven Real-Time Distributed Computer Systems," Department of Computer Science, Wright State University, 1983.

6. Soh, S.E., Data-Driven Computing Systems: A Survey., M.S. Thesis, Department of Computer Science, Wright State University, 1981.

7. Swan, R.J., et al., "The Implementation of the Cm* Multi-Microprocessor," Proceedings of the National Computer Conference, Dallas, TX, 1977.

8. Trealeaven, P.C., Brownbridge, D.R., and Hopkins,R.P., "Data-Driven and Demand-Driven Computer Architecture," Computing Surveys, Vol. 14, No. 1, March 1982.

9. Welch, H.O. and Moquin, W.A., "An Analysis of a Multicache Shared Memory Ring Interconnection," Proceedings of the IEEE Real-Time Systems Symposium, 1982.

10. Weitzman, C., Distributed Micro/Minicomputer Systems., Prentice-Hall 1980.

Biographical Sketches

James Edward McDonald was born in Troy, Ohio on August 18, 1950. He graduated from the University of Cincinnati in 1973 with a B.S.E.E. degree. He is currently finishing thesis work toward a M.S. Computer Science degree at Wright State University, Dayton, Ohio.

In June 1973 he began an engineering career at the Avionics Laboratory, Wright Patterson Air Force Base. Early career duties involved military laser measurement devices. Current duties include the development of system avionics support systems that involve real-time computer architectures. He has published two papers in this and related fields and has authored one technical report with another in progress. Mr. McDonald has one invention disclosure and is a registered professional engineer and a member of IEEE.

Duane O. Hague was born in Akron, Ohio on June 22, 1945. He graduated from the University of Akron in 1973 with a B.S.E.E degree. After graduation, Mr. Hague began a career at the Avionics Laboratory, Wright Patterson Air Force Base.

Early career achievements include hardware developments concerning digital timing and recording for inertial navigation systems and airborne computers. current career objectives include avioncs support system design for VHSIC based systems. Mr. Hague has published several technical reports in related career technical fields.

Richard Michael Wallace was born in Dayton, Ohio on December 26, 1957. He graduated from the University of Idaho in 1980 with a B.S. degree. He is currently finishing course work toward a M.S. Computer Science degree at the University of Dayton, Dayton, Ohio.

After working for the University of Dayton Research Institute, Dayton, Ohio and System Architects Inc., Dayton, Ohio in the airborne ECM & ECCM modeling areas, Mr. Wallace began his computer scientist career at the Avionics Laboratory in March 1983. His early career duties included the Air Force's test management for the Tri-Service testing of the Army's Ada Language System (ALS) and operating system software design review for the Integrated Communications, Navigation, Identification, Avionics (ICNIA) Program. His current duty is software development manager for the VHSIC Hardware Design Language (VHDL).