Report No. 2930 Bolt Beranek and Newman Inc.

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

January 1975

Sponsored by:

Advanced Research Projects Agency
ARPA Order No. 2351
Contract No. F08606-73-C-0027

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PREFACE

"Pluribus Document 2: System Handbook" is one of a set of nine which, taken together, provide complete documentation of the Pluribus line of computer systems. In the present document, Part 1, entitled "Guide to Documentation," gives an overview of the entire set. Part 2, "System Description," contains an extensive discussion of the Pluribus line and the ways in which it can be used. This system description is the primary text for anyone seeking familiarity with the Pluribus, although, of course, there are many details which can only be found elsewhere in the set. Part 3 is a glossary of specialized Pluribus terms used throughout the set. Part 4 is an index to the present document. Part 5 contains reprints of several papers relevant to the Pluribus.

Of the five parts of "Pluribus Document 2," parts 1, 2, 3, and 5 are presently included here; part 4 is in production and will be added when it becomes ready.

TABLE OF CONTENTS

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PART 1: GUIDE TO DOCUMENTATION

Update History:

Originally written - February 1975

The Pluribus line of computer systems is documented in a series of nine volumes entitled as follows:

"Pluribus Document 1: Overview," BBN Report No. 2999
"Pluribus Document 2: System Handbook," BBN Report No. 2930
"Pluribus Document 3: Configurator," BBN Report No. 3000
"Pluribus Document 4: Basic Software," BBN Report No. 3001
"Pluribus Document 5: Advanced Software," BBN Report No. 2391
"Pluribus Document 6: Functional Specifications," BBN Report No. 3002
"Pluribus Document 7: Construction," BBN Report No. 3003
"Pluribus Document 8: Card Testing," BBN Report No. 3004
"Pluribus Document 9: System Integration," BBN Report No. 3005

The set of documents taken as a whole is intended to cover all aspects of the Pluribus; e.g., the decision to use a Pluribus, the design of systems involving the Pluribus, programming the Pluribus, actually fabricating the Pluribus hardware, and maintaining Pluribus systems. On the other hand, the set of documents is organized so that any one aspect of Pluribus endeavor (e.g., Pluribus manufacture) should be documented with a subset of the documents; thus, not everyone need carry all documents with him at all times--only those he needs.

The chart on the following page suggests which Pluribus documents will be useful for which areas of endeavor and for what types of people.

FOR (AREA OF ENDEAVOR):	SEE DOCUMENT NUMBER:
Designing an application system based on the Pluribus	1, 2, 3, 6
Building a Pluribus Computer	2, 4, 6, 7, 8, 9
Using a Pluribus Computer	2, 4, 5, 6
FOR (TYPE OF PERSON):	
One considering buying a Pluribus	1, 2
A Pluribus factory worker	7, 8, 9
A Pluribus hardware maintainer	2, 4, 6, 8, 9
A Pluribus programmer	2, 4, 5, 6

Table 1: Guide to Pluribus Documents

The documents have a loose-leaf format to facilitate updating.

Of the nine, documents 1 through 6 will be available in reasonably large quantities. Documents 7, 8 and 9 contain much detail of little general interest (e.g., wire lists, assembly drawings) and are extremely cumbersome to produce; therefore, their availability from BBN will be quite limited, although they will be submitted to the National Technical Information Service to allow general access.

In the following paragraphs, we discuss each of the nine documents in turn, presenting the contents of each and discussing its expected use.

Document 1: Overview. This document is meant to provide a quick summary of the Pluribus's capabilities, possible applications, and architecture, and is the first document one should read to determine if he is at all interested in using a Pluribus.

Document 2: System Handbook. This document is the primary text for one seeking familiarity with the Pluribus. The fundamental ideas of the Pluribus are introduced and then discussed in detail, including the structure of the hardware and guidance on how we think the hardware should be configured and programmed. In particular, after a brief general description of the Pluribus system structure, there are discussions of the processor structure and of the addressing structure for the system, an outline of how programs might be written to use the Pluribus structure effectively, a discussion of Pluribus device handling and I/O handling, a discussion of the structure of the Pluribus busses and how they are coupled together, summaries of the

various devices which can be connected to the Pluribus, and a discussion of the Pluribus reliability mechanisms. While this document might best be thought of as a programmer's reference manual for the Pluribus, or alternatively, as the reference manual for Pluribus systems analysts, we think that everyone associated with any phase of Pluribus development and use will be likely to want it, with the possible exception of those concerned with only very local aspects of Pluribus construction.

This document is enhanced by the inclusion of a glossary, a guide to other documentation (which you are reading), an index, and some reprints of relevant papers written during the Pluribus development process which may give the reader greater insight into the use and structure of the Pluribus.

Document 3: Configurator. This document lists the various components that make up Pluribus systems (e.g., memories, processors, busses) and gives rules for configuring Pluribus systems. These rules are of two forms: rules of the first form are concerned with performance limitations; rules of the second form are concerned with physical limitations. An example rule of the first form tells how effectively multiple processors on a bus can share a memory on the same bus as a function of processor speed and memory speed. An example rule of the second form says that if more than some number of cards are to be used on a bus, then a bus extender will be needed. Of course, in some areas these two forms of rules are not independent; for instance, adding a bus extender may slow down the bus.

This document will be used primarily by the systems analyst or system architect for a computer system using the Pluribus. Further, it will be necessary in order to price Pluribus systems accurately, since only careful configuration will list all the system components actually needed.

Document 4: Basic Software. This document presents only enough about the Pluribus software to enable the reader to program in basic machine language for the Pluribus. The Pluribus instruction set is presented, the several different Pluribus assembly languages are introduced, and there is a discussion of the basic debugging package which allows Pluribus memory locations and machine state information to be inspected and changed.

Every Pluribus programmer will need to read this document as this is the software he will need to do "hands on" debugging of his program. Additionally, those building and maintaining Pluribus hardware systems will need to read this document because it describes the software they will need to operate hardware diagnostic programs.

Document 5: Advanced Software. This document describes software beyond that needed just to debug programs and operate hardware diagnostics. The software available for the Lockheed SUE, the processor used in the Pluribus, is listed. Detailed descriptions and operating procedures are given for the two cross-assemblers available to assemble programs for the Pluribus.

The somewhat unstructured "package" that has been developed to permit reliable operation of the Pluribus is also discussed.

Every Pluribus programmer, whether he is writing application programs, utility programs, or diagnostics, will need to refer to this document.

Document 6: Functional Specifications. This document provides the physical characteristics, operating characterisitcs, and necessary programming details for every Pluribus card. One way to think of this document is as an extension to Document 2, giving greater detail on specific devices.

Document 7: Construction. This document provides the information necessary to build the components of Pluribus systems. For every Pluribus card, the following are included: parts list, wire list, art work, assembly drawing, and assembly procedure. For every mechanical part and cable used in a Pluribus, this document includes the following: parts list, assembly drawing, and assembly procedure. In addition, this document contains a section which includes detailed instructions for any modifications and option selections for Pluribus cards.

Document 8: Card Testing. This document gives instruction for testing every card that can be used in a Pluribus system. For Pluribus cards obtained from Lockheed, the Lockheed maintenance bulletin and diagnostic procedure are provided. For every card specially designed and constructed for the Pluribus,

this document includes the following: schematic diagrams, logic description, wire lists, test program, and test procedure.

This document is necessary for anyone debugging cards, either after initial construction in the Pluribus factory or after failure in the field.

Document 9: System Integration. This document describes how the components of a Pluribus system are assembled into a complete hardware system; e.g., how chassis mount in racks, how cards mount in chassis, and how to test the whole thing once it is together. Included in the document are an overview of the hardware system assembly process and the hardware system assembly procedure; option selection information; system test programs; and finally, system quality control and acceptance porcedures for newly constructed systems.

This document is necessary for anyone debugging Pluribus systems, either after initial construction in the factory or after failure in the field.

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PART 2: SYSTEM DESCRIPTION

SYSTEM DESCRIPTION

Update History:

Originally written by C. R. Morgan
and G. Falk, December 1974

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION

The Pluribus* system is a general-purpose multiprocessor computer suitable for applications ranging from those normally identified with minicomputers to those typically associated with larger machines. Pluribus hardware has been designed so as to provide a suitable basis for the development of ultra-reliable hardware/software systems.

Pluribus systems contain an arbitrary number of identical processors each of which has access both to its own private memory and to a common memory accessible by all processors. I/O devices which are part of the system can be controlled by any processor. The number of processors, size of common memory, and amount of I/O gear on a Pluribus system can be quite large.

The Pluribus system achieves modularity and reliability by making all the processors equivalent. Any processor can perform any system task or control any device. Since each subsystem of the Pluribus system (processor, memory, and I/O) is expandable, systems can easily be configured to meet the throughput requirements of a particular job. The scheme for interconnecting system components is also modular; hence, interconnection costs vary smoothly with system size.

The Pluribus system was originally developed to serve as a modular reliable packet-switching node for the ARPA Network [1]. A node consisting of a 13-processor system is currently operational. The Pluribus approach is appropriate, however, for many other applications where reliability, modularity, or large logical computing power is required.

*Trademark of Bolt Beranek and Newman Inc. (BBN)

This handbook will describe the structure and operation of the Pluribus system. It will emphasize utilization of the Pluribus architecture in the manner for which it was originally designed, although additional possibilities will become clear as the discussion progresses. The handbook is oriented both to the programmer who will use it as a basic reference document and to the system designer who will have to determine if the Pluribus is appropriate for his particular application. Section 2 presents an overview of the Pluribus architecture. Section 3 contains a brief description of the processor. Sections 4, 5, and 6 present the basic information concerning the Pluribus system; addressing, programming, and device handling. Section 7 discusses reliability machanisms in the Pluribus system, both hardware and software, in detail. Sections 8 and 9 discuss the Infibusses and bus couplers. Finally, Section 10 describes many device dependent features and bits and will be useful most likely for reference purposes only.

The Pluribus is constructed out of components manufactured both by BBN and by Lockheed Electronics Company, Inc. (LEC). The BBN-produced hardware is described in detail in this document. LEC hardware from the SUE minicomputer line is discussed only in sufficient detail to make the description of the Pluribus coherent. More complete information on the SUE components can be found in the Lockheed product literature [2,3,4,5].

2. PLURIBUS SYSTEM STRUCTURE

A Pluribus system consists of a number of components (processors, memory modules, and I/O devices), a number of busses over which these components communicate, and a number of bus couplers which provide the mechanism for interconnecting the individual busses. Within this framework a wide variety of systems can be configured ranging from small single bus systems to large multi-bus systems with tens of processors, up to 1024K bytes of main memory, and a large assortment of I/O gear. The subsequent discussion will focus on a medium sized Pluribus configuration. Very small and very large systems both involve additional considerations not discussed in detail here.

The basic skeletal unit of the Pluribus system is the SUE Infibus* onto which all BBN and LEC devices are connected. The Infibus not only serves as a chassis into which device cards are plugged but also provides a means for communication among all attached devices. In general, a single Infibus can have an assortment of cards on it: processors, memories, or I/O devices. However, only one device can be in control of the Infibus at any given instant. An Infibus arbiter (Bus Control Unit Card or BCU) which must be present on the Infibus guarantees that this is the case. The total number of components which can be plugged onto a single Infibus is dependent on the number of slots available for cards and the type of power supply used (see section 8.). Throughout the remainder of this document the word "bus" will be used as a shorthand for Infibus.

---

*Trademark of Lockheed Electronics Company, Inc.

A small Pluribus system (even a single processor system) can be built using only a single bus. For many applications, however, the bandwidth capability and/or card capacity of a single bus is exceeded and a multi-bus structure is required. In addition, applications which take advantage of the full capabilities of the Pluribus hardware for bandwidth and reliability will require multi-bus configurations.

With more than one bus, the question becomes how to assign processors, memory, and I/O devices to the individual busses and how to connect them together. A typical configuration is illustrated in Figure 1. Lines between busses represent bus couplers. Typically, busses in a Pluribus system are configured as one of three types: processor busses, memory busses, or I/O busses. Processor busses support processors and private memory associated with each of the processors. Up to four processors (numbered 0-3) can logically be put on a single bus although contention for the bus is likely to reduce the effective processor bandwidth. In the ARPA Network application, for example, four processors with contention produce the same computational capacity as would three processors if there were no interference among the processors (i.e., if the processors were actually independent). Although the contention is application-dependent, Pluribus systems will generally be configured with one, two, or three processors per processor bus. Two processors are indicated for the system illustrated in Figure 1. The other components normally residing on the processor bus are the processor private memories. These memories will contain the "hot code" (i.e., those routines most frequently referenced) so as to reduce competition for the pool of shared (common) memory, and other code which is important to protect by removing it from shared memory. One useful technique

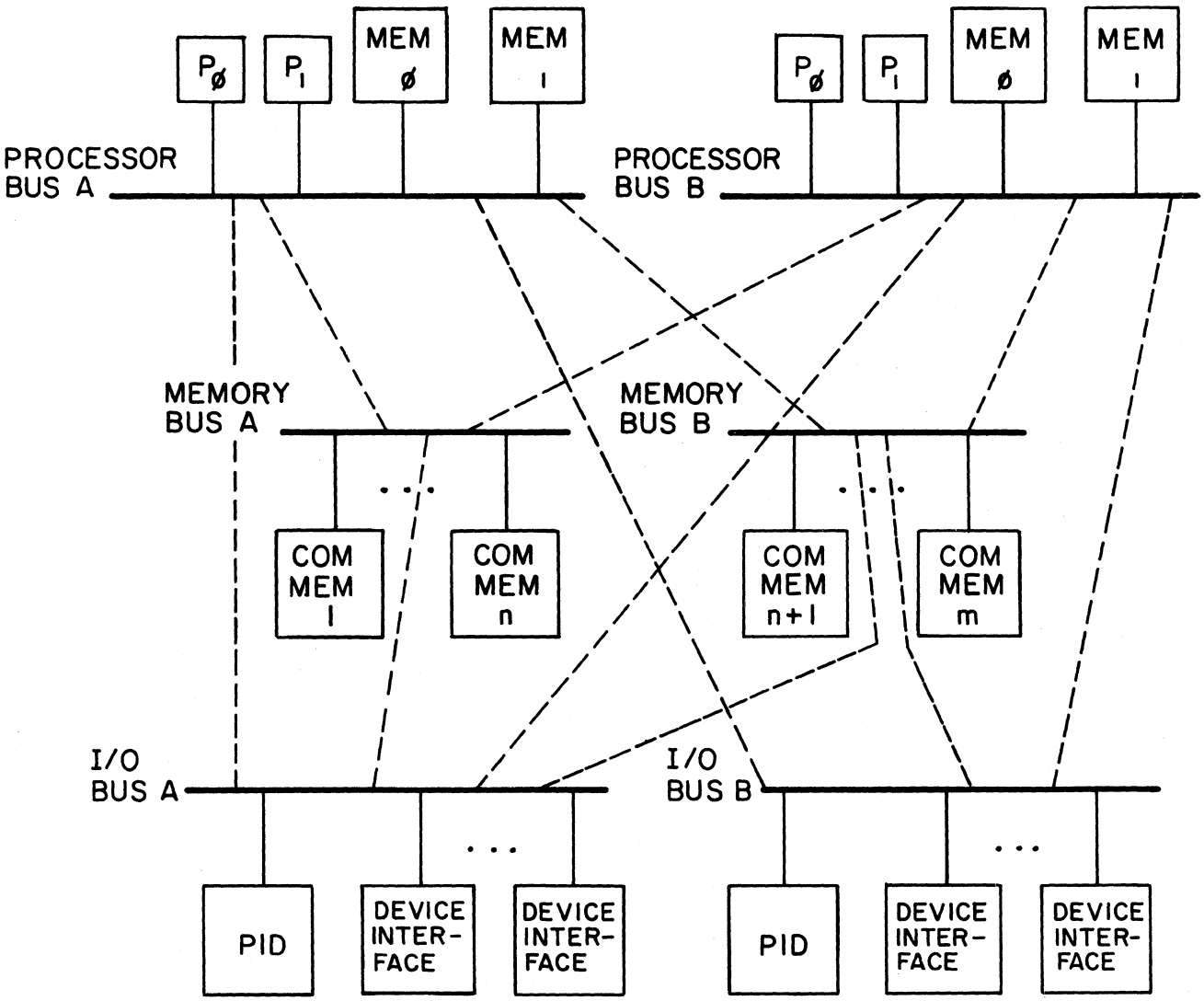


Figure 1 Typical Pluribus System Configuration

is for all private memories to contain identical copies of the same code. Much of the system reliability software will be held in the private memories to guarantee that redundant copies exist in case of any memory failure. The maximum amount of private memory addressable by each processor is 16K bytes. Not shown in Figure 1 but existing on every bus is the BCU (bus arbiter) card. In certain cases it may be desirable to have some I/O devices on the processor bus, but this will be the exception rather than the rule and is discussed further in section 6.4.

Memory busses contain common memory shared by all processors. Up to 1008K bytes of common memory can be added in 8K or 16K byte increments. The common memory will typically contain code which is referenced less frequently than the "hot code". Generally, shared data structures, variables, and buffers will also be held in common memory.

The configuration of common memory, that is, the assignment of memory modules to memory busses, depends on considerations of reliability and memory contention. For both reasons it is desirable to have multiple memory modules on a bus, multiple busses, and redundant copies of code and data structures. The details are application dependent and relate to the cost/performance (reliability) trade-offs which the system designer must consider. For reliable operation at least two memory busses, two processor busses, and two I/O busses will be required.

The I/O busses contain I/O devices and the Pseudo Interrupt Device (PID) central to the Pluribus system operation. The PID keeps in hardware a list of what to do next. A number can be written to the PID at any time and it will be remembered. When read, the PID returns (and deletes) the highest number it has

stored. By coding the numbers to represent tasks, and keeping the parameters of the tasks in memory, a processor can access the PID at the end of each task and determine very rapidly which task to do next. This approach is an important departure from the use of conventional interrupts and avoids the costs associated with saving and restoring machine state.* Further, this approach neatly sidesteps the problem of routing interrupts to the proper processor. More detail on the use of the PID is given in section 5.

There can be no more than four PIDs in a Pluribus system. Even though some I/O busses may conceivably not contain a PID, or a bus may contain more than one, the usual configuration is one PID per I/O bus.

In a Pluribus system, processor, memory, and I/O busses are connected by devices called bus couplers. The different types of bus couplers required to connect different bus pair types together are discussed in more detail in section 9. For moderate sized systems, there will generally be a bus coupler between any two busses between which communication is required. Usually this implies a coupler from each bus to all busses of other types. Thus the total number of bus couplers for such a Pluribus system with P processor busses, M memory busses, and I I/O busses is $P \cdot M + M \cdot I + P \cdot I$. For smaller systems it is possible for one or more Infibusses to serve as combined memory and I/O busses, reducing the number of required bus couplers. For applications requiring large numbers of components (processors, memories and I/O devices) it will be possible to reduce the required number of bus couplers by building hierarchical Pluribus systems where

---

*Although not used within application software, conventional interrupts can result from errors and are used for special purposes. See section 3.3.

busses are not completely connected. A more detailed discussion of the issues and procedures for configuring a Pluribus system can be found in a separate document [6].

Several distinct processor models are available. The SUE is a relatively slow and inexpensive processor. Typical memory-to-register instructions have execution times on the order of 4 microseconds. For a given application, the required processor power can be attained by using as many processors as are necessary. This approach to generating high throughput systems has the advantage of permitting extreme modularity and high reliability as well as graceful degradation.

3. PROCESSORS

3.1 Instruction Set and Format Summary

Lockheed SUE processors are used as processor components for Pluribus systems. The basic processor is a microprogrammed general purpose 16-bit minicomputer with 8 general registers (one of these registers is the program counter), and a status and a control register. These registers (general purpose, status, and control) may be accessed externally by other devices via the Infibus. In a multiprocessor this allows one processor to stop another, examine and change its registers, and restart it. There are 8 general instruction classes: MOVE, ADD, SUBTRACT, INCLUSIVE OR, EXCLUSIVE OR, AND, COMPARE, and TEST. Each of these instructions can use a variety of addressing modes including register-to-register, memory-to-register, register-to-memory, indexed, indirect, and auto-indexed. Also available are rotate, shift, conditional branch, unconditional branch, and subroutine calling instructions. The conditions tested for branching are bits in the status register of the SUE. There are tests for result being zero, result being negative, carry on last arithmetic instruction, register value odd, overflow, value greater-than on last comparison, value equal on last comparison, and loop completion. The branch can occur on either value TRUE or FALSE. Instructions are either one or two words long. The processor also contains 3 programmable flags, contained in the status register, that can be manipulated directly by instructions. The SUE processor recognizes and generates 16-bit addresses. In addition it contains a 2-bit KEY register which is settable by the SKEY instruction in the processor. The contents of this register are appended to the most significant end of the 16-bit address to generate an 18-bit address. Every memory access by a processor has these two bits appended.

Certain of these 18-bit addresses are mapped into 20-bit system addresses as described in section 4. The processor operates on either 16-bit words or 8-bit bytes of data. Bit Ø is identified with the low order bit and bit 15 with the high order bit. Details of the SUE instruction set and the various processor types may be found in reference 4.

3.2 PROCESSOR STATES

SUE processors can be in one of three states: halted, running, or idle. Transitions between these states may be effected either by the processor itself or by external manipulation. The implications of each of these three states are as follows:

Halt: No instructions executed, interrupts disabled, registers externally accessible

Run: Instructions executed, interrupts enabled, registers not externally accessible except control register.

Idle: No instructions executed, interrupts enabled, registers not externally accessible except control register.

External references to registers which are not accessible will result in a QUIT, as described in sections 5.6.1 and 9.2. The Idle state is entered from the running state by executing a WAIT instruction; the HALT state, by a HALT instruction. The Run state is entered from the Idle state by the occurrence of an interrupt. External manipulation of these states operates as follows:

The processor control register is the only register accessible while the processor is running. To halt a processor, a one is written to its control register. The processor will normally halt

when the instruction it is currently executing is completed. However, if the control register is read between the time that a zero is written to the control register and the time that the processor completes its current instruction, the halt signal will be lost. Hence, some algorithm such as the following should be used to guarantee that a processor does in fact halt:

```
L:  Write 1 to the processor control register.
    Read some other processor register.
    If QUIT results go to L (see section 3.3).
    At this point the processor is halted.
```

To start a processor, one must initialize all important registers to needed values and store the number two into the control register. This is done as follows: First it must be assured that the processor is halted. The program counter is initialized to the address of the program to be executed. The general registers are loaded with any values to be passed to the program. The status register is initialized to specify the enabled interrupt levels, initial status flags, and programmable flags. Bit 11 (hexadecimal constant Ø8ØØ) should be set to activate the processor. Finally, writing a 2 to the control register starts the processor. An additional feature of the SUE processor is the ability to single step through an instruction sequence. The procedure for doing this is identical to that of starting a processor except that a 3 is written to the control register rather than a 2.

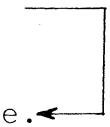
3.3 QUIT Handling

Processors requesting access to memory locations or I/O registers do so by directly or indirectly placing the desired address on the appropriate bus along with the required operation (e.g. read, write) and any data required (for a write operation). When the requested operation is complete, the processor will receive a signal called DONE. If no device on the destination bus recognizes the address provided or if the device recognizing the address malfunctions, no DONE signal will be returned to the processor. Instead, after a fixed period of time, the bus arbiter on the requesting processor bus will send a QUIT signal to the processor, causing a conventional interrupt. (An equivalent thing happens to requests by I/O devices from I/O busses.)

In many applications a programmer will want to take some action based on the presence or absence of QUIT interrupts. In the ARPA Network application, for example, a device discovery routine in the reliability software searches system address space and determines if known devices have disappeared or new ones appeared by attempting to read the devices' registers and checking for resulting QUITS. To provide this mechanism the interrupt level routine which responds to QUITs should be written so that control will be passed back to the application program

if the application programmer has indicated that he wishes this to happen. He indicates this wish by surrounding the instruction which potentially causes the QUIT by an "unusual pattern" of other instructions. For example, if a programmer wants to check for a QUIT occurring when location ABC is referenced he might write the following:

```
        LDA         A2, ABC
        NOP
        BR          . + 4
L:      QUIT BRANCH ADDRESS
    If no QUIT, program continues here.
```



The QUIT interrupt service, upon receiving control, would check to see if the two instructions following the one which caused the QUIT were NOP and BR . +4. If they were, it would simply store the two bytes starting at location L (the address of the instruction causing the QUIT plus 8) in the program counter and dismiss the interrupt. If the two instructions at L-4 and L-2 do not match the NOP BR . +4 pattern, the interrupt service routine would take its usual action in handling the QUIT. Of course, references to ABC which do not cause QUITs cause execution to continue at L+2 as indicated.

4. ADDRESSING

A typical Pluribus configuration incorporating both private memory associated with each processor and a pool of common memory shared among all processors has been presented in section 2. In this section the Pluribus address structure is described in more detail. The application of this address structure to Pluribus program structures will be discussed in section 5.

In Pluribus systems all devices communicate with one another by writing into or reading from addresses. These addresses may be memory locations, locations for controlling or interrogating I/O devices, or they may have some other special function. In any case it is important to understand two things; first, how addresses are generated and routed through the system and second, what things are referenced by what addresses.

Addresses are generated by active devices, that is devices wanting to read or write some location. This includes both processors and I/O devices. Consider first a processor.

As indicated in Figure 3(a), SUE processors normally generate 16-bit addresses. With the addition of the 2-bit KEY register in the SUE, however, the Pluribus processors actually generate 18-bit addresses which are put on the processor bus. The KEY register can be changed under program control by execution of the SKEY instruction. For the class of applications being considered in this document, however, each processor on a bus will initially set its KEY register to a distinct value indicating its physical processor number and will not

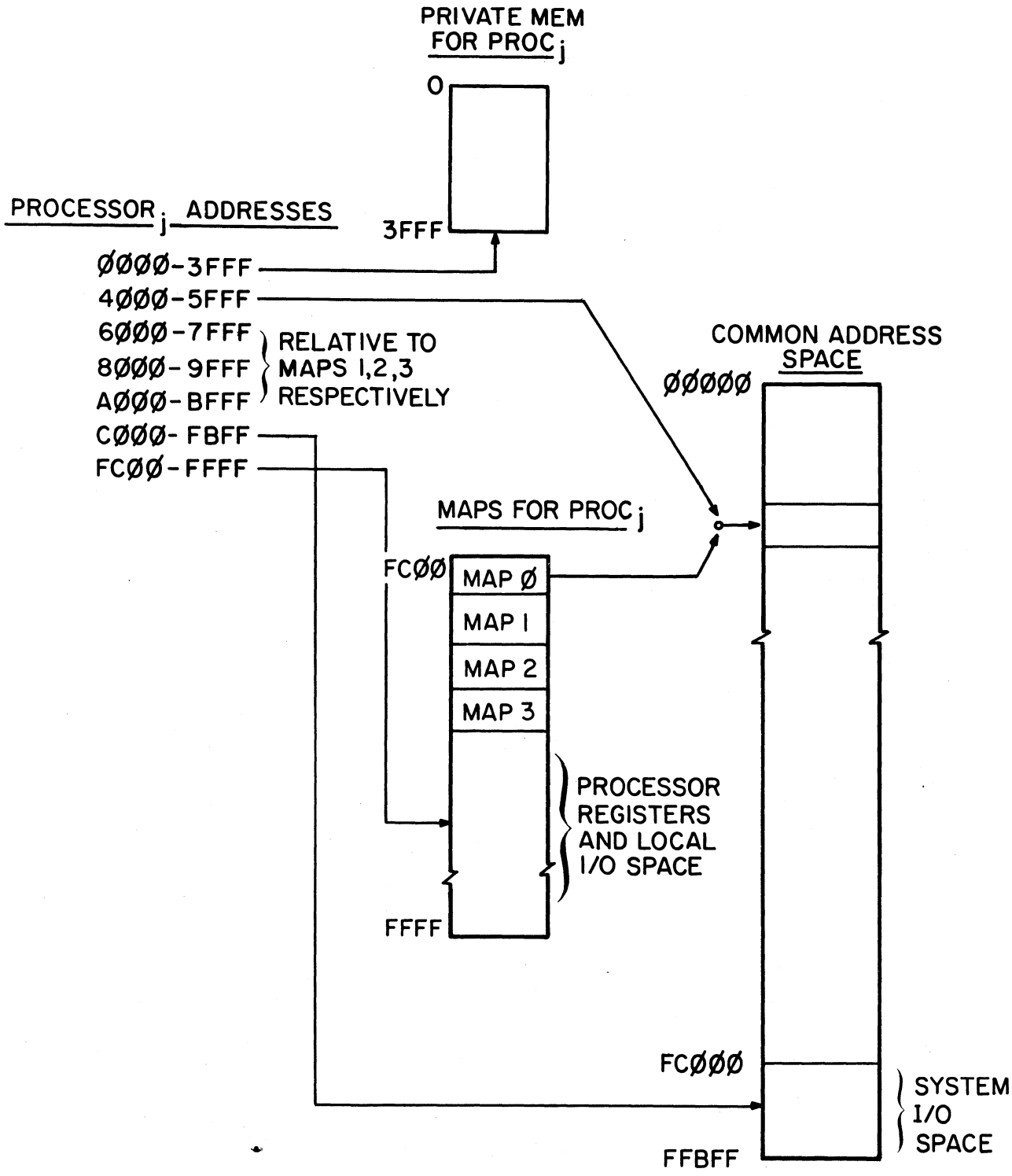


Figure 2 Processor Address Space

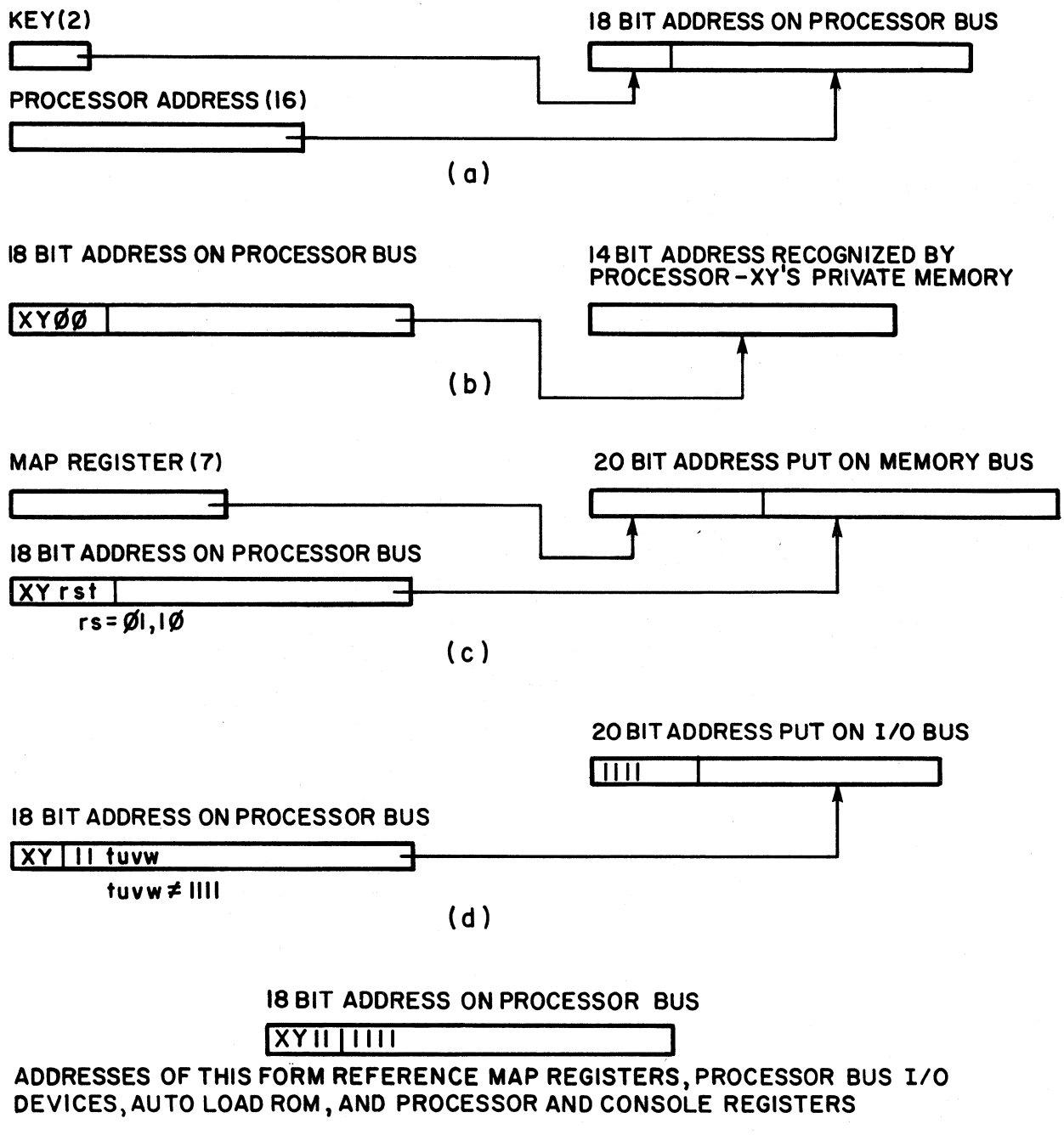


Figure 3 Address Mappings

normally change this setting.

The KEY bits thus serve to differentiate the address spaces of the various (up to four) processors on the bus. The right-most bit of the 16 left to each processor is used to select left or right byte in byte mode instructions, allowing 2^{15} = 32K addressable words. In order to allow larger common memory and I/O space than this, provision has been made for mapping portions of this 32K processor address space onto a 512K word system address space.

The addresses shown at the left in Figure 2 are the 16-bit addresses generated by a typical processor (processor j). The manner in which the address space is accessed by any processor generated address depends on the range in which that address lies. The four types of access will be discussed individually below. With all 4 types of access being discussed, the 18-bit address is simply put on the processor bus. Devices (memory, bus couplers, and I/O gear) able to recognize that address will respond and all others will ignore the address.

4.1 References to Private Memory (ØØØØ-3FFF):

Any processor generated address in this range refers to a location in the private memory associated with processor j on its processor bus. Up to 16K bytes of private memory can exist. As shown in Figure 3 (b), the high order two bits of the 18-bit address are used to select the proper memory module and the low order 14 bits select the location within the private memory.

4.2 References to Common Memory (4ØØØ-BFFF):

Any processor generated address in this range refers to a location in common memory, that is, memory on one of the memory busses. There are 4 distinct sub-ranges within this range, each associated with a distinct hardware mapping register. This association is indicated in Figure 2. Each map register allows a contiguous set of 8K bytes of commom memory locations to be referenced. A separate set of 4 map registers is associated with each processor on a processor bus. The map registers are physically located in the bus couplers (see section 9.1.)

Figures 2 and 3(c) illustrate how this mapping into common memory is accomplished. An address in the range 4ØØØ-5FFF implicitly refers to map register Ø. A 2Ø-bit system address is developed at the processor end of the coupler by appending 7 bits from the map register to the low order 13-bits of the 18-bit address on the processor bus. This address is then forwarded to the common memory bus with the access request.

Addresses in the range 6ØØØ-7FFF, 8ØØØ-9FFF, and A000-BFFF implicitly refer to map registers 1, 2, and 3 respectively and the identical type of mapping occurs when these sub-ranges are referenced. The only special feature in the way that the four maps work is related to memory read operations via map register 3 which are transformed into read-modify-write accesses to common memory where the data rewritten is always zero. This allows the implementation of multiprocessor locks in the Pluribus system. More detail on the use of this feature is discussed in section 5.3 where Pluribus program structures are considered.

One final complication arises from the fact that a few of the first addresses on every memory and I/O bus are allocated for accessing the bus coupler control registers. The amount of this allocation depends on the number of couplers connected to the bus. In general, the memory words at these addresses should not be used. For more detail on the bus coupler control registers see section 9.2.

4.3 References to System I/O Space (CØØØ-FBFF):

Any processor generated address in this range refers to a location in system I/O Space. In general, each Pluribus system device on an I/O bus appears to the processor as a set of 8 contiguous registers (locations) in system I/O space. This block of registers is referred to as the device register block. A processor can activate a device by writing commands or data to certain (device dependent) addresses within the device register block. A processor can interrogate a device by reading data or status registers within the 16 byte device register block. More detail about the allocation of system I/O space to multiple I/O busses and about the internal structure of the device register block can be found in section 6 where device handling and I/O is discussed further.

As indicated in Figure 3(d), the way that system I/O space addresses are developed is by appending four ones to the low-order 16 bits of the 18-bit address on the processor bus. This is done automatically as is the appending of the map registers (discussed above) by hardware in the bus couplers.

4.4 References to Maps, Processor Registers, and Local I/O Space (FC00-FFFF):

Any processor generated address in the range FC00-FFFF (see Figure 3(e)) refers either to the map registers for that processor (in all the bus couplers attached to the processor bus) or refers to a part of the address space shared by all processors on a processor bus. The map registers must be addressable, of course, so that a processor can dynamically modify the portions of the potentially large common memory to which it has access. Map registers Ø, 1, 2, and 3 can be referenced via addresses FC00, FC02, FC04, and FC06 respectively.

The local (to the processor bus) shared address space is assigned as shown in Figure 4. In general, I/O devices will be attached to an I/O Infibus in a Pluribus system. In some cases, however, it may be desirable or necessary to connect I/O devices directly on a processor bus. Addresses in the range FC08 to FDFF will be used in such a configuration to refer to the device registers, similar to the way that the device register block is used for referencing devices on the I/O Infibus. The auto load ROM is an optional hardware device attached to the processor bus which contains a program that when executed will cause a processor to load memory from a paper tape reader on the processor bus. The registers of all the processors and the processor bus console are accessible at addresses above FF00. A processor should be halted before an attempt to read any of its registers occurs. Halting a processor is described in section 3.

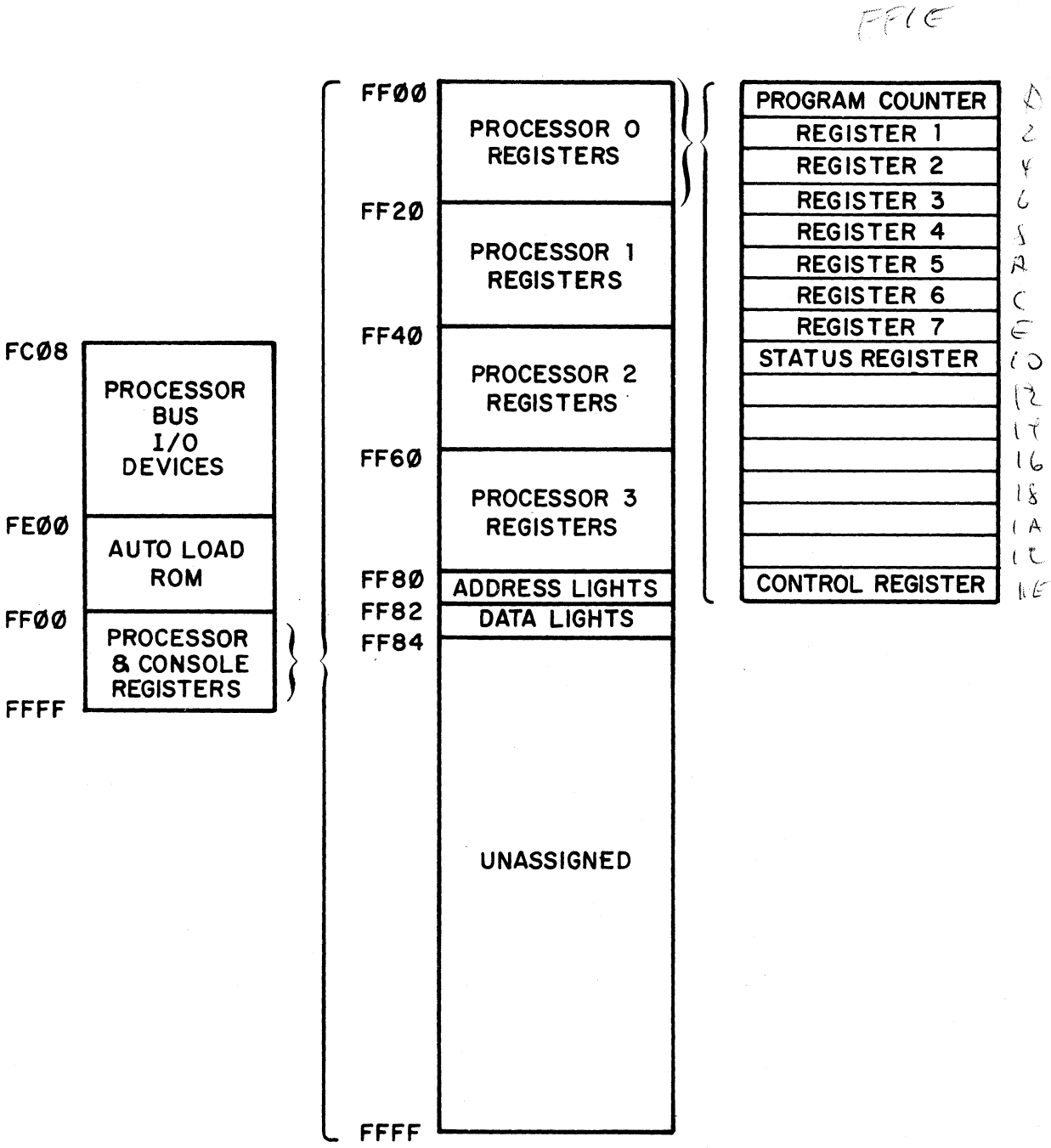


Figure 4 Processor Bus Shared Address Space

5. PLURIBUS PROGRAM STRUCTURE

In most current computer systems a hardware priority interrupt mechanism is used to inform the program of the occurrence of asynchronous external events. Since Pluribus systems do not generally use interrupts for this purpose, Pluribus programs tend to be structured differently from programs developed for conventional machines. The fact that Pluribus programs are designed to operate in a multiprocessor environment imposes additional constraints on the program structure. This section presents some of the issues and programming techniques which we believe are useful in developing Pluribus programs.

5.1 Basic Control Structure:

Before giving an example of a typical Pluribus program control structure, the basic operation of a PID will be reviewed (more detail on the PID can be found in section 10.1). The PID is a priority ordered memory device. It has a read address and a write address. When an even 8-bit number is written to a PID, the number is stored. When a PID is read, the largest 8-bit number stored in the PID will be returned and the number deleted from the PID. If nothing has been written to the PID, the read will return a value of zero. Numbers may be written to the PID both by hardware I/O devices and by software. Processors poll the PID for tasks to be executed. As a simple example of a Pluribus control structure, consider a system consisting of a number of tasks which service a set of I/O devices. The following assembly language code could provide the framework for the required program.

```
TASKDISPATCHTABLE:   MAINLOOP, TASK1, TASK 2, ..., TASKN

     MAINLOOP:  LDA A1, PIDREADADDRESS
                JMP @ TASKDISPATCHTABLE (A1)

          TASKi:
                }
                }
                }
                JMP MAINLOOP
```

The main loop of the program simply reads the PID and jumps to the appropriate task indirectly through TASKDISPATCHTABLE (i) where i is the value obtained by reading the PID. At the end of any task (e.g. TASKi), a jump to the main loop returns the processor to look for the next task to perform. If there is nothing in the PID, zero is returned and the processor simply cycles at MAINLOOP. Note that it is useful to have the PID store even numbers only since the number retrieved will be used as an index into a table with two-byte entries.

To allow tasks to be initiated by the software (e.g. TASKi to be initiated by TASKj), the following type of structure would be used:

```
     TASKj:
                }
                }
                }
            LDA   A1, TASKiPIDLEVEL
            STA   A1, PIDWRITEADDRESS
                }
                }
                }
            JMP   MAINLOOP
```

5.2 System Response Time and Strips:

As indicated in the above example, the way that I/O devices obtain service in a Pluribus system is to write the priority level of their service routine to the PID when they need attention, and wait for some processor to return to the main loop and pick up the associated task. Since the time that a device can wait for service before losing data may be critical, it is essential to configure systems and design software so that response time requirements can be met.

The two main factors which influence the rate at which a Pluribus system can respond to high priority external events are the total number of processors in the system and the duration of task servicing instruction sequences. For example, in a single processor system where the tasks are all of the form illustrated by the two previous examples, if the longest task execution time were T milliseconds, the maximum time which it could take to respond to an external event (i.e., notice that it had occurred) would also be T milliseconds. This worst case would happen only when the event occurred just after the single processor had picked up the longest task to run. Since in a Pluribus system there are no interrupts, the entire task currently being executed runs to completion before there is a reaction to the event (even though it may be of higher priority than the task currently being run).

In the multiprocessor case, things are slightly more complicated. Considering the worst case response time as above, if the ordered task execution times are T_1 (smallest), T_2, T_3, ... Tn (largest) and there are P processors, the maximum time to respond to an external event assuming n>p will be between between T_{n-p} and T_n depending on the number of incarnations of

a particular task which can exist simultaneously. Of course, the probability of such worst response times may be exceedingly small if the large tasks are run less frequently than the smaller tasks.

Typical (average) rather than worst case response times will depend on three factors: (1) average task execution time, (2) number of processors P, and (3) average number of tasks, N_Q, queued on the PID. If the average task execution time is T_{av} and $N_Q \geq P$, the typical time taken to service a high priority event will be $T_{av}/2$. If $P > N_Q$ then there will usually be an idle processor which will immediately react to the external event and average response time will be essentially zero.

In general, the application will dictate where strict real-time response must be guaranteed or if more flexible system response characteristics are adequate. If strict real-time response is required, then some program structure which permits both logical tasks of arbitrary length and fast response to critical external events may be required. To accomplish this, Pluribus program tasks can be partitioned into code segments referred to as strips. A strip is simply a sequence of instructions within a task. A task can give up control of its processor at the end of each strip so that any higher priority tasks may be run. Of course, if the task is incomplete at the end of a strip, the task queues itself on the PID for further execution before yielding its processor. The idea is illustrated by the example below where TASKk is broken down into two strips.

```
DISPATCHk:   K1          /INITIALIZE TO THIS VALUE.

    TASKk:   JMP @ DISPATCHk
      K1:
               LDA   A2, = K2
               STA   A2, DISPATCHk
               LDA   A1, TASkPIDLEVEL        } Strip 1
               STA   A1, PIDWRITEADDRESS
               JMP   MAINLOOP

      K2:
               LDA   A2, = K1
               STA   A2, DISPATCHk           } Strip 2
               JMP   MAINLOOP
```

The first instruction of TASKk is a dispatch to the segment (strip) of the code to be executed. This dispatch is initialized to K1 so when TASKk is first initiated, execution will begin at K1. At the end of strip 1, the task stores a new dispatch address (K2) in the subtask dispatch location, DISPATCHk, writes its own PID level back into the PID and gives up the processor. The next time this PID level is serviced, the task will be resumed in strip 2 starting at K2. At the end of Strip 2, the subtask dispatch location is restored so that strip 1 will be executed the next time that TASKk is activated. It must be kept in mind that a task writing its own level to the PID will prevent the processor which is executing the task from picking up a waiting task with lower priority. In certain situations it may be desirable for a task to yield the processor and also "sleep" a specified period prior to getting rewritten to the PID. This can be accomplished by the task setting a software timer which

gets counted down by a periodic clock routine. When the timer reaches zero, the clock routine can write the sleeping task's level to the PID. The 5 instructions at the end of strip 1 in the above example might, therefore, be replaced by the following:

```
LDA  A2, = K2
STA  A2, DISPATCHk
LDA  A2, SLEEPTIME
STA  A2, TIMERk
JMP  MAINLOOP
```

Then after TIMERk has been counted down, the timer routine will execute the instruction:

```
LDA  A1, TASKkPIDLEVEL
STA  A1, PIDWRITEADDRESS
```

The decision of precisely where to segment a task into strips is somewhat arbitrary; the main rule is that the strips must be short enough so that the proper response characteristics can be guaranteed. In the ARPA Network application of the Pluribus, for example, it turned out that the proper typical strip size was on the order of 100 instructions (although a few infrequently run ones are much longer). As a rule of thumb, it will generally be sufficient to segment a task into strips assuming each instruction takes 4 μsec for execution.

Two other related practical issues relevant to strip size selections are convenience and overhead. In general, tasks should be broken into strips at convenient points in the code; that is, points at which little information (e.g. in the registers) needs to be preserved. It may occasionally be desirable to have strips somewhat smaller or larger than the nominal size so that such a

partitioning will be possible. Data which must be saved and restored across strip boundries adds to the already existing overhead associated with breaking the code into strips. In many applications it is likely that little or no breaking of tasks into strips will be required. In the ARPA Network application, for example, multi-strip logical tasks are the exception rather than the rule.

The fractional overhead associated with breaking a task into strips depends directly on the strip size since the number of instructions required for strip switching is essentially fixed. For example, in TASKk presented above 8 overhead instructions are associated with switching from one strip to another (6 in TASKk and 2 in the main loop). If the strip size were 100 instructions as is typically the case in the ARPANET application, then the processor overhead due to using strips would be 8%. In applications where larger strips are acceptable, of course, the overhead will be even smaller. Experience with a number of Pluribus system applications has indicated that the processor overhead and programmer effort associated with breaking tasks into strips is not a serious problem and is a relatively small price to pay for the increased reliability and performance of the novel Pluribus architecture.

5.3 Shared Data Structures, Shared Code, and Locks

In a multiprocessor care must be exercised when a piece of data may be referenced (read and/or written) simultaneously by more than one processor. In this context, "simultaneously" means that a process running on one processor desires access to the data while another process running on a second processor already has access to the data. Consider, for example, two processors that are concurrently executing processes which

obtain buffers from a common free storage list. If some interlock is not used, it would be possible for both processors to get the same buffer since the second processor could access the list after the first processor had accessed it but before the pointer was updated.

To avoid this and a multitude of similar situations involving shared resources, a lock mechanism is typically used in programs for multiprocessors. Before a shared resource is accessed by a process, a logical lock is set. All processes determine if the lock is set prior to accessing the resource, and if so, then the process will wait. Only one process can, therefore, have access to the shared resource at any one time.

To be effective it must be possible to test and set a lock in a single operation. A typical implementation provides the ability to read, test, and provisionally modify a memory location in a single interruptable operation. In Pluribus systems this feature is provided by turning memory reads through map register 3 into read-modify-write accesses where the data rewritten is all zeros.

To implement a lock on a shared resource one simply assigns a location (LOCKVAR), addressed through map register 3, to the lock. The resource is unlocked if the lock is non-zero and locked otherwise. A segment of code which accesses a locked resource might look as follows:

```
L:    LDA  A2, LOCKVAR       (Lock and continue) or (WAIT)
      BZ   L
        }
        }                     access to shared resource
        }
L1:   STA  PC, LOCKVAR  /     Unlock Lock
        }
        }
```

If a processor falls through the loop at L, the resource was unlocked but is now locked by the process running on this processor. If a processor loops at L, then the shared resource is in use and the processor waits until the lock is released. To unlock the resource at L1 any non-zero quantity could have been stored in LOCKVAR. The current program counter (PC = general register 0) contains one such value which has the additional advantage of leaving a partial trace of the program execution in the lock registers. This trace may be helpful for debugging purposes.

When a process encounters a locked resource, it may take one of two actions. As in the above example, it can remain in a tight loop checking the lock until it is unlocked. This type of waiting will be called busy waiting since the processor running the process remains occupied while waiting. Alternatively, a form of non-busy waiting may be implemented where the process may either write itself to the PID or set a timer so that a clock routine will later write it to the PID as described earlier. In either case the processor then is free to seek other tasks while waiting. The busy form of a lock is appropriate in situations where the resource will be locked for only a short period. An example of this is the free buffer list accessing mentioned at the beginning of the discussion on locks. The lock implementation which dispatches the processor to do other useful work will be more suitable in situations where the shared resource is likely to remain locked for a relatively long time. A paper tape reader shared by two processors might be such a resource.

The preceding discussion leaves considerable latitude with respect to what should be locked and when. For example, if each incarnation of a piece of shared code references a set of shared variables, it may be more efficient to associate a single lock

with the set of shared variables than a lock with each of the individual variables. What needs to be balanced against this goal of fewer locks is the desire to keep locked segments short. Large locked segments, while reducing the total number of locking/unlocking operations required, will tend to increase overhead due to increased busy waiting or processor task switching. This overhead can become quite large on the percentage utilization of the shared resource increases beyond 60 - 70%. For this reason, the system designer must use considerable judgement in deciding on the extent of locked segments. In addition, locks should not remain locked across strip boundaries. Locked segments should also be executed with interrupts disabled so that prompt unlocking of the shared resource is assured.

One further consideration is that a processor may fail while executing a locked segment. Two problems can arise in this case, (1) the locked resource will be unavailable to other tasks and (2) if busy waiting is implemented, processors may be executing infinite loops. Therefore, a processor should only be allowed to wait for the maximum amount of time which the lock can legitimately be set before deciding that a malfunction has occurred and activating a recovery procedure.

Cooperation with respect to the use of shared variables is required between tasks corresponding to different code segments and especially tasks corresponding to different incarnations of the same reentrant code segment. In general, reentrant coding is particularly appropriate in a multiprocessor such as the Pluribus system. The shared code may exist in common memory or multiple copies of the code may exist in the private processor memories to reduce contention. In the ARPA Network application, for example, shared code is used to transmit data from the IMP to each of a number of modems. In this case, the control structure

illustrated earlier in this section is modified to look as follows:

```
TASKDISPATCHTABEL:  MAINLOOP, TASK1, ••• , MODEMOUT, MODEMOUT, •••TASKN

CONTROLBLOCKS:  Ø, BLOCK1, ••• , MBLOCK1, MBLOCK2, ••• BLOCKN

    MAINLOOP:   LDA  A1, PIDREADADDRESS
                JMP  @TASKDISPATCHTABLE (A1)

    MODEMOUT:   LDA  A2, CONTROLBLOCKS (A1)
                LDA  A3, MODEMLOCK (A2)
                     }
                STA  PC, MODEMLOCK
                JMP  MAINLOOP
```

The modem interfaces each write different levels to the PID when output of a buffer is complete but all these levels activate the same piece of shared code, MODEMOUT. The PID levels are used, however, to select the address of a control block which contains the variables specific to the modem being serviced. At the start of MODEMOUT, an instruction is executed which loads an accumulator, (A2), with the address of this control block. One of the words in this block is a lock used to lock all the other shared variables in the block. These variables remain locked for the duration of the modem output tasks.

5.4 Using the Map Registers:

The map registers allow four independent 8K byte segments of the common memory to be referenced by each processor. The only constraint is that a read done through map register 3 will be a read and clear. The other three map registers may be used to

point to program or data as required by the application. It is possible to have two map registers point to the same segment of memory. In the ARPA Network application, for example, map 3 and one of the other map registers point to a segment containing system variables which can be accessed normally or used as lock variables.

In Pluribus systems with small memory configurations little of no map changing may be required. For applications requiring large primary memories, map changing will be more frequent. Of course, it is desirable to design a system so that as little map changing as possible will be required. To change the area of common memory addressed through a particular map register, one simply stores into the map register a constant whose high order 7 bits are to become the contents of the map. As already mentioned in section 4.4, the four maps have addresses FC00, FC02, FC04, and FC06. The code which changes a map must not itself be referenced through that map/ One way to make sure that this does not occur is to execute all map changing code out of private memory.

5.5 Using Multiple PIDs

The PID is the heart of the Pluribus system. Essentially all task dispatching is done via this device. It is important, therefore, that reliability provided by redundancy in the remainder of the Pluribus system components not be jeopardized by availability of only a single PID.

In a multi-PID system, the PIDs will themselves be priority ordered. Typically, the control program in such a system will read the highest priority PID first. If a PID other than the lowest priority PID returns zero, the next lower priority PID will be read. If all PIDs return zero, the control program simply

cycles by reading the highest priority PID again.

As indicated earlier, a Pluribus system can have up to 4 PIDS, one on each of 4 I/O busses. A hardware device on an I/O bus is associated with a PID on that bus. Software tasks, on the other hand, may write to any of the PIDs in the system. Redundant I/O devices will generally be on different I/O busses and associated with different PIDs.

6. DEVICE HANDLING AND I/O

Pluribus systems may be comprised of two types of I/O devices, BBN-developed devices and Lockheed-developed devices. The primary distinctions between the two are that BBN devices interpret 20-bit addresses and use the PID while Lockheed devices interpret 16-bit addresses and utilize the standard SUE priority interrupt mechanism. Since SUE I/O programming is discussed at length in [2], most of this section will be devoted to the specifics of programming BBN-developed devices. Special considerations relevant to the programming of Lockheed I/O devices in a Pluribus environment are given at the end of the section.

6.1 Address Structure

As shown in Figure 2, system addresses FC000 to FFBFF are reserved for Pluribus system I/O space. The detailed structure of this space depends on the allocation of addresses to I/O busses. Figure 5 shows one possible allocation of addresses in the case of a Pluribus with 2 I/O busses. Possible variations on this structure will be indicated later.

The total system I/O space in Figure 5 is divided into four almost equal parts, two of which are assigned to each bus. The high address segment for each bus will be referred to as the primary I/O space and the low address segment as the auxiliary I/O space. Note that the primary address space of bus 1 (from address FF000 to FFBFF) is shorter than the other 3 segments by 1024 bytes because these 1024 addresses are allocated to individual processor maps, registers, and local I/O space as shown in Figure 2. At the beginning of each primary address space are 144 bytes of reserved addresses. These locations are associated with the clock (RTC) and PID on the bus (see sections 10.1 and 10.2), contain the bus coupler (BCM) control registers

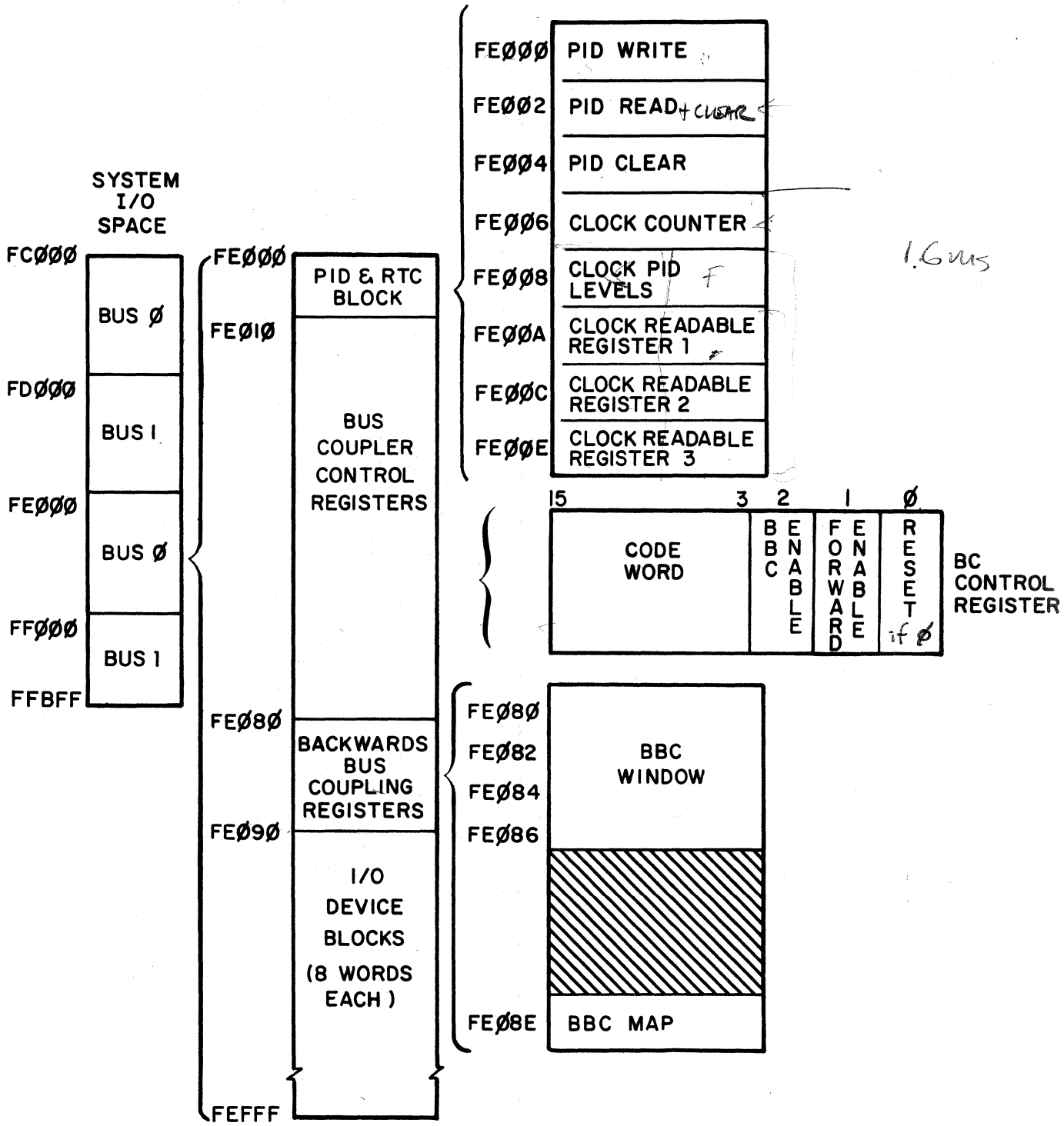
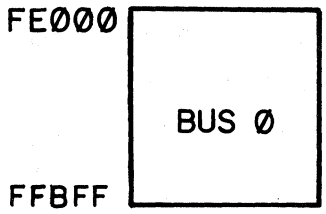


Figure 5 System I/O Space

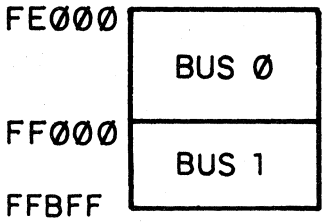
(see section 9.2), and provide mapping for backwards bus coupling (see section 7.1.3) using this bus.

The remainder of the system I/O space is divided into 16-byte blocks where each block is associated with an I/O device (other than the clock and PID) attached to the bus. These blocks are called device register blocks. A processor activates an I/O device by writing to a certain address within the device register block. A processor can interrogate a device by reading the contents of status registers contained in this block. More detail on the structure of device register blocks is given below and is also contained in section 10. where individual I/O devices are discussed.

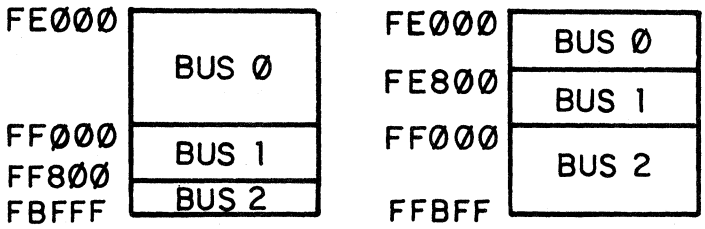
Variation of the structure shown in Figure 5 depends on the number of I/O busses and the allocation of system I/O addresses among them. This allocation is determined by switches on the bus couplers (see section 9.2). Figure 6 indicates allocations of system I/O space for 1, 2, 3, and 4 I/O busses. Only the primary I/O space allocations are shown; the auxiliary allocations are identical to these except that the highest address segment of auxiliary is the same size as the rest of the segments, that is, it is not reduced in size by 1024 bytes. The low 144 bytes of each primary segment is reserved on each bus as indicated in Figure 5. While other allocations are possible, the ones shown in Figure 6 constitute all of the reasonable ones. Switch settings resulting in non-contiguous primary and auxiliary segments for individual busses, while possible, are not considered here.



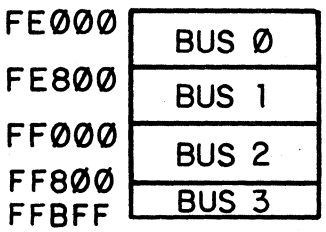
(a) ONE BUS



(b) TWO BUSSES



(c) THREE BUSSES



(d) FOUR BUSSES

Figure 6 Allocations of Primary System I/O Space

6.2 Programming BBN DMA (Direct Memory Access) I/O Devices

BBN DMA (Direct Memory Access) devices provide a means for the automatic transfer of blocks of data to (from) memory from (to) I/O devices on the I/O busses. While the DMA hardware and its associated device interface are on separate cards, from the programmer's viewpoint they may be thought of as a single unit.

In general, each data transfer will involve sending or receiving a number of data buffers. Each data buffer will consist of an integral number of words. For each direction of data flow (read, write) there are three main registers used by the programmer to control I/O operations; the begin memory (buffer) address register, the end memory (buffer) address register, and the status register. These registers are contained in the 16-byte device register blocks. The structure of the device register blocks for BBN DMA devices is shown in Figure 7. Each of these registers is described in detail below.

DEVICE TYPE - The high order byte contains a number indicating the type of device interface involved (e.g. modem, host, etc.). This number is fixed by hardware in the device interface associated with the DMA. In general, the low order byte contains the value set in the device number switches in the device interface. The device type register is readable; writing to it will have no effect.

RECEIVE/TRANSMIT BEGIN ADDRESS - These registers contain the high order 16 bits of the 20-bit system address specifying the first location of the buffer to be read or written. Bits 1-3 of the 20-bit starting address are contained in the receive or transmit status register (see below). Bit 0 of the 20-bit system address is always Ø. The beginning address registers may be

either read or written. If read, the result returned is simply zero. Normally when writing into this location, no data transmission will be in progress in the direction corresponding to the register written (receive or transmit). The device will simply be initialized to transfer a buffer; actual data transfer does not commence until the buffer end register is written. If a transfer is in progress when the location is written, the transfer is aborted, the error bit (in the end address register - see below) is set, the PID is written, and the corresponding half (receive or transmit) of the device is initialized for transmission of a new buffer.

RECEIVE/TRANSMIT END ADDRESS - These registers may be read or written. Normally, bits 0-12 of these registers will be written with the low order 13 bits of the address of the end of the buffer. (Bit 0 is actually ignored and assumed to be zero.) Writing to this address initiates the data transfer. After the data transfer has ended, these registers can be read to determine information concerning the way that the transfer completed. Bit 15, if set, indicates that no error has been detected and that this was the last buffer of the transfer. (Bit 15 will be set when the last buffer is transmitted correctly.) Bit 0 serves as an error bit and will be set if: (1) the device was reinitialized during the previous transmission (see above), (2) a QUIT occurred during transmission of the previous buffer, (3) the device is currently active (see RECEIVE/TRANSMIT STATUS below) or (4) the device itself is reporting an error. Bits 1-12 of the end address register indicate the address, modulo 2^{12}, of the last work actually transferred. The top 7 bits of the DMA pointer into the buffer come from the begin address (see above) and never change. Therefore, the buffer will "wrap around" on 8K byte boundaries in memory.

RECEIVE/TRANSMIT STATUS - The receive and transmit status registers may also be both read and written. Writing the RESET bit causes the particular half of the interface (receive or transmit) to reset itself. If that portion of the interface is active when the reset is initiated, the operation in progress will be aborted, the error bit in the end address register will be set, and the receive or transmit level for the device will be written to the PID. Before initiating a DMA data transfer, bits 1-3 of the buffer beginning location must be written into bits 0-2 of the corresponding status register. Reading one of the status registers allows a processor to determine the PID level associated with that direction of data transfer and to interrogate the QUIT flag. The PID will be written and the QUIT flag will be set if a QUIT occurred during the previous data transfer performed by the DMA. This could indicate a parity error, non-existent address, etc. In this case, when the end pointer is read, the error bit will be set.

The interpretation of the device dependent status bits varies from device to device but in general these bits provide for direct two-way communication between a processor and a device interface.

One of the device dependent bits will be the ACTIVE bit which, if set, indicates that a transfer to or from the device is in progress. More precisely, a DMA device is active from the time that its end pointer is written (which starts the device) until the time that it writes its level to the PID (indicating it is done).

DEVICE DEPENDENT - This register can be optionally used by the device interface for any appropriate function. The assignment of data bits is arbitrary.

To cause a transfer to be performed by a BBN DMA device, the program will typically perform the following steps:

1. Write the STATUS REGISTER - This sets up the low-order 3 bits of the buffer start word address and selects any desired options (e.g., looped modem). This will normally be done only once for a sequence of DMA transfers.
2. Write the BEGIN ADDRESS REGISTER - This sets up the 16 high order bits of the buffer start address.
3. Write the END ADDRESS REGISTER - This sets the end address of the buffer and initiates the DMA transfer.

When the PID level, indicating device completion, is picked up by a processor, it will:

4. Read the END ADDRESS REGISTER and check bit 15 (completion). If it is not set, the transfer has completed (i.e., no error occurred and this is the last buffer of the transfer). Bits 1-12 are used to give the length of the buffer.
5. If this bit is not set, bit Ø (error) is checked. If bit Ø is zero, then no error occurred but this buffer is not the last of the transfer. As above, bits 1-12 are used to determine the length of the buffer.
6. If bit Ø is one, then an error has occurred. These are differentiated by examining the STATUS REGISTER. If bit 13 (active) is set, the device is still active and the PID value was spurious. If bit 8 (QUIT) is set, a QUIT occurred during the transfer. Device dependent status bits may further define the error.

In addition to the registers mentioned above, each BBN block transfer type of device has a number of manually settable switches. These switches, located on the device interface, are as follows (number of switches provided for each purpose shown in parentheses):

(i) Device Address Switch (10) - These switch settings define

the address of the device register block in I/O space (see Figure 5). The ten switches specify bits 4-13 of the address of the first register of the block. (Bits 14-19 of this address are all ones and bits 0-3 are all zeros.)

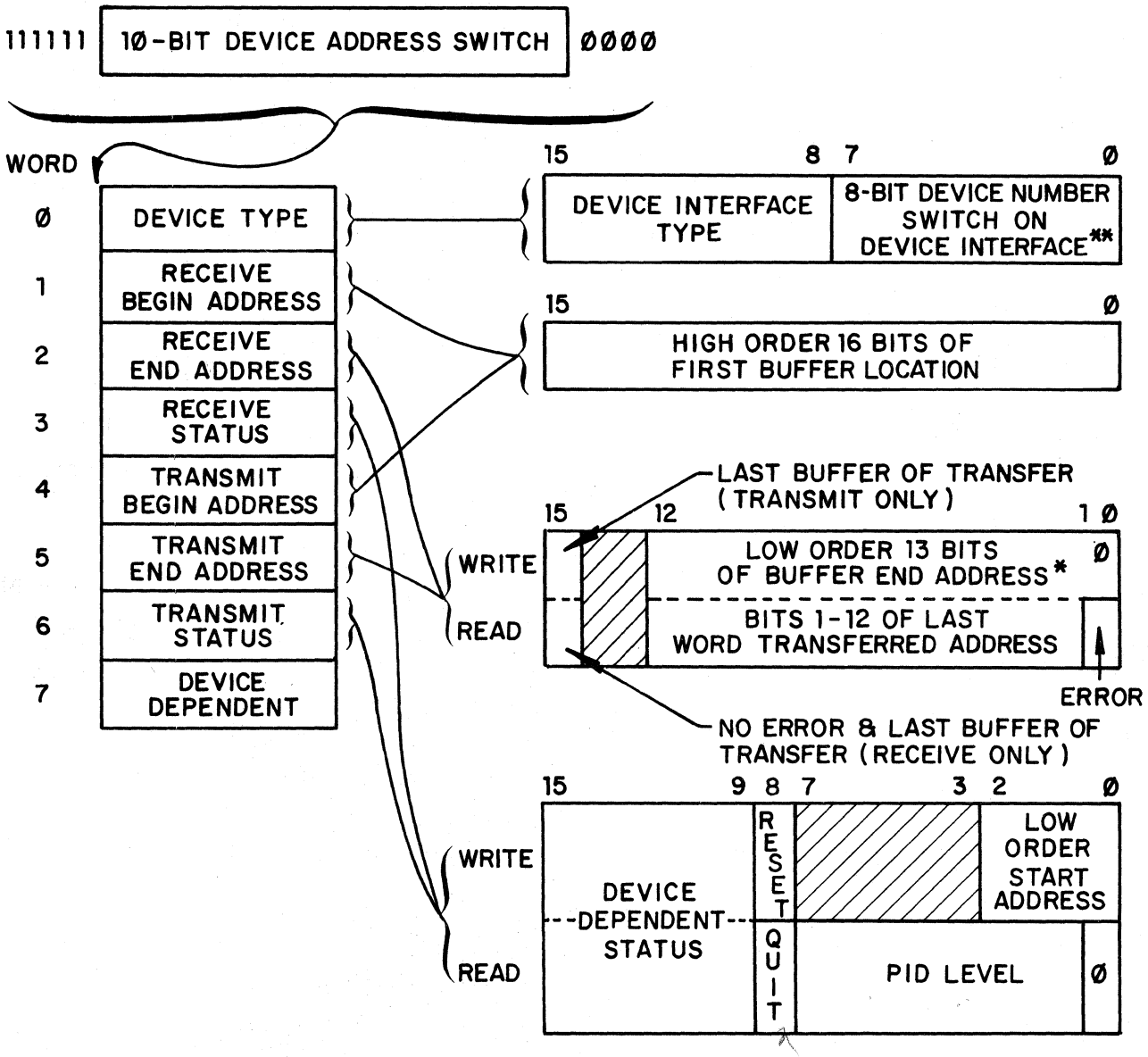
(ii) Receive/Transmit PID levels (7) - These seven switches define the number written to the PID upon completion of a data transfer. For duplex devices there are two sets of switches. For simplex devices only a single set is provided (e.g., CBT - see section 10.5).

(iii) PID Address (2) - Selects which of the 4 PIDs will be written to by the device. The selected PID must be on the same Infibus as the device itself.

(iv) Device Number (8) - In general, a set of 8 switches readable as the low-order byte of the first word in the device register block (see Figure 7). The CBT device, however, has only one such switch.

6.3 BBN Non-DMA I/O Devices

Typically, non-DMA devices will only have a small amount of internal hardware buffering, therefore, they need to be serviced by a processor no slower than every few byte times. The mechanism by which such a device is serviced can take one or two forms in Pluribus systems. One approach is to let the device be passive and put the responsibility for servicing the device completely on the processors. For input, the processors would have to poll the devices faster than the input rate so that no data is lost. For output, the processors would have to deliver data to the devices at a rate sufficient to guarantee that no undesirable gaps within the data occur. Although such an approach permits a relatively simple hardware interface implementation, it may require an undesirable amount of processor overhead.



* BIT 0 ON TRANSMIT END HAS A SPECIAL INTERPRETATION FOR THE CBT DEVICE (SEE SECTION 10.5)

** ONLY ONE SWITCH EXISTS FOR CBT DEVICE (SEE SECTION 10.5)

Figure 7 DMA Registers

An alternate approach is to make the device active with respect to notifying the processors when it requires service. In a Pluribus system this implies that the device will write its level to the PID when its internal buffers are ready. Checking whether the device needs service will, therefore, be done automatically as part of the main PID reading loop of the program. Such an approach, of course, requires more hardware in the device interface than does implementation of the first approach mentioned above.

The only BBN non-DMA I/O device which currently exists is the synchronous line interface (SLI) which is described in detail in section 10.7. This device is passive and consequently requires polling by the processors. Both DMA and non-DMA I/O devices which are addressed through system address space will have 16 byte device register blocks associated with them. In contrast to the DMA device register blocks which have a common format for all DMA devices, the structure of the non-DMA device register blocks will be device dependent.

6.4 Lockheed SUE I/O Devices

As indicated above, standard SUE I/O devices differ from those developed specifically for the Pluribus system in that (1) they interpret 16-bit addresses rather than 20-bit addresses and (2) if they are set up to actively notify the processor when they require service, they do so via a hardware priority interrupt mechanism rather than via the PID. Since it will often be desirable to incorporate such devices in a Pluribus system, some procedures for interfacing and programming them need to be developed. There are two distinct approaches that may be taken. First, sufficient modifications could be made so that the device will work on the system I/O busses. This approach has the advantage that the I/O device will be accessible to any processor in the system. It has

the disadvantage that hardware modifications probably need to be made to the device hardware. The other approach is simply to have the LEC device reside on one of the processor Infibusses, the place for which it was designed. This approach has the disadvantage of essentially isolating the device from the processors in the system on other processor busses but has the advantage of requiring no hardware modifications.

If the first approach is taken, that is, the device is put on an I/O bus, the hardware modifications required depend on whether the device will be active or passive. In either case it will be essential to modify the device interface to recognize 20-bit addresses. If it is to be a DMA-type device it would also be required to generate 20-bit addresses. In the ARPA Network application, for example, a system control teletype has been interfaced in this manner and is handled by the processors as a passive device. Programming of LEC devices on the I/O busses will be similar to the BBN I/O devices discussed above. The details, of course, depend on how the device interfaces are modified.

The only logical difficulty with putting LEC I/O devices on the I/O busses arises in the case of high speed DMA devices which require fast servicing for proper operation. To guarantee that such devices will be serviced within a specified time is likely to impose unacceptable constraints on the size of the strips into which tasks are partitioned. In such cases, e.g., handling a disk, it will probably be essential to take the second approach mentioned above and interface the device on one of the processor busses.

Programming LEC I/O devices on a Pluribus processor bus is essentially identical to programming them in a standard multiprocessor SUE configuration. The only difference arises with DMA devices which are dealing with buffers in Pluribus common memory.

Since the devices only produce 16-bit addresses, some mapping mechanism similar to the processor address mapping is required here. This can be accomplished by simply dedicating one of the first three map registers associated with processor 0 to the I/O device for the duration of the time the device is being used. The I/O device addresses which have no key bits set appear to the bus couplers as requests from processor 0 and are mapped accordingly. I/O interrupts, when they occur, are always routed to processor 0. Even though it is undesirable from an overall system reliability standpoint, this dependence on a specific processor is unavoidable.

More detail in programming specific LEC peripherals can be found in the LEC SUE Computer Handbook [2].

7. SYSTEM RELIABILITY MECHANISMS

The hardware architecture of Pluribus systems which provides a foundation for the development of reliable computer systems has already been presented. This section describes both some additional hardware mechanisms which have been included to improve system reliability and a general description of some software mechanisms which when operating on the Pluribus hardware can create a reliable computing environment in which to execute application programs.

The interpretation of reliability is strongly related to the type of applications for which a computer system is intended. At one extreme are computations which do not have strict real-time constraints, for example, large numerical computations. For these applications, reliability may mean simply checkpointing, that is, dumping intermediate states of the computation on a mass storage device so that the computation can be continued without much wasted effort should a system outage occur. At the other extreme are real-time control applications in which no outages are allowed and every request must be serviced within a short fixed time period. Such applications may require simultaneously running the system on several identical hardware configurations with decisions based on a majority vote. Although the Pluribus system can be applied to applications in both of these two classes, the applications for which it was specifically designed fall somewhere between these two. The Pluribus system will be most appropriate in situations where it is important to maximize MTBF and minimize MTTR but where occasional outages and minor delays in servicing requests can be tolerated.

A reliable Pluribus system will generally be configured with at least one redundant copy of each hardware resource essential for running the overall system. It will be the goal of the reliability software to maintain a "working set" of resources and, if

possible, backup spares for each of them. In general, the reliability software will attempt to recover from failures of single hardware resources.

7.1 Hardware Reliability Mechanisms

The following mechanisms can be used both by the Pluribus reliability software described later in this section and applications programs which choose to use them directly.

7.1.1 Power Failure/Restart Interrupts:

(i) Processor Infibusses - The Infibus provides power for each of the devices it contains. If the power supply is about to fail on a processor Infibus, processor Ø on that bus receives a level 4 interrupt with device code 2. Processor Ø then has approximately 2.5 milliseconds to signal the other processors and save any important data on a non-failing Infibus or in non-destructive memory. When a processor Infibus is without power, the Control Register of any bus coupler connecting this Infibus to another is still modifiable.

When power is restored to the Infibus, a reset of the Infibus is executed by the BCU and each device on the bus will reinitialize itself. Each processor will enter an idle state with all registers zeroed. Processor Ø will then execute a level 4 interrupt with device code 4 (power restart).

(ii) Memory and I/O Infibusses - When power is about to fail on a common Infibus, processor Ø of each Infibus connected to the common Infibus executes a level 1 interrupt with device code 1. The processor must then

read the control register of each bus coupler on its Infibus to determine which one caused the interrupt. If the Control Register is Ø*, then the attached Infibus is losing power.

When the processor determines which common Infibus is losing power, it has 2.5 milliseconds to signal other processors, save important data stored on the failing Infibus, and mark that Infibus unusable by the program. While the Infibus is without power the bus coupler map registers are still modifiable.

It will be necessary for the processors to periodically check the dead Infibus to see if power has been restored. When this is the case, the Control Register will read (hexadecimal) 21ØØ*.

7.1.2 Hardware Timeouts:

A philosophy prevalent in the Pluribus hardware and software is that the system should perform sufficient introspection to recognize illegal and deadlocked states. If such states are detected, actions sufficient to move the system into some legal state should be initiated. The Infibus and device timeouts discussed below are two implementations of this general philosophy.

*This may be modified by the contents of any overlapping memory location (see section 9.2).

7.1.2.1 **Infibus Timeout** - The Bus Control Unit monitors the frequency of activity on the Infibus. If there are no accesses for one second, the BCU will execute an Infibus RESET and each device on the Infibus will reinitialize itself. Processors on the bus will enter an idle state with their registers set to zero. Processor Ø will subsequently receive a level 4 interrupt with device code 4 power restart.

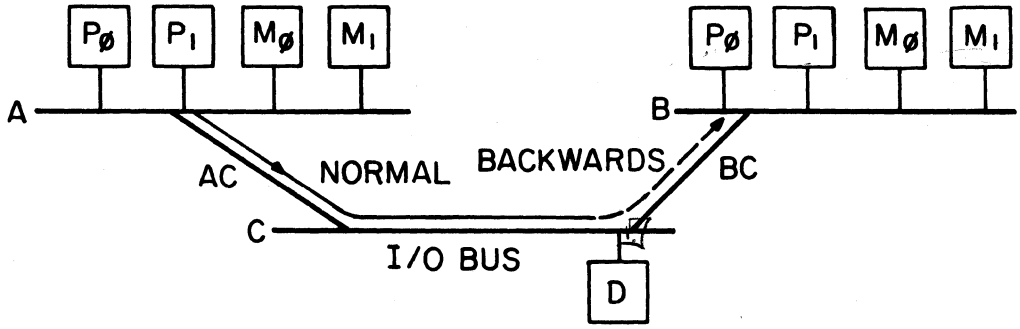
7.1.2.2 **Device Timeout and Multiple Interfaces** - In many applications of the Pluribus it will be important to have redundant (multiple) interfaces to one or more of the I/O devices in order to improve system reliability. In the ARPA Network application, for example, multiple interfaces to modems and Host computers are planned. Since such multiple interfaces will share a single I/O device, it will be necessary to electrically disable the device-interface transmit path on all interfaces but the one currently in use at any given time.

Rather than have the enabling/disabling of these paths controlled by the processors, the interfaces themselves are provided with sufficient logic so that the decision can be made locally. Each device interface which permit other interfaces like itself to be connected to a shared I/O device is equipped with a hardware timer. This timer is reset whenever a specific (device dependent) word in the corresponding device register block is accessed. If the word is not accessed for a fixed time period, the timer runs to zero and the associated device - interface path is disabled. The path will be enabled whenever the specific word in the device register block is next

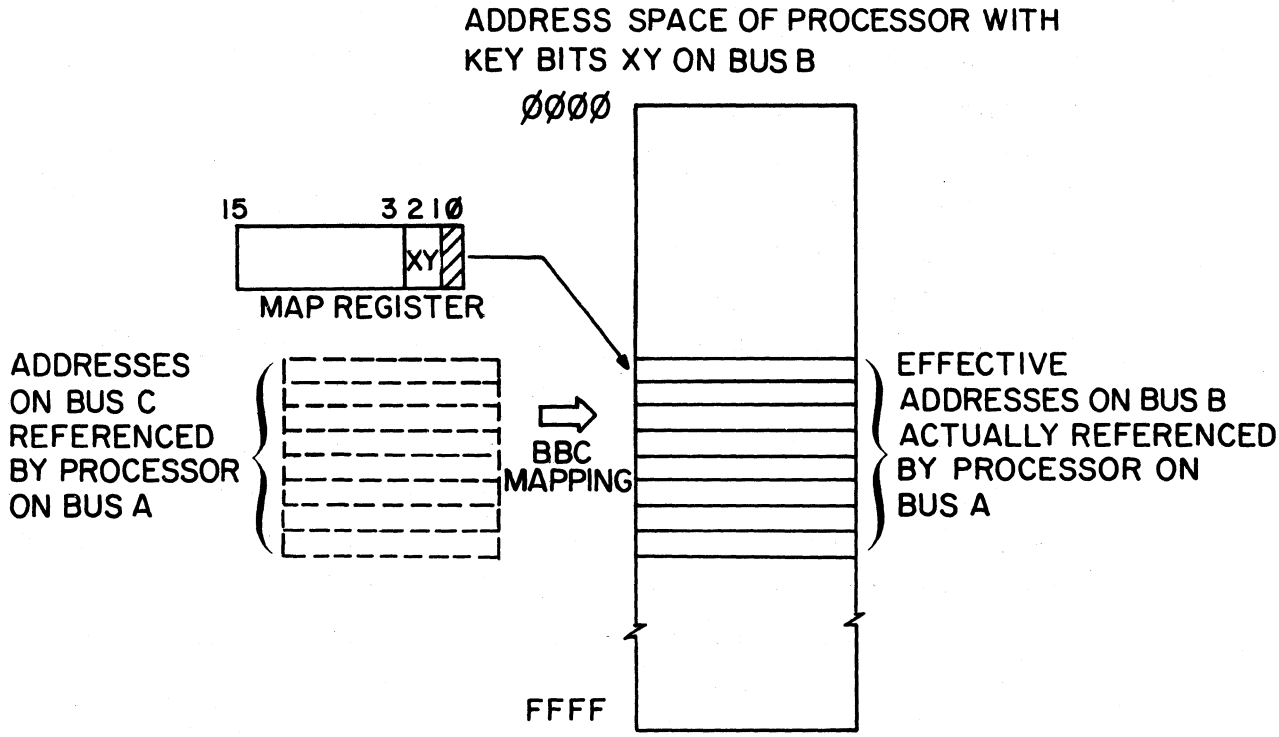
referenced. A processor can, therefore, switch from one interface to a spare by simply stopping references to one interface and starting references to another. If for some reason an interface cannot be referenced, it will soon return to its stable, legal, disabled state. All DMA devices currently implement this facility and have one second timeouts. If a transfer is in progress when the interface timer reaches zero, the interface will short the transfer, write its level to the PID, and set the error bit in the associated device register block. The specific words in the device register blocks which must be referenced in order to reset the timers are given in section 10, where the different I/O devices are discussed.

7.1.3 Remote Reference/Control of Devices on a Processor Bus:

7.1.3.1 **Backwards Bus Coupling** - Using the type of interbus communication described so far, it is not possible for a processor on one bus to interact with a processor on another bus except by voluntary communication on the part of both processors through a mutually agreed upon portion of shared memory. A processor cannot directly halt, restart, or modify the registers or local memory of a processor on a different processor bus. To overcome this shortcoming, the Pluribus has an additional mechanism known as backwards bus coupling (BBC). Backwards bus coupling permits requests to be transmitted to a processor bus via a bus coupler as well as from a processor bus, the normal direction. This is illustrated in Figure 8a where a processor on bus A is trying to reference some address local to bus B via I/O Infibus C.



(a)



(b)

Figure 8 Backwards Bus Coupling

Backwards bus coupling from one processor bus to another is only possible over a shared I/O Infibus. In addition, only one bus coupler on an I/O bus can be enabled for backwards bus coupling at a time. Attempting to enable more than one BBC path on a single bus will produce unpredictable results. A lock will typically be associated with this shared "BBC path" resource.

For backwards bus coupling to proceed, the BBC enable bit in the Control Register of the bus coupler connecting the shared I/O bus to the target processor bus must be set to one. (See Figure 5) This can be accomplished by writing the hexadecimal constant DE7D to the control register if forward coupling is to be disabled or DE7F if forward coupling is to be enabled. The first 13 bits of these constants are a code word required to prevent indiscriminate modifications of the register by malfunctioning devices. After the BBC enable bit is set, the BBC Map Register (see Figure 5) may be set to point to the desired area of address space on the target processor bus (attempting to write the BBC map prior to enabling BBC will result in a QUIT). With BBC enabled and the map register set, reference to locations on the target processor bus may proceed by making reference to corresponding bytes within the BBC window (see Figure 5). After BBC accessing is complete, the coupler control register should be returned to its previous state by resetting the BBC enable bit.

Figure 8b illustrates the details of the BBC mapping in the context of the situation shown in Figure 8a. To reference locations within the address space of the processor with key bits XY on bus B, processor A loads

bits 2 and 3 of the bus C BBC map register with XY and bits 3-15 of this map register with the high order 13 address bits of an area on processor bus B. Subsequent references to bytes or words within the bus C BBC window are translated into 18-bit references on the target processor bus A at the address formed as follows:

XY	Bits 3-15 of Map Register	(low order 3-bits of address of byte referenced in BBC Window)

The BBC Map Register essentially serves as a base register which allows up to 8 bytes starting at the BBC map address to be referenced. Of course, the BBC Map Register will need to be updated quite a bit if any significant number of BBC references are required.

One further complication is the fact that simultaneous forward and backward bus coupling requests conflict. The result of such conflicts will be short term deadlocks while the Infibusses at both ends of the bus coupler time out their respective requests prior to sending QUITs to the requesting devices. In Figure 8a, for example, if processor P_0 on bus A were attempting to access memory M_0 on bus B at the same time that processor P_1 on bus B were attempting to access device D on I/O bus C, a deadlock would occur with respect to coupler BC. Busses B and C would, therefore, timeout the requests made to BC prior to sending QUITs to processor P_1 on bus B and bus coupler AC on bus C respectively. Since the time until a QUIT is returned is typically longer on a processor bus than on an I/O bus, however, the bus coupler AC

will generally receive the QUIT first and terminate the BBC request passing on the QUIT to the requesting processor, P_o on bus A. The forward bus coupling will then continue until completion. The point of this discussion is that the application program using the BBC mechanism should be aware that QUITs may result, be prepared to test for them (see section 5.5.1), and repeat the BBC request if necessary.

The following list summarizes the previous discussion with a typical sequence of steps to follow for BBC references:

1) Lock (or wait on lock) BBC path resource on I/O bus
2) Set BBC Enable bit in Control Register
3) Write Map Register
4)
⋮ Set of BBC References: Will involve sensing and reacting to QUITs and may involve changes to map register
n-2) Reset BBC Enable bit in Control Register
n-1) Unlock BBC path resource on I/O bus.

7.1.3.2 **Remote Resetting of a Processor Bus** - Writing a zero to bit Ø of the control register of a bus coupler connecting a shared (I/O or memory) bus to another processor bus will cause that processor Infibus to execute a RESET. All devices on the processor bus will be reinitialized; each processor will enter an idle state with all registers zeroed. A subsequent 60 cycle clock interrupt (level 4, device code 1) will reactivate processor Ø on the bus. As with writing the BBC Enable bit, the first 13 bits of the word written to the control register must agree with the hexadecimal constant DE78. In addition, bits 1 and 2

of the control register should be written so as to create the proper state with respect to forward and backward coupling after the reset.

7.1.3.3 **Bus Amputation** - Bus amputation provides a means of isolating selected active devices (I/O devices and processors) from the remainder of a Pluribus system. In this way malfunctioning devices can be prevented from affecting the healthy system components. The unit of amputation is the bus; that is, a whole bus must be amputated if any device on that bus is to be disabled.

Bit 1 of the control register of a bus coupler must be 1 if the coupler is enabled for forward bus coupling. By setting this bit Ø, therefore, all forward requests over the coupler can be blocked. By zeroing this bit in all bus couplers coming from a bus containing a malfunctioning device, that device can be removed from the operational Pluribus system.

A processor is not able to write the control registers of any couplers connected to its bus (see section 9.2), therefore, amputation must be accomplished by references to the control register arriving over a different path. In Figure 9, for example, coupler ac could not be shut off by Processor P_0 on bus A. Processor P_0 on bus B, however, could cut path ac with an access to the ac control register (in the address space of bus C) via coupler bc. As with writing the RESET and BBC Enable bits, the first 13 bits of the word written to the control register must agree with DE78 hexadecimal, therefore, the word to write to the control register to amputate a bus is DE79 if BBC is to be disabled and DE7D if BBC is to be enabled.

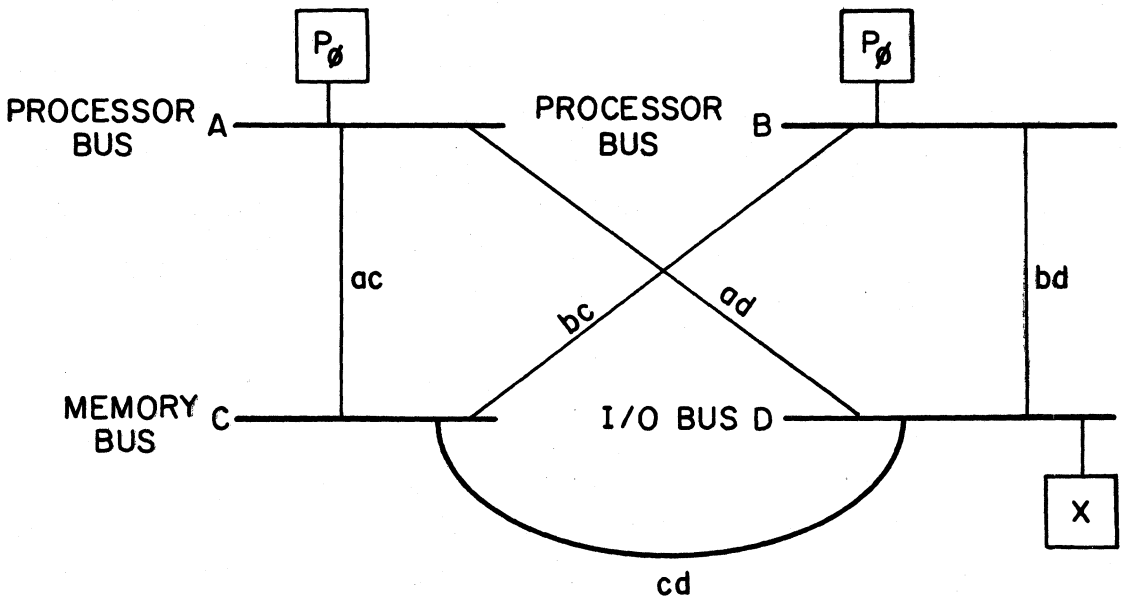


Figure 9 Bus Amputation Example

To illustrate, suppose that in Figure 9 processor P_o on bus B decided that processor P_o on bus A was malfunctioning and its bus should be amputated from the system. It could do this by simply zeroing bit 1 in the control register for couplers ac and ad. Similarly, if processor P_o determined that device X on I/O bus D was writing spurious data bits to common memory, it could isolate the device by zeroing bit 1 in the control register for coupler cd. This would effectively remove bus D from the system as far as memory transfers are concerned although addresses on bus D could still be referenced via couplers ad and bd.

7.1.4 Externally Initiated Reloads:

For the ARPA Network application it was necessary to develop a means of reloading Pluribus systems remotely over phone lines. In other communication applications, the ability to do this may also be important.

A special piece of hardware called the Reload (RLD) Device is available which resides on an I/O bus and monitors up to 8 modem interfaces (receivers). When the RLD observes a command in the input stream, it interprets the next 20 bits as a system address and the following 16 bits as a data word to be stored at that address. The address and data are heavily checksummed. Sequences of such commands can be sent to cause the RLD device to fill a portion of common memory or write via backwards bus coupling to the processor bus address spaces. Details concerning line protocol and device operation are given in section 10.6.

7.1.5 Parity Generation/Checking:

Parity generation and checking schemes provide a simple and effective way to detect many of the errors which occur in computer systems. The Pluribus uses such a scheme to automatically recognize memory failures and failures along the data transfer paths (i.e. bus couplers). This mechanism is invisible to the programmer except for the fact that a QUIT may result from a data access if bad parity is detected.

The type of parity calculated is called "address XOR Data" parity or AXD parity for short. AXD parity involves two parity bits for each 16-bit word, one associated with each 8-bit byte. Each parity bit is calculated as the exclusive-or of the address parity and contents parity of the byte. The advantage of this parity function is that it detects: (1) one data bit in error, (2) all data bits zero, (3) all data bits one, and (4) one address bit in error.

There are essentially four distinct paths in a Pluribus system that implement parity checking. Each of these paths involves inter-bus transfers and is described below. Parity checking for data accesses on a single bus is not implemented.

(i) **Processor/Common Memory Path** - When data is being written to common memory the processor end of the bus coupler computes the AXD parity bits and sends them to be stored in the memory. On reading a memory location the stored parity bits are retrieved and returned to the processor end of the bus coupler which recalculates the parity and matches it against the retrieved bits. A QUIT will be generated if these two sets of parity bits do not match.

(ii) **I/O Device/Common Memory Path** - The procedure here is virtually the same as for the Processor/Common Memory Path above except that the I/O end of the bus coupler rather than the processor end checks the parity bits.

(iii) **Processor/I/O Device Path** - When a processor writes (reads) data to (from) one of the devices on an I/O bus, a different sort of parity checking is performed. A special device on the target I/O bus, the Parity (PAR) card, continuously generates AXD parity from addresses and data placed on its bus during accesses to devices on that bus. This parity is fed back through the bus coupler involved in the reference and checked against parity computed at the processor end. A QUIT will be generated if the two sets of parity bits do not match. This technique is referred to as feedback parity checking.

(iv) **Backwards Bus Coupling Path** - Parity checking during BBC references is restricted to the forward part of the overall processor bus-to-processor bus path. The BBC registers are treated by the PAR card as if they were the registers of an I/O device on that bus, consequently the parity checking described in (iii) above applies.

7.1.6 Transfers Between Private Memories On the Same Processor Bus

Using the BBC mechanism it is possible for a processor on one processor bus to transfer data into the private memory of a processor on another processor bus (see section 7.1.3.1). It is also desirable for a processor to be able to effect transfers into the private memory of another processor on the same processor bus. Such transfers will be required, for example, when a processor on a bus wants to reload and restart another processing on that bus.

A recommended technique for doing this is described below. It involves running a short program out of the registers in one of the processors.

Suppose processor 0 wishes to transfer N words from its private memory starting at location SOURCE to consecutive words starting at location DESTINATION in processor 1's private memory. Processor 0 first stores the following program in processor 1's registers (starting at FF20):

```
LDA   A2, SOURCE (-A1)
KEY   1
STA   A2, DESTINATION (A1)
KEY   0
BLP   FF20                  /Address of Processor 1 register Ø
JMP   (A7)
```

Next, Processor 0 sets up the count N in one of his registers (A1 in the example above) by executing the following:

```
LDA   A1, = N
```

Finally, processor 0 executes the program on processor 1's registers via a:

```
JSB   A7, FF20   .
```

This example has assumed, of course, that processor 1 was initially halted and that the original contents of processor 1's registers either did not matter or were initially saved and later restored.

An attempt by the reader to work out some more straight forward solution should demonstrate the necessity of the sort of implementation described above.

7.2 Software Reliability Mechanisms

There are no strict constraints on the programmer concerning how the Pluribus hardware features can be used. These hardware mechanisms have been developed, however, with a particular hardware/software structure in mind. This structure will be described below. It should be pointed out that there does not currently exist any reliability software package that is available with a Pluribus system. The Pluribus reliability software which now exists is integrated into the ARPA Network IMP application of the Pluribus. Nevertheless, we believe that the basic ideas embodied in this software (and perhaps much of the code itself) can be applied in other Pluribus applications and are, therefore, worthwhile describing here.

One view of the relations between the three major software components in a reliable Pluribus system is shown in Figure 10. From the figure it can be seen that there are two major modules of the reliability software. The system reliability code is application independent and attempts to maintain a suitable set of resources in which to run the overall system. The application reliability code, on the other hand, is totally dependent on the particular application since it has responsibility for checking and fixing the data structures internal to the application program. To develop this module one must have a detailed knowledge of the states of that program. For this reason, the following discussion will focus on the structure of system reliability code module.

Under normal circumstances, the application program will be continuously running, executing application tasks fetched from the PID. The system timer routine which runs off of the real-time clock (RTC) causes both the application reliability code and the system reliability code to be periodically executed. The system

reliability code is comprised of a sequence of stages that are performed when activated. These stages include such tasks as calculating the checksum on programs in local and common memory, checking whether any memory or I/O device has either appeared or disappeared, maintaining original and spare copies of code and variable segments, and maintaining the running status of all processors by reloading and restarting them if necessary. If all these tasks can be performed successfully, the system reliability software will return to the application program. This will normally be the case. In some situations, however, the system reliability code may be required to supervise the initialization of the application program itself. Reinitialization of the application code, would be required, for example, if a segment of memory containing variables were taken out of service and a new portion of memory were allocated for this purpose.

An important concept associated with the system reliability module is that of processor consensus. Before a processor is allowed to run either the application program or the application reliability code, it is necessary to establish a common environment for all processors. This process of reaching an agreement about the environment is called "forming a consensus", and we dub the group of agreeing processors "the Consensus". The work done by the Consensus is in fact performed by individual processors, but the coordination and discipline imposed on the Consensus members make them behave like a single logical entity. An example of a task requiring consensus is the identification of usable common memory and the assignment of functions (code, variables, buffers, etc.) to particular segments. The members of the Consensus may not agree in their view of the environment, as for example when a broken bus coupler blinds one member to a segment of common memory. In this case the Consensus, including the processor with the broken

coupler, will agree to run the main system without that processor.

In addition to periodic activation by the system timer routine, the system reliability code will also be activated following certain exceptional conditions indicated in Figure 10. Several of these conditions have already been discussed. An extremely important mechanism not yet mentioned, however, is the 60Hz interrupt which is used to guarantee that each processor does, in fact, periodically run the system reliability code. Each processor upon executing the system reliability code sequence will reset a timer which the 60Hz interrupt service will count down. If the timer ever reaches zero, a processor has been lax for one reason or another and the reliability code will try to get the processor running correctly again. As is the case for periodic activations, the system reliability code will eventually either go to sleep or supervise the initialization of the application reliability routine or the application program itself.

The discussion in this section has only provided a brief overview of the Pluribus software reliability mechanisms which are, in fact, currently in state of flux. More details and additional motivation for many of the design decisions relating to Pluribus reliability mechanisms may be found in [7].

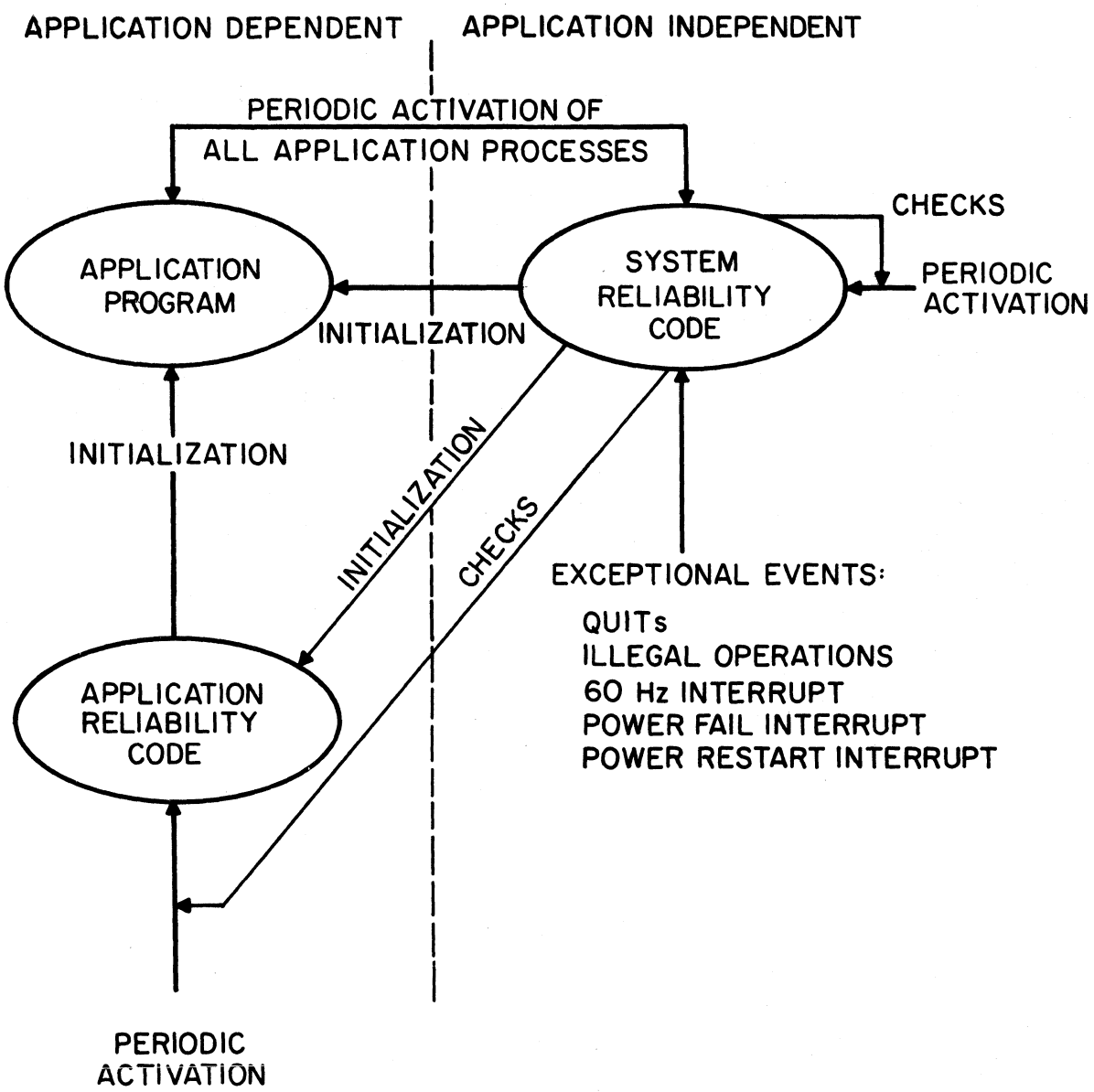


Figure 10 Reliability Software

8. INFIBUSSES

The Infibus is the primary power and communication pathway between devices. Physically, the bus is a panel containing 24 slots. Each device is inserted into one or more of these slots. Power and signal circuits connect all of the slots together. Power for the bus is provided by one of two possible power supplies: the smaller power supply is plugged into 8 of the slots of the bus, leaving 16 slots for devices; the larger power supply is external to the bus (leaving 24 slots for devices) and can provide power for one or more busses (depending on power requirements of the devices). The Bus Control Unit (BCU) module is necessary to control every Infibus. It occupies one slot, leaving either 15 or 23 slots for other devices. A bus can be extended by the addition of another bus cabinet. The electronics for the extension will occupy one slot in each cabinet, leaving 29, 37, or 45 slots for devices. The number of slots occupied by the major components of the Pluribus system are as follows:

Device	Number of Slots
Processor	2
8k bytes Memory	3
16k bytes Memory	3
Bus Control Unit (BCU)	1
Bus Coupler (BCP, BCM, or BCI - see section 9)	1
PID Pseudo Interrupt Device	1
RTC Real Time Clock	1
HLC Local Host Interface	2
CBT Checksum/Block Transfer	2
ML Low Speed Modem Interface	3
RLD Reload Card	1
PAR Parity Module	1
SLI Synchronous Line Interface	1

Electronically, the Infibus is the communications channel between devices. At any time, at most one device contained in a bus has access to that bus. This device can request data from another device contained in the bus (read) or request that another device receive the data that this device is providing (write). The device which has access to the bus is called the bus Master. The Slave device is the device the bus Master is transferring data to or from. The Master communicates with the other devices contained in the Infibus by providing the following information:

20-bit Address
One of the control functions:
    Read
    Write
    Read-Modify-Write
Whether data is word or byte
Data (if function is Write)
Parity

Each device contained in a bus continuously monitors the address being transmitted by the bus Master. A device becomes the Slave when it observes an address on the bus that it recognizes as its own. The device then performs the activity indicated by the address and control functions. When this activity is completed, a completion signal called DONE is returned. When the Master observes the DONE signal it accepts any data expected from the bus and relinquishes access to the bus. The Bus Control Unit has at that time already chosen the next device to be Master from among the devices which have requested access to the bus but have not yet received it. If no device recognizes the address that the Master provides the bus or if the Slave device malfunctions, then no action will be taken and no DONE signal will be returned. After

a fixed period of time (dependent on the particular bus), the Bus Control Unit will send a QUIT signal to the bus Master. The bus Master then relinquishes control of the bus and access is provided the next requesting device. The time allowed between access and a QUIT signal is established by the BCU hardware and is normally between 5 and 5ØØ microseconds. Processor busses will normally have the longest QUIT timeouts with I/O busses and memory busses having the next longest and shortest timeouts respectively.

Two different devices on a bus can recognize the same address. If both of these devices respond with action and a DONE, the system will likely malfunction. Devices must, therefore, use some criteria external to the bus to resolve which device becomes the Slave. Normally the address recognition switches on each device in a system will be set to recognize disjoint portions of the system address space.

The bus provides an initialization signal, called RESET, to each of the devices attached to it. This signal is transmitted to the devices whenever power is being restored, whenever the bus is reset from the console or from another processor, or whenever there has been no transaction on this bus in the last second. Each device will terminate any activity when it receives the RESET signal and reinitialize the state of all registers and indicators.

9. Bus Couplers

The functions of the bus couplers as components in an operational Pluribus system have already been discussed in several earlier sections. In this section the internal structure of the bus couplers is considered in more detail.

Each bus coupler connects two busses and, therefore, has two ends. Each end of a bus coupler appears as a normal device on its containing Infibus. The 3 types of ends (BCP, BCM, and BCI) and two types of bus coupler (BCP-BCM and BCI-BCM) that may exist in a Pluribus system are illustrated in Figure 11. BCP-BCM couplers are used to connect processor busses to either memory or I/O busses. BCI-BCM couplers are used to connect I/O busses to memory busses. The operation of the BCP, BCM, and BCI devices are presented below.

9.1 BCP:

Each BCP contains four 7-bit MAP registers for each of the four possible processors on the Infibus. The MAP registers are numbered 0-3 and are located in the address space of each processor at locations FCØØ-FCØ6. Each processor has its own set of MAP registers, selected by bits 16 and 17 of the 18-bit address of data on the Infibus. These two bits are specified by the last execution of the SKEY instruction in the particular processor. The MAP registers can be modified by writing the new contents of the MAP to the corresponding address FCØØ, FCØ2, FCØ4, or FCØ6. The high order 7 bits of the word written become the new contents of the map register. In general, bus coupler registers may be written but not read. Reading a MAP register gives a result of zero and does not change the register. Infibus RESET does not effect the contents of the MAP register. The contents of the MAP registers are unpredictable at power-up.

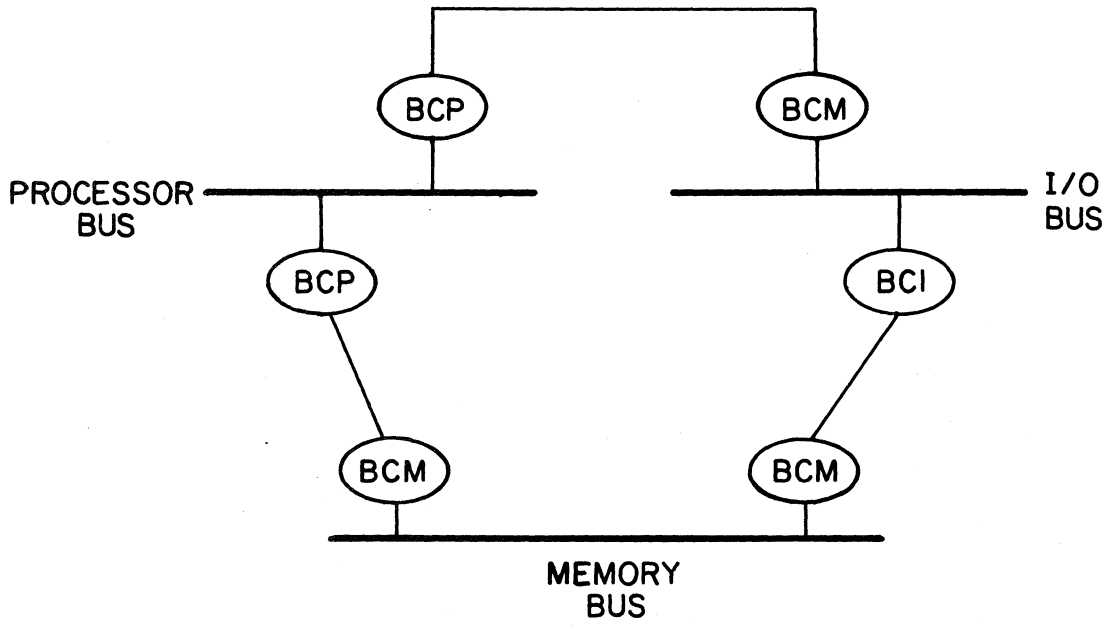


Figure 11 Types of Bus Couplers

During forward (normal) bus coupling the BCP is a Slave device on its bus and the BCP transforms each 18-bit processor address into a 20-bit system address to be sent to the BCM. As discussed in section 4., each processor's address space is divided up into 7 components:

Addresses	Description
0000-3FFF	References to Local Memory
4000-5FFF	Transform address using map 0
6000-7FFF	Transform address using map 1
8000-9FFF	Transform address using map 2
A000-BFFF	Transform address using map 3
C000-FBFF	Transform address to I/O space
FC00-FFFF	References to Processor Registers and Local I/O space

Addresses within 0000-3FFF or FC00-FFF are ignored by the BCP. For those 8k byte segments of processor address space utilizing a particular MAP, the BCP forms a system address by preserving the low order 13 bits of the processor generated address while replacing the high order 3 bits by the 7 bit contents of the corresponding MAP register. Addresses in the segment C000-FBFF are considered to be references to Pluribus device registers. The system address for such a reference consists of appending four bits of 1's to the most significant portion of the address.

When the BCP transforms an address, this address, any data, and one of the control operations (read, write, read-modify-write, byte) are communicated from the BCP to the attached BCM through a cable. The control operations will be used by the BCM to generate a bus access on the target bus, generally identical to the bus access on the source bus except for the transformation of the address. Read operations using MAP 3, however, will be

transformed as previously described into read-modify-write accesses on the target bus (where the write data is zero) to allow implementation of multiprocessor locks.

When backwards bus coupling is enabled, the BCP acts as a Master on its bus and simply passes along the 18-bit references generated by the BCM at the other end of the cable.

9.2 BCM:

When a processor accesses a shared resource on a memory or I/O bus, all of the BCPs on the source bus map the initial address and pass it along to the BCM end of the bus coupler. Similarly, when an I/O device accesses a shared resource on a memory bus, all the BCIs on the source bus transmit the initial address to their BCM end. Each BCM then determines if the address sent to it is one to which it can respond. If it is not, the BCM simply ignores the request. If it is, the BCM requests access to its Infibus. When it receives control, the BCM transfers the 2Ø-bit address, any data, and all control signals to its bus and returns any responses received to the originating end of the bus coupler pair. The addresses to which a BCM will respond are determined by the Cable Recognition Switch described below.

There are two important reasons for making the bus coupler perform address discrimination. The first is to reduce hardware contention. If each BCM simply passed all addresses to the containing bus, every processor reference to common memory would be in contention for each memory bus rather than just the single bus on which the referenced memory was located. A similar contention problem would exist for processor references to I/O busses and I/O references to common memory. The second reason for BCM address discrimination is to eliminate multiple responses by the connected busses. Since a bus always responds either positively (by DONE) or

negatively (by QUIT), one DONE and multiple QUITs would result from every access to common memory if no address discrimination was done. The QUITS would, of course, confuse the device that previously requested the access since it would already have taken actions based on the previous DONE. This same problem is the motivation for configuring Pluribus systems so that BCMs connected to different busses recognize disjoint areas of system address space. In general, the addresses recognized by all BCMs connected to the same bus will be identical.

The BCM contains several physical switch registers which must be manually set and a single 16-bit control register which may be referenced under program control. The switch registers along with the number of bits (switches) in each register are indicated below:

Switch Register	Number of Bits
MEMSW (Memory or I/O Bus)	1
BCM CONTROL REGISTER ADDRESS	6
BCM ADDRESS RECOGNITION:	
BASE	8
RELEVANCE	8

The algorithm used by the BCM for address discrimination is as follows: if the 20-bit address it receives is less than FCØØØ, then the high order 6 bits of the address are compared against the high order 6 bits in the BCM ADDRESS RECOGNITION switches. The comparison is satisfied for a particular address bit if either the corresponding RELEVANCE switch is OFF or the RELEVANCE switch is on and the address bit matches the corresponding switch (bit) in BASE. If all 6 high order bits satisfy the comparison, then the 20-bit address is accepted and used to request a bus access.

Typically, the BASE AND RELEVANCE switches will be set to recognize a contiguous portion of system address space. This is done by setting the high order 6 bits of BASE to some starting address and turning off some number of low order switches (within the high order 6) in RELEVANCE. Of course, more complicated memory access patterns can be implemented by other settings of the RELEVANCE switches.

If the 2Ø-bit address passed to the BCM is greater than or equal to FCØØØ and MEMSW is on, the address will not be recognized by the BCM. If the address is greater than or equal to FCØØØ and MEMSW is off, bits 11 and 12 of the address must satisfy the comparison test described above with respect to the two low order bits of the BCM ADDRESS RECOGNITION switches if the 2Ø-bit address is to be recognized and put on the (I/O) bus*.

The BCM contains one internal register, the BCM Control Register. As already indicated in section 4.3 and 6.1, the block of addresses where these control registers can be found is at the beginning of one of the address space segments recognized by the bus to which the BCMs are connected. The precise location of a BCM Control Register is specified by the 6-bit BCM CONTROL REGISTER ADDRESS Switch. The number set in this switch is used as the displacement in words of the BCM Control Register from the starting address of the control register block. To state this more succinctly, the address of each BCM control register is:

*Early models of the bus couplers required all 8 high order bits of the address to match the switch bits for address recognition to occur.

Address Bits	From
14-19	High order 6 bits of BCM ADDRESS RECOGNITION BASE switches
13	Negation of MEMSW
7-12	0
1-6	Contents of BCM CONTROL REGISTER ADDRESS switches.
0	0

The 3 switches (BBC Enable, Foreward Enable, and Reset) which can be set in the BCM control register by a processor have already been discussed in detail in section 7.1.3. It has also been pointed out that the data written to a BCM control register must agree with the high order 13 bits of the hexadecimal constant DE78 if the write is to take effect.

One additional complexity in the BCM arises since the Control Registers for BCMs on a memory bus and those on an I/O bus will differ in one respect; those on a memory bus will share addresses with the memory devices on that bus whereas those on an I/O bus will not share locations with any I/O device. Since devices referencing BCM control registers will expect a single DONE signal upon completion of an access, the BCM works as follows: if MEMSW is on, the BCM does not return a DONE since references to the control register also reference a memory location and the memory device returns the DONE signal. If MEMSW is off, on the other hand, the BCM generates and returns a DONE signal for references made to its control register since there is no other "overlapping devices" that will produce it.

The effect of two devices (BCM and memory) sharing the same address must also be kept in mind when the BCM control register is read or written. For BCMs attached to I/O busses, reading will return 2100 if the attached bus is up or 0 if the bus is in the process of going down due to a power failure-- see section 7.1.1 (of course a QUIT will be returned if the attached bus is completely down). If the BCM shares the address of its control register with a memory module, however, this 2100 or 0 will be Inclusive-Or'ed with the contents of the associated memory word. For this reason and to permit proper operation of the Pluribus system parity mechanism, any read of a BCM control register will normally be preceded by a write of zero to the control register. This will clear any "shadow" memory location but not effect the control register contents. The response to stores at the address of the BCM control register will also depend on whether the address is on an I/O bus or a memory bus in addition to whether or not the high order 13 bits of the data written match the key DE78. If the Key matches, the write will take effect and a DONE will be returned. If the Key does not match, a DONE response will be returned if the BCM control register is on a memory bus and a QUIT response will be returned if the BCM control register is on an I/O bus.

9.3 BCI:

The BCI serves in place of a BCP when coupling an I/O Infibus to a memory Infibus. Its relation to the BCM is identical to that of the BCP with the following three exceptions: (1) no address mapping is performed (devices on I/O busses generate 20-bit addresses), (2) any address less than FC000 (with any data and control signals) is passed directly through the BCI to the BCM, and (3) any address greater than or equal to FC000 is ignored. Devices on an I/O Infibus cannot directly communicate, therefore, with I/O

devices on another I/O Infibus any such communication must be done via common memory. The BCI-BCM bus coupler cannot be used for backwards bus coupling.

10. DEVICES

In section 6, the general information necessary for programming both BBN-developed and LEC devices was discussed. This section provides additional device-dependent information for each of the BBN-developed devices. Similar information for Lockheed devices can be found in the LEC Product literature.

10.1 Pseudo Interrupt Device (PID):

The PID is a priority memory device. The application of this device to the control of processes in a Pluribus system has been described at length in section 5. A Pluribus system may have up to four independent PIDs. As indicated in Figure 5, the PID and Real-Time Clock (RTC) share a 16-byte device register block in the address space of the containing I/O bus. The three registers in this block associated with the PID are the following:

PID WRITE:

When data is written into the PID WRITE register, bits 1 through 7 of the data get a zero appended as bit 0 and the resulting even 8-bit number is stored. Only one copy of any number is retained. When the PID WRITE register is read, the largest number stored in the PID is returned but not deleted.

PID READ:

When the PID READ register is read, the largest number stored in the PID is returned and that number deleted from PID storage. If there is no number stored in the PID, a zero is returned. An attempt to write the PID READ register will result in a QUIT.

PID CLEAR:

When data is written into the PID CLEAR register, bits 1 through 7 of the data set a zero appended as bit 0 and the resulting even 8-bit number is compared with the memory of the PID. If that number is already in PID memory, it is deleted. When the PID CLEAR register is read, the largest number in PID memory is returned but not deleted.

The address of each PID is specified by a two bit switch on the device which selects bits 11 and 12 of the device register block starting address. Bits 0-10 of the device register block starting address are zeros and bit 13-19 are ones. The PID card has a set of lights which display the high order 7 bits of the largest number stored.

10.2 Real-Time Clock (RTC):

The Pluribus has two methods for timing events. Processor Ø on each processor bus can recive an interrupt every 1/6Øth of a second on processor interrupt level 4. This rate is too infrequent for many applications, however, and does not fit into the processor independent structure of the Pluribus. For these reasons another timing device, the RTC, has been developed. The RTC also provides a common time reference for all the processors. The RTC provides access to a clock which is incremented every 100 microseconds and which generates two distinct PID levels periodically, one every 1.6 milliseconds and another every 25.6 milliseconds.

As indicated in Fibure 5, the RTC and PID share a 16-byte device register block in the address space of the containing I/O Infibus. The five registers in this block associated with the RTC

are the following:

CLOCK COUNTER: The 16-bit clock counter. The RTC increments this register every 100 microseconds.

CLOCK PID LEVELS: The high-order byte is the number which will be written to the PID every 25.6 ms. The low-order byte is the number which will be written to the PID every 1.6 ms.

CLOCK READABLE REGISTER 1 - A switch-settable register.
CLOCK READABLE REGISTER 2 - A switch-settable register.
CLOCK READABLE REGISTER 3 - A switch-settable register.

These five registers are all read-only. Attempting to write to them will have no effect.

The RTC has a device timer which will stop the clock if it has not been accessed (i.e. none of its 5 registers have been read) within the past second. This allows the Infibus timeout mechanism to detect and recover from the illegal state where the RTC is the only device putting requests on an I/O Infibus. The clock will also stop on master (bus) reset. Reading any RTC register will start the clock again in these two cases.

The address of each RTC is specified by a two-bit switch on the device which selects bits 11 and 12 of the device register block starting address. There is also a two-bit switch which specifies the address of the PID to be written to in the same way. Normally, these two bits will be identical to the two bits which locate the RTC device register block. In any case, bits 0-10 of the device register block starting address are zeros and bits 13-19 are ones. Two seven-bit switches are also available on the RTC to specify the 1.6 second and 25.6 second PID levels.

10.3 Low Speed Modem Interface (ML):

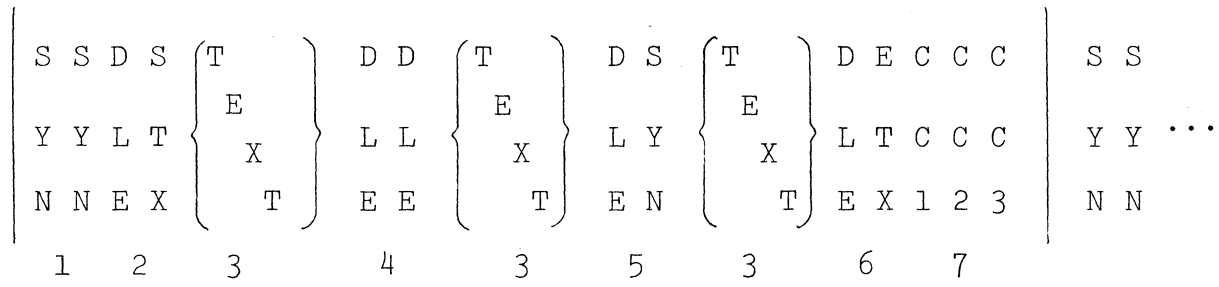
The ML connects the Pluribus to a 303 modem at speeds up to 250kb. The ML will transmit or receive messages of an arbitrary even number of data bytes to or from the 303. The Pluribus need only specify a portion of the message to the ML at any time. This portion of a message is called a buffer and is delimited in core by two addresses provided to the ML by the Pluribus DMA. On transmission, the ML will transmit end of message characters if an end of message flag is associated with a buffer. On reception, a buffer will be read until either the input area provided is full or the end of message characters are detected. Three test options are available on the ML under program control: (1) the ML can be crosspatched to itself so that it takes its transmitted data back in as receive data ignoring the 303 modem, (2) the ML can loop the 303 modem back thus testing both the ML and the 303 modem, and (3) the ML can be forced to send a zero checksum to test the error logic of the ML.

To allow for multiple MLs connected to the same modem, the device timeout feature described in section 7.1.2.2 has been implemented for the ML. Data buffering is provided on the ML card to tolerate delays in servicing of approximately 32 characters on both input and output without loss of data or line utilization. Additional delays of indefinite length are tolerated on output by sending a line protocol idle sequence.

All data on the communication line is organized as 8-bit bytes, and sent low-order bit first. There are four control bytes:

NAME	CODE (Hexadecimal)
SYN	16
DLE	10
STX	02
ETX	83

The protocol on the line looks as follows:



Notes:

1. At least two SYN characters separate adjacent messages. An idle line is filled with SYN characters.

2. The beginning of a message is indicated by the sequence DLE STX. The following character is text. Note that if the RLD card is used, the first bit of text must be zero if the message is not to be interpreted as a special reload message.

3. The text is made up of characters taken from the buffer to be sent. The right half word is sent first, then the left half-word. There are always an even number of text characters in a valid message, otherwise messages are of arbitrary length.

4. When the escape character DLE appears in the text, the hardware inserts an additional DLE.

5. If text is not available to the ML in time, the sequence DLE SYN is sent as an idle protocol until text becomes available.

6. The end of a packet is indicated by the sequence DLE ETX.

7. Each packet is followed by a 24-bit CRC checksum, sent as 3 8-bit characters. The checksum is computed based on all of the characters in the message starting with DLE STX and ending with ETX. A bad checksum will cause the device receiver to report an error.

Note that a DLE is always followed by a DLE, SYN, STX, or ETX. Any other character following a DLE will cause an error on receiving. An error may also be reported if the ML receiver is not serviced quickly enough, that is, within 64 character times of the time that it writes the PID. A receive reset command flushes the input buffer and forces the receive half of the interface to look for character sync. Detected errors also reset the interface to search for character sync, and flag the data as end of packet and error, but do not flush the buffer.

The ML is a DMA device and as such is programmed as described in section 6.2. That section also describes the switches and device independent bits in the ML (DMA) registers. The interpretations of device dependent bits in the registers are indicated below. In each case, the description assumes that a one is read or written.

DEVICE TYPE: The high order byte contains a 1.

RECEIVE STATUS (15) "Loop":

Write: Cause the 303 modem to send back the transmission of the ML to the receive portion of the ML. The receive portion of the ML should be initialized before the transmit portion so that no data is lost.

Read: The ML is looped.

RECEIVE STATUS (14) "Crosspatch":

Write: Cause the ML to take back its transmitted data into the receive portion ignoring the 303 modem. The receive portion of the ML should be initialized before the transmit portion so that no data is lost.

Read: The ML is cross-patched.

RECEIVE STATUS (13) "Active":

Read: The ML receive portion is either waiting for or transferring a buffer from the 303 modem to memory.

TRANSMIT STATUS (15) "Device Timeout":

Read: A one second timer has deactivated the ML. If the transmit status word has not been written for one second, all activity of the ML is aborted. All ML circuits which communicate with the 303 modem are deactivated. The ML will become usable again when the transmit status word is written.

TRANSMIT STATUS (14) "Zero Checksum":

Write: Generate a zero checksum for this message.

Read: A zero checksum will be generated for this message.

TRANSMIT STATUS (13) "Active":

Read: A buffer is being transmitted.

10.4 Local Host Interface (HLC):

The Local Host module provides an interface between the Pluribus and another computer (called a Host) according to the hardware specification for IMP to Host connections described in the BBN report, "Specifications for the Interconnection of a Host and an IMP" [8]. This is a general purpose asynchronous serial interface. The Local Host module can perform block transfers of data in either direction between the Host and the PLURIBUS. A data block can be either read or written as a number of separate buffers if required. Transfer of the last buffer will have an associated end of data block flag. Padding is provided by the HLC receiver at the end of data blocks to account for word length mismatch between the Pluribus and the attached Host. Two padding options are available: (1) a 1 followed by 0's as described in Report No. 1822 or (2) all zeros. The choice is fixed by hardware jumpers on the interface. Another set of jumpers permits the Pluribus and Host ready lines to be permanently disabled (ignored). The Local Host module can be programmed to be looped. In this state, all data transmitted from the transmit half of the HLC is returned to the receive half of the HLC. This mode of operations is convenient for hardware and software debugging. To allow for multiple HLCs connected to the same Host, the device timeout feature described in section 7.1.2.2 has been implemented for the HLC.

The HLC is a DMA device and as such is programmed as described in section 6.2. That section also describes the switches and device independent bits in the HLC (DMA) registers. The interpretations of device dependent bits in the registers are indicated below. In each case, the description assumes that a one is read or written.

DEVICE TYPE: The high order byte contains a 2.

RECEIVE STATUS (14) "Loop":

Write: Connect the receive portion of the HLC to the transmit portion so that data transmitted by the HLC will be returned to the receive portion of the HLC. To initiate the transmission both RECEIVE END AND TRANSMIT END must be written. RECEIVE END should be written before TRANSMIT END so that no data is lost. When the HLC is looped, the HOST READY indicator is the same as the Pluribus READY indicator.

Read: The HLC is performing in looped mode.

RECEIVE STATUS (13) "Active":

Read: The receive portion of the HLC is active receiving a data block.

RECEIVE STATUS (12) "Host Ready":

Read: A one indicates that the Host has set its ready indicator.

RECEIVE STATUS (11):

Read: There was an error in the last buffer received from the Host. No end of message terminated the data block. This bit will be set if the Host Ready signal went away and returned during the previous transfer. Note that the error bit in the RECEIVE END register will also be set.

RECEIVE STATUS (10):

Read: Same as RECEIVE STATUS (11) above except if this bit is set, an end of message indication did terminate the data block.

TRANSMIT STATUS (14) "Loop":

Read: The HLC is performing in looped mode.

TRANSMIT STATUS (13) "Active":

Read: The transmit portion of the HLC is active transmitting a data block.

TRANSMIT STATUS (12) "Pluribus Ready"

Write: Writing a one sets the Pluribus ready indicator. Writing a zero clears the Pluribus ready indicator. This indicator will also be cleared if neither the transmit or receive status words has been written in the last second in order to implement the previously described device timeout feature.

Read: The Pluribus ready indicator is set.

10.5 Checksum/Block Transfer Device (CBT):

The Checksum/Block Transfer Device performs one of three operations on a source data buffer: (1) it calculates a 16-bit checksum on the data (2) it transmits the data words from the source buffer to a destination buffer, or (3) it does both (1) and (2) simultaneously. Aside from providing a convenient way to move data around with a Pluribus system, this device provides a key service for the system reliability software (see section 7.2). To the DMA, the device appears as two separate sections - source (transmit) and destination (receive) which deal with the DMA data transfer independently but are linked closely together within the device. Only the source interrupt and status, however, are used. Checksum calculation is performed serially, low order bit first with provision for either reinitializing or continuing the computation when a new data block is specified. Either an IBM CRC 16-bit checksum or a CCITT 16-bit checksum may be calculated. The choice is switch-selectable. The generator polynomials for these checksums are as follows: IBM: $X^{16} + X^{15} + X^2 + 1$ and CCITT: $X^{16} + X^{12} + X^5 + 1$. Transfer rate is limited only by bus access time when no checksum is being computed and by the slower of bus access or checksum computation when a checksum is being computed. Checksum computation time is approximately 1.3 microsecond per 16-bit word.

The CBT is a DMA device and as such is programmed as described in section 6.2. That section also describes the switches and device independent bits in the CBT (DMA) registers. The interpretations of device dependent bits in the registers are indicated below. In each case the description assumes that a one is read or written.

DEVICE TYPE: The high order byte contains a 3. Switch (bit) 0 selects a CCITT checksum (off) or an IBM CRC16 checksum (on).

TRANSMIT (SOURCE) END (15):

Write: This is the last buffer of the block.

TRANSMIT (SOURCE) END (0):

Write: Clear the checksum accumulator register. Writing a zero indicates that the previous checksum should be preserved, e.g. when checking a multi-buffer block.

Read: Error

TRANSMIT (SOURCE) STATUS (15) "Check":

Write: Calculate checksum. Changing this bit while an operation is in progress will cause an interrupt and a device reset.

Reset: Checksum being calculated.

TRANSMIT (SOURCE) STATUS (14) "Transfer":

Write: Move data from source buffer to destination buffer. Writing this bit while an operation is in progress will cause an interrupt and a device reset.

Read: Data being moved from source buffer to destination buffer.

TRANSMIT (SOURCE) STATUS (13) "Active":

Read: CBT operation in progress.

TRANSMIT (SOURCE) STATUS (12) "EOB Destination":

Read: Interrupt requested since the destination buffer is too small for source buffer. CBT has suspended activity until new destination buffer addresses are supplied for the remainder of data. No data is lost. The error bit is also set.

TRANSMIT (SOURCE) STATUS (11) "NOP":

Read: Interrupt requested since CBT initiated action but registers indicate that there is nothing to be done. The error bit is also set.

TRANSMIT (SOURCE) STATUS (10) "Last":

Read: TRANSMIT (SOURCE) END (15) was set when this buffer was written.

TRANSMIT (SOURCE) STATUS (9) "Destination QUIT":

Read: The receive (destination) portion of the device received a QUIT during the previous operation. The error bit is also set.

DEVICE DEPENDENT DMA REGISTER - The 16-bit checksum is accumulated here. This register may be initialized prior to the start of a checksum computation. Writing this register during a check operation will cause an erroneous checksum to be calculated.

10.6 External Reload Device (RLD):

The Reload card (RDD) monitors the input data from up to eight modem interfaces. When the RLD observes a command, it decodes the command as a 20-bit system address, a 16-bit data word, and a 16-bit CRC16 checksum. This single card device is not processor controllable, but is controllable by external signals arriving over the normal communication lines. The purpose of the RLD is to change, control, or restart the Pluribus system from a remote site without

on-site supervision. The RLD resides on an I/O bus, thus the RLD can modify common memory busses and access processor busses by backwards bus coupling as well as access devices on its own bus.

Whenever a message is received over a communication line, the RLD checks the first bit (the least significant bit of the first 16-bit word). If this bit is one, the RLD determines that a sequence of RLD commands is arriving over that communication line and ceases to monitor the other 7 modem interfaces until all commands in the incoming message have been processed. Except for the first bit of the first command in the message, each of the remaining bits in each command are doubled to increase the uniqueness of the reload packet and to guarantee that the DLE character will not occur in the reload data stream. The format of an arriving RLD message is indicated below:

	DS LT EX	1 ADDRESS	A DDRESS	A DDRESS	A DDRESS	A DDRESS	1 1	DATA	DATA	DATA	DATA	CHECKSUM	CHECKSUM	CHECKSUM	CHECKSUM	ADDRESS ...	CHECKSUM	DLE	ET EX
Bits:		0 ↓ 3	4 ↓ 7	8 ↓ 11	12 ↓ 15	16 ↓ 19		0 ↓ 3	4 ↓ 7	8 ↓ 11	12 ↓ 15	0 ↓ 3	4 ↓ 7	8 ↓ 11	12 ↓ 15	0 ↓ 3	12 ↓ 15		

It should be noted that:

1) The low-order two bits of the first word of the first command must be 01. The low order two bits of the first word of sub-

sequent commands in the message must be 00.

2) An eight-bit padding byte follows the high order address bits. This byte can contain any non-zero "doubled" four-bit pattern. The pattern can be set by jumpers on the card. (Note that four ones are shown in the figure above.)

3) An arbitrary number of commands may contained in an RLD message.

4) The 16-bit checksum on the address and data bits in the IBM CRC-16 checksum with generator polynomial $X^{16} + X^{15} + X^2 + 1$.

For each command in the message, the RLD device stores the incoming data word at the specified system address. This process is repeated until either a bad checksum is detected, bad padding is detected, non-doubled data is detected, a delayed bus access occurs, or the RLD device times out after one second of inactivity. In each of these cases, the RLD device releases the communication line and is available to service one of the other modem interfaces.

There are 3 lights on the RLD device which provide a visual indication of the device operation. One light is on from the time that the device is first activated until the bus containing the device is reset. The second is on for the duration of a single RLD command sequence. The third indicator is briefly lit by completion of each bus access.

10.7 Synchronous Line Interface (SLI):

The SLI provides a simple synchronous full-duplex interface to a wide variety of modems. In contrast to the other interfaces previously described, the SLI is a single passive device and does not use either the DMA facility or the PID. To guarantee that

neither data not line bandwidth will be lost, the processors must poll each SLI in the system faster than the byte rate being used.

Each physical SLI card provides interfaces for two independent lines. The allocation of the 8 words in the device register block is given below. The location of the register block is fixed by jumpers on the card.

Register 1: Device Type - Modem 1
Register 2: Status - Modem 1
Register 3: Control - Modem 1
Register 4: Data - Modem 1
Register 5: Device Type - Modem 2
Register 6: Status - Modem 2
Register 7: Control - Modem 2
Register 8: Data - Modem 2

The Device Type and Status words are read only registers whereas the Control and Data words are read-write registers. The interpretation of bits in each of these four registers are given below. In each case, the interpretation assumes that the particular bit is one unless otherwise stated.

DEVICE TYPE (8-15): The high-order byte of the Device Type register contains a 4.

DEVICE TYPE (6-7): These bits are set by switches on the SLI card and indicate information concerning the speed of the modem to which the SLI is connected.

Bits (6,7)	Speed
00	Under 2.5K bits/sec
01	Under 5K bits/sec
10	Under 10K bits/sec
11	19.2K bits/sec and over.

STATUS (15): The transmitter buffer is empty. The next character may be written to the Data register. (It is assumed that Clear to Send status (10) - has previously been found to be on.)

STATUS (14): A SYNC character has been transmitted. The program was too slow in responding to STATUS (15) above and in the absence of new data a SYNC character was transmitted. This bit remains set until the next data character starts being transmitted.

STATUS (10): Clear to Send. This is a signal received from the modem. In general, this indicator signifies that the modem is ready to transmit data. Refer to the modem literature for more detail.

STATUS (8): Data Set Ready. This is a signal received from the modem. In general, it indicates to the SLI that its modem is not in a test mode and its power is on. Refer to the modem literature for more detail.

STATUS (0-7): After a Receiver Reset (CONTROL bit 0) this half of the Status Register will monitor the input data stream, that is, bits will be detoured here as well as going to the Data register. This will continue until a zero has propagated to STATUS 0 at which point these 8 bits will no longer change. A subsequent receiver reset will cause this first even character search mode to start again. It is expected that this feature will be used to handle the case of devices transmitting to the SLI which employ different sync characters. The first even character received can by mutual agreement be the sync character that will be recognized by the interface hardware (see DATA (8) and DATA (9) below).

CONTROL (10): Request to Send. This signal is passed directly to the modem. In general, it indicates to the modem that the SLI is ready to transmit data. The modem will normally respond by setting Clear to Send STATUS (10). Refer to the modem literature for more detail.

CONTROL (9): Loop Test. Loop the SLI output back into the SLI input. This feature allows the SLI to test itself without a modem.

CONTROL (7): Transmit and recieve in 7-bit plus parity mode. This bit will be set by the program when communicating with an ASCII terminal. When writing to the data register bits 0-6 will be accepted and the SLI hardware will add the correct bit 7 to create odd parity in the 8-bit character transmitted. Data words received will be checked for odd parity (see DATA (9) below) but bit 7 of the data byte read will be zero. For communication with EBCDIC terminals, CONTROL (7) is cleared. In this case parity is neither generated nor checked. The 8-bit character transmitted is the 8-bit byte written to the data register.

CONTROL (0): Receiver Reset. Clear Received Parity Error - DATA (9), Receiver Overrun Error - DATA (8), Sync Received - DATA (14), and Data Ready - DATA (15). As described above, writing this bit also initiates sync character search mode and initializes STATUS (0-7) to all ones.

In contract to the DMA devices previously described, the input and output halves of the SLI share a single address for the two (read and write) DATA registers. The proper SLI internal register is referenced when the SLI is accessed as described below.

DATA (15)

Read: Data Ready. Finding this bit set segnals that a new character is in bits 0-7 of the DATA register. If DATA (15) is zero, then no change has occurred to bits 0-7 of the DATA register since the last time the DATA register was read. Reading the DATA register sets bit 15 if it was one. In normal operation, the DATA register is read more frequently than the byte rate, bit 15 is tested, and bits 0-7 extracted or ignored as appropriate.

DATA (14)

Read: Sync Received. This bit will be one from the time that a sync character is detected until a non-sync character is detected. Although available, this information will generally not be used by most programs.

DATA (9)

Read: A parity error has been detected. This bit is cleared only by Receiver Reset. It is never set unless CONTROL (7) has been set to one. Parity checking is enabled when data mode is entered, that is, when the first non-sync character after two successive sync characters arrives.

Write: Store Transmit Sync. Route DATA (0-7) to a special holding register (rather than transmit it). This character will become the transmitted sync character, i.e. it will be transmitted whenever the last bit of a character has been sent but no new data character has been written to DATA (0-7). This register remains unchanged until rewritten.

DATA (8)

Read: The incoming data stream has not been serviced quickly enough and a character has been lost. Since the input is double buffered, two byte times must have elapsed since Data Ready was last set for this to occur. This bit is cleared only by Receiver Reset.

Write: Store Receive Sync. Route DATA (0-7) to a second special holding register (rather than transmit it). This character will become the receive sync character, that is, this character will be compared to the received bit stream to achieve character synchronization. Data mode will be entered after at least two adjacent sync characters have been received. This register will remain unchanged until rewritten.

DATA (0-7)

Read: Input Data Byte. This is the data to be extracted when Data Ready is Set.

Write: Output Data Byte. This is the location to which the next 8-bit byte should be written when Transmit Buffer Empty is found set. This register is not protected against premature writing and no indication is provided if it is written when Transmit Buffer Empty is zero. If this happens, the character previously written will have been lost without being transmitted. Each receive and transmit portion of the SLI device is actually double buffered in addition to the serial-to-parallel shift register in the card. This extra buffering implies that the programmer actually has longer than a single character time in which to service the device. In addition, the programmer should also be aware that this buffering

has other implications since the contents of the status and data registers are not synchronized. A status indicator can not be associated with the data byte currently available in the data register.

The SLI device can be used with either switched or dedicated channels. The Data Set Ready, Data Terminal Ready, and Carrier Detect signals will be useful primarily in switched applications. They can all be strapped to "true" values for unswitched operation. If the SLI is used with a full-duplex channel (i.e. modem and circuit) the Request to Send and Clear to Send Signals could also be strapped "true". They are included to allow the option of half-duplex operation.

REFERENCES

1. Heart, F. E., Ornstein, S. M., Crowther, W. R., and Barker, W.B., "A New Minicomputer Multiprocessor for the ARPA Network," Proceedings of the 1973 AFIPS National Computer Conference, Vol. 42, pp. 529-537.

2. Lockheed Electronics Company, SUE Computer Handbook

3. Lockheed Electronics Company, SUE Computer System, General System Bulletin G2, included in Pluribus Document 3.

4. Lockheed Electronics Company, SUE Processor Instruction Set, General System Bulletin G3, included in Pluribus Document 4.

5. Lockheed Electronics Company, SUE Infibus Interface, General System Bulletin G4, included in Pluribus Document 6.

6. Bolt Beranek and Newman Inc., Pluribus Document 3: Configurator.

7. Ornstein, S.M., Crowther, W.R., Kraley, M.F., Bressler, R.D., Michel, A., and Heart, F.E., "Pluribus - a Reliable Multiprocessor," to appear in the Proceedings of the 1975 AFIPS National Computer Conference.

8. Bolt Beranek and Newman Inc., "Specifications for the Interconnection of a Host and an IMP," BBN Report No. 1822.

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PART 3: GLOSSARY

GLOSSARY

Update History:

Originally written by M.F. Kraley, February 1975.

60 Hz. interrupt - a classical interrupt occurring at the power line frequency on level 4, device number 1.

abort - a QUIT.

access time - time from the initiation of the request (rise of STRB) to the presentation or acknowledgment of data (rise of DONE).

active - said of a DMA device while it is transferring data: from the writing of the end pointer to the setting of the PID level.

active I/O device - an I/O device which indicates its need for service directly, usually either by classical or pseudo interrupt; cf. passive I/O device.

address halt - a feature of the control panel which halts a processor when a selected address is accessed on the bus.

address recognition - the process in which a module checks the Infibus address lines for an address which is in the range of those which pertain to that module.

address space - the set of locations accessible to (addressable by) a device; cf. memory space, I/O space, system address space, processor address space, etc.

amputate - to disconnect a bus (usually a processor bus) from the rest of the system by turning off the forward enable bit in all bus couplers coming from that bus.

ALD - a LEC card which implements the AutoLoaD function.

arbitration - the act of choosing the next prospective user of a resource.

ARPA Network - a national network of heterogeneous computers linked to facilitate research; the original design environment for the Pluribus.

asynchronous - not necessarily occurring at a certain time or at fixed time intervals.

asynchronous line - a serial communications line where the receiver derives timing information from the initial transition of a character's start bit; characters are sent individually, at arbitrary times, bounded by start and stop bits.

attention - a classical interrupt on level 1, device number FF80, caused by pushing the "attn" button on the control panel.

autoload - a LEC module which contains some read only memory programmed to do loading from any of a number of I/O devices; when commanded by a bus signal, the autoload will initiate a classical interrupt, having first set the vector address to point to the ROM.

auto restart - see power recovery.

auxiliary I/O space - the portion of I/O space from FC000 through FDFFF.

auxiliary processor - a processor which is not number 0 and thus does not handle classical interrupts.

AXD parity - a scheme wherein the parity bit(s) are derived from both the address and data; specifically, the parity bit of each byte is the exclusive-OR of the odd parity of the data and the odd parity of the byte address of that byte.

backwards bus coupling - the process by which a master on a common (usually I/O or M/I) bus can access a slave on another bus (usually a processor bus); used by processors to access other processors' address space.

bandwidth - the rate at which information may be transferred or processed.

BBC - Backwards Bus Coupling.

BBC enable bit - bit 2 of the bus coupler control register; controls whether that coupler is selected for BBC.

BBC map - a register in the BCM which specifies the high-order address bits of a BBC reference.

BBC window - the four word region of system address space through which BBC references are performed.

BBN - Bolt Beranek and Newman Inc.; the developer of Pluribus.

BCI - Bus Coupler I/O end; the card which forms the I/O end of an I/O-to-memory bus coupler.

BCM - Bus Coupler Memory end; the card which forms the I/O end of a processor-to-I/O coupler and the memory end of processor-to-memory and I/O-to-memory bus couplers.

BCP - Bus Coupler Processor end; the card which forms the processor end of processor-to- memory and processor-to-I/O bus couplers.

BCU - Bus Control Unit; a LEC card which performs bus supervisory functions; it is chiefly responsible for arbitrating the use of the bus, but also assists in classical interrupts and other specialized functions.

BDR - Bus Driver/Receiver: a custom IC used in both LEC and BBN boards to interface with the Infibus.

begin pointer - in a DMA device, the address of the first word of the buffer.

bezel - the decorative front of a bus unit which also contains an air filter.

block transfer - the act of copying the contents of a series of contiguous memory locations to another place.

buddy - the other processor(s) on the same bus.

buffer - a series of contiguous memory locations which holds a block of data.

bus - usually an abbreviation for Infibus.

bus arbiter - a BCU.

bus controller - a BCU.

bus coupler - a module which allows transactions on one bus to be transformed into transactions on another bus, depending upon address; composed of a BCM, either a BCP or BCI, and connecting cables; performs other special features such as parity generation and checking, mapping, power isolation, and amputation.

bus extender - a LEC module which allows one logical bus to span more than one bus unit; the extended bus looks just like one long bus; it consists of two cards, BXD and BXR, and two connecting cables.

bus timer - usually refers to the reset timer.

bus unit - the basic mechanical module of the Pluribus; contains various combinations of Infibusses and power supplies, and has integral cooling.

BXD - Bus Extender Driver; the card which forms part of the bus extender; plugs into the same bus as the BCU.

BXR - Bus Extender Receiver; the card which forms part of the bus extender; plugs into the bus which does not have a BCU.

byte - 8 bits; two bytes to a word.

cable - an assembly which electrically connects two or more modules and/or external equipment; each type has a four letter designation.

card - a logic board which plugs into the Infibus; each type has a three letter designation.

CBT - a BBN card which forms part of a Checksum-Block Transfer module.

CCITT checksum - a 16 bit checksum computed with polynomial x**16 + x**12 + x**5 + x**0.

CCP - Communications and Control Processor, a Pluribus application which involves the collection, limited processing and routing of seismic data.

central processor - the number 0 processor electrically connected to the bus arbiter which handles all classical interrupts.

checksum - a number of bits associated with a block of data computed via a fixed function from the data; the implicit redundancy can then be used to detect changes in the data.

checksum-block transfer - a BBN module which allows the computation of a cyclic checksum and/or the copying of a block of memory; consists of a DMA and a CBT.

classical interrupt - the diversion of the control stream of a processor in response to an external event; the device number of the interrupting device, status and program counter at the time of interruption are saved and the processor jumps indirect through a fixed location; also refers to the bus transaction which causes the interrupt.

classical parity - the parity scheme wherein the source generates on writes and the source checks on reads.

clock - usually refers to RTC.

CMB - the LEC printed circuit board which forms the actual Infibus; holds the edge connectors for the cards.

common bus - a bus which is not a processor bus: a memory, I/O, or M/I bus.

common memory - that memory which can be accessed by all processors, that is, all memory on memory or M/I busses.

configuration - the process by which a group of Pluribus modules are selected and an arrangement designed to create a machine for a particular application.

consensus - the agreement between processors that to take a particular action would be in their common interest; also refers to the process by which agreement is reached.

console - usually refers to the control panel.

contention - the situation where multiple users are attempting to simultaneously use a resource; this usually causes delay.

continuous read/write - a feature of the control panel which when set, repeatedly performs the access requested by depressing the "read" or "write" buttons; located on the rear of the control panel.

control panel - a LEC module which allows manual reading and writing of addresses, typically memory locations, processor registers, and starting and stopping of processors, and other special functions; consists of two cards, PCB and PBI, connected by a DIP connector cable, and a front panel, SWB, which connects to the other cards via three ribbon cables.

control register - a location associated with a module whose bits correspond to program settable functions; in a processor, register 15; in a serial or parallel interface, the address of the device + 6; in a bus coupler, set by the jumpers on the BCM.

cooling module - the external shell of a bus unit which provides mechanical support for the Infibus chassis, fan pack, and bezel, and airflow isolation and deflection.

CPA - a LEC card which forms part of the processor.

CPB - an early LEC card which used to form part of the processor; superseded by CPC.

CPC - a LEC card which forms part of the processor.

CPU - central processor; more generally, but incorrectly, a processor.

CRC-16 checksum - a 16 bit checksum computed with polynomial x**16 + x**15 + x**2 + x**0.

cycle time - the time from the beginning of a request until the device has completed all activity related to that request and is ready to start or accept another; usually longer than access time.

cyclic checksum - a checksum computed by dividing the data by a specific polynomial and taking the remainder.

D-cable - a cable which connects a card plugged into an Infibus with the fantail.

DBAL - Dual Bus Access Logic, a custom IC that contains much of the logic necessary to be a bus master.

DDT - a program which allows the user to inspect and change memory locations and processor registers, start and stop processors, copy memory, field traps and other useful things; in many ways, can be thought of as a simple executive, providing an environment for user programs.

deadlock - a state in which two (or more) processes (hardware or software) are each waiting for a resource held by the other; each now waits indefinitely for the other's resource to become available.

device - usually a module that performs I/O functions; sometimes refers just to DMA devices.

device dependent - a register or bit whose interpretation or function is determined by the particular module with which it is associated.

device independent - a register or bit with a common interpretation or function over a range of different module types.

device register block - the eight word segment of address space which is associated with a DMA device.

device type - a number indicating the type of the associated module; usually program readable in the low order byte of the first register of the device.

device number - a number associated with each device which causes classical interrupts; when servicing an interrupt, this number can be read from the first word

of the interrupt vector, indicating which device caused the interrupt.

DMA - Direct Memory Access; a BBN card which performs the bus interaction and pointer management for I/O devices.

DMA device - an I/O device which uses a DMA; it transacts directly with memory with data in buffers.

DONE - an Infibus signal that indicates successful completion of a bus access cycle; also serves as the strobe for data in a read access.

doubled cable - a cable which connects the two parts of a doubled interface with the fantail.

doubled interface - an interface which, for reliability considerations, consists of two modules on different busses, connected such that either one can serve the external equipment.

elastic buffer - a buffer which allows input and output to proceed asynchronously, at different rates.

end pointer - in a DMA device, the address of the last word of a buffer.

executive core - usually refers to locations 0-5E; the area in which interrupt, QUIT, and ILLOP information is stored.

EXY - Eight X and Y; a LEC card which forms part of the memory; contains the core stack itself.

F-cable - a cable which connects external equipment to the fantail.

fan pack - a chassis containing six fans that provide the cooling for each bus unit.

fantail - a panel which contains connectors for cables from external equipment which interface to internal cables; used to facilitate the reconnection of external equipment.

feedback parity - the parity scheme wherein the destination generates and the source checks parity on all transfers.

flop - flip-flop.

force reload - a scheme by which memory locations may be loaded, processors started, etc., via special,, heavily

passworded messages on modem lines; used for remote start-up of machines.

forward enable bit - bit 1 of the bus coupler control register; when cleared, prevents all forward accesses through that coupler, thus amputating the bus associated with the BCP or BCI of that coupler.

F-stick - to map an address in I/O space via the implicit fixed mapping.

full duplex - a communications path wherein transmission can take place in both directions simultaneously.

ground modem - not a satellite modem.

half duplex - a communications path wherein transmission can take place in either direction, but not both simultaneously.

halt(ed) - a state of the processor wherein instructions are not being executed, interrupts cannot be honored, and the registers are externally accessible.

hex - abbreviation for hexadecimal, base 16.

high speed modem - a BBN module which interfaces to a Bell 306 modem at speeds up to 1.5 Mbaud; consists of MHX, MHR, and DMA.

HLC - Local Compatible Host; a BBN card that forms part of a Host interface.

HIT - the name of the general Pluribus system test program.

Host - a computer which provides and uses the actual network services; connected into the network via an IMP.

Host interface - a BBN module which interfaces to a local Host; comprised of a DMA and HLC.

hot code - frequently executed code which is located in local memory.

IBM - four card interconnect module.

IBM checksum - CRC-16 checksum.

ICM - three card interconnect module.

IDM - two card interconnect module.

I-cable - a cable which connects two internal cards.

idle - a state of the processor wherein no instructions are being executed, registers are not externally accessible, but interrupts may be honored.

illegal operation - trap caused by attempted execution of an instruction not in the repertoire of the processor.

ILLOP - illegal operation.

ILLOP vector - the four word block holding information pertinent to the current ILLOP; starts at 20 for processor 0, 30 for processor 1; contents are: illegal instruction, status, program counter, address of service routine.

IMP - Interface Message Processor; the node computer of the ARPA Network, which performs the basic packet-switching functions.

Infibus - the bus which physically and electrically connects the cards of a Pluribus system.

interface - a module which allows access and information flow to and from external equipment.

interrupt - usually a classical interrupt.

interrupt vector - the four word block holding information pertinent to a given interrupt level; for levels 1-4, starts at locations 0,8,10,18 respectively; contents are device number, status, program counter, address of service routine.

I/O bus - a bus which contains primarily I/O devices.

I/O space - the part of system address space from FC000 to FFBFF; the area which may be accessed via fixed mapping from processor address space; also refers to the corresponding section of processor address space (C000-FBFF)

isochronous line - a serial communications scheme wherein bit timing is derived from a separate clock line, but characters may be sent at arbitrary intervals and are bounded by start and stop bits.

jiffy - a 60 Hz. interrupt or 1/60th of a second.

JIG - the name of the bus coupler stand-alone test program.

K - 1024(decimal).

key bits - address bits 16 and 17, asserted on processor references according to the contents of a two bit register set by the SKEY instruction; used to differentiate among the various processors on a bus.

LEC - Lockheed Electronics Company.

level 5 interrupt - an ILLOP.

level 6 interrupt - a QUIT.

local Host - a Host interconnected via a bit serial, handshook interface, usually over distances of less than 30 feet.

local memory - memory on a processor bus, as opposed to common memory.

lock - a data structure (usually a single word) used to interlock processes; also refers to the act of reading a lock with a read-clear cycle.

Lockheed Electronics Company - the manufacturer of several Pluribus parts.

low speed modem interface - a BBN module which interfaces to a Bell 303 modem at speeds up to 250 Kbaud; consists of a MLX, MLR, and a DMA.

map value - the 7 bit number that determines which 4K page of system address space is referred to by accesses in the associated map segment.

map segment - one of the four 4K regions of processor address space through which accesses are made to common memory.

mapping - the act of transforming an address in one address space to that in another.

master - the participant in a bus transaction which initiates the access; e.g. the processor, when it is accessing memory.

memory - a LEC module, either 4K or 8K by either 16 or 18 bits of random access core memory, consisting of three cards: TAG, SID, and EXY; also refers to a more generic collection of the above.

memory bus - a bus which contains primarily common memory.

memory space - the part of system address space from 0 to FBFFF.

message - the unit of data communicated between Hosts; messages are broken up by IMPs into one or more packets for transmission in the subnetwork.

M/I bus - a common bus which contains both Memory and I/O.

MHR - a BBN card which forms the Receive half of a High-speed ground Modem interface.

MHX - a BBN card which forms the transmit half of a High-speed ground Modem interface.

MLR - a BBN card which forms the Receive half of a Low-speed ground Modem interface.

MLX - a BBN card which forms the transmit half of a Low-speed ground Modem interface.

modem - a piece of external equipment which converts digital signals from the computer to analog signals for communication and vice versa; also refers to modem interface.

modem interface - the module which interfaces to a high speed synchronous modem, either ground or satellite, low or high speed.

module - a unit consisting of one or more cards which performs a unified function.

MSR - the BBN card which forms the Receive part of the Satellite Modem interface.

MST - the BBN card which performs the Timing functions of the Satellite Modem interface.

MSX - the BBN card which forms the transmit part of the Satellite Modem interface.

multiprocessor - a system which contains several tightly coupled processors with some common resources.

Multiwire - a technology for making cards, midway between printed circuit and wire wrap in the dimensions of cost and difficulty; consists of a printed circuit card which carries power and ground, covered by a sticky insulating layer, in which insulated wires are laid to form the signal paths.

P-cable - a cable which carries primarily power.

packet - the unit of data communicated between IMPs on modem lines; several packets may form a message; usually on the order of one or two thousand bits.

packet-switching - a communications scheme in which packets of data from many sources are forwarded to many destinations along the same line, multiplexing the use of the line at a high rate.

page - a 4K region of common memory, or more generally, system address space.

PAR - I/O PARity; a BBN card which generates parity for references to I/O devices.

parallel interface - a LEC module that can interface up to 20 parallel bits of information; can be polled or use classical interrupts; used primarily as the paper tape reader interface; card type PPB.

parity - the exclusive OR of a collection of data and /or address bits; also refers to schemes which detect changes by generating and later checking the parity of a collection of bits.

parity memory - memory which is 18 bits wide, allowing a parity bit to be stored for each byte.

passive I/O device - a device which must be polled, does not interrupt; cf. active I/O device.

password - a specific combination of data bits which must be written in order for an action to take place; used for reliability considerations.

PBI - Panel Bus Interface; a LEC card which forms part of a control panel.

PCB - Panel Control Board; a LEC card which forms part of a control panel.

PCD - PreCeDence passer; a BBN card which serves only to pass the precedence pulse by an empty slot; used for debugging.

PDU - Power Distribution Unit; usually refers to a BBN module which accepts site power and distributes it, with appropriate switches, circuit breakers and indicators; also refers to a LEC module which provides two key switches, one for power, the other for processor selection.

PID - Pseudo Interrupt Device, a BBN module which serves as a hardware pending task queue.

PID level - the number that a device or a processor writes to the PID to signify that the associated task should be run.

Pluribus - a line of modular, reliable, multiprocessor/minicomputer systems produced by BBN.

poll - the act of periodically checking a device to see if some event has occurred, as opposed to the device doing its own notification when a change in status occurs.

power fail - a classical interrupt which occurs 2.5 ms. before bus operations are ceased preparatory to complete power loss; occurs on level 4, device number 2.

power restart - a classical interrupt which occurs on restoration of local bus power on level 4, device number 4.

power sense - an Infibus signal that indicates the condition of bus power, gives advance notice of a power failure; also refers to circuitry in the bus coupler that checks the status of power at each end of the coupler, allowing one end to disregard signals coming from a card with inadequate power.

power supply - a LEC module which supplies Infibus logic power and a 60 Hz. signal; comes in two styles: internal (plug-in, 5951) which takes up 8 of the 24 slots of an Infibus, and external (stand-alone, 5952) which requires its own bus unit.

PPB - Peripheral Parallel Buffer; parallel interface.

precedence pulse - an Infibus signal which is daisy-chained between cards; used to resolve priority for the selection of the next bus master.

primary I/O space - the portion of I/O space from FE000 to FFBFF.

printed circuit - a technology for fabricating cards which involves etching away copper-clad epoxy boards to form the signal paths.

private memory - local memory.

processor - a LEC module which executes instructions; consists of two cards, CPA and either CPB or CPC; three

microcode versions exist: standard, business, and scientific.

processor address space - the address space seen by an individual processor; 32K words long.

processor bus - a bus which contains processors and (usually) local memory.

processor bus address space - the aggregate of the four potential processor address spaces on a processor bus; 128 K words long.

pseudo interrupt - the act of writing a number to the PID to indicate that a task associated with that number should be performed.

PSB - Peripheral Serial Buffer; serial interface.

QUIT - an Infibus signal that indicates abnormal completion; e.g. non-existent device, malfunctioning device, parity error, etc.; also refers to the trap that is taken when a processor-initiated access results in a QUIT.

QUIT timer - the timer on the bus arbiter which regulates how long the bus will wait for a DONE before deciding that the intended slave will not respond and thus should issue a QUIT, terminating the access; these timers have different values on different bus types.

QUIT vector - the four word block of memory which records information pertinent to the most recent processor-initiated QUIT; contents are: address referenced, status, program counter, and address of service routine; located at 28 for processor 0, at 38 for processor 1.

rack - the unit which houses bus units; up to five bus units, a PDU, and a fantail may be mounted in one rack.

read - an access in which data is transferred from slave to master.

read-clear - a read-modify-write access in which the written data is zero.

read-modify-write - an access in which data is read from memory and then (potentially) different data is written back to the same address, all within one memory cycle.

real time clock - RTC.

reload card - RLD.

remote power fail - a classical interrupt which indicates that a common bus's power is failing and about 2.5 ms. of usable power remains; occurs on level 1, device number 1.

remote reset - the low-order bit of a bus coupler's control register; when cleared, causes a reset to occur on the bus which the BCP is plugged into.

reset timer - see bus timer.

resource - a part of a system needed by more than one of the parallel users and therefore a possible source of contention.

ribbon cable - a multiconductor cable made of several parallel wires bonded together in a flat shape.

run - a state of the processor in which instructions are being executed, interrupts may be honored, and registers are not externally accessible.

ribbon - the part of an algorithm associated with a single PID level; may be one or more strips.

RLD - ReLoaD; a BBN module which allows forced reloads.

round robin - a feature of the bus arbitration scheme which enforces fairness of access to the bus; when a device has been granted a bus access that terminates normally, it is not allowed to request another access until all those devices on that bus which are currently requesting access have been granted same.

RTC - Real Time Clock; a BBN card which causes PID levels at intervals of 25.6 ms. and 1.6 ms., has a program readable 16 bit counter that increments each 100 us., and three 16 bit readable switch registers.

SACK timer - a timer on the bus arbiter which guards against the case wherein a potential bus master requests an access, subsequently stops the precedence pulse, indicating that it will be the next master, but fails to assert SACK, acknowledging this fact.

satellite modem interface - a BBN module which interfaces to a satellite ground station transmitter; has features to enable use of the satellite channel in broadcast mode such as provision for switching the carrier and accurate timing of transmission and receipt of packets; consists of four boards: MSR, MST, MSX, and DMA.

scientific processor - the processor module with extended instruction set, including multiply, divide, double precision, etc.

select cycle - that part of an access which is concerned with selecting the next master.

self interrupt - a trap.

serial interface - a LEC module that interfaces asynchronous (start/stop) I/O devices; strappable for various speeds, character sizes, EIA vs. current loop, modem options, etc.; is half duplex and may either be polled or use classical interrupts; card type PSB.

service cycle - that part of an access wherein the master actually transacts with the slave.

SIMP - Satellite Interface Message Processor; an IMP which uses broadcast satellite channels as some of its communications links.

simplex - a communications path wherein communication can take place in only one direction.

slave - the participant in a bus transaction which responds to the master's request; e.g. the memory, when the processor is accessing it.

SID - Sense and Inhibit Drivers - a LEC card which forms part of the memory.

SLI - Synchronous Line Interface - a BBN card which interfaces two synchronous lines; a passive device which must be polled.

SMS - Synchronous Modem Simulator; a BBN card which interfaces two synchronous data sources, giving the appearance that the Pluribus is a modem; a passive device which must be polled.

SRN - System Release Notice.

status register - a location associated with a module whose bits report various combinations of the module; in a processor, register 8; in a serial or parallel interface, the first register; in a DMA device, the fourth and seventh registers.

step - the act of causing a processor to execute a single instruction and then halt.

start pointer - begin pointer.

STRB - an INFIBUS signal which indicates that a master is transacting with a slave; used to strobe address and, on a write, data.

strip - a set of instructions which are executed as a unit between references to the PID.

SUE - System User Engineered; the name of the line of LEC parts which make up part of a Pluribus.

SWB - SWitch Board; the front panel of the control panel.

subnetwork - the collection of node computers (IMPs) and communication lines of a network which perform the actual routing and transmission of the data.

synchronous - occurring at fixed time intervals.

synchronous line - a communications line where the timing information is derived from the transitions between data bits; data is usually sent in blocks.

synchronizer - a device which resolves two asynchronous time references.

system address space - the address space of common busses; seen directly by I/O devices and indirectly by processors after mapping; 512K words long.

system release notice - a document associated with each Pluribus system describing the location and configuration of each component.

TAG - Timing And Gating; a LEC card which forms part of the memory.

three phase wye - the type of AC power that the Pluribus PDU requires; there are five wires: one is for protective (green) ground; one is common for the other three, each of which has a 117 volt AC potential with the common, but the phase of the legs is staggered by 120 degrees.

throughput - the rate at which information can be processed.

TI - usually refers to Texas Instruments Silent 700 terminal.

timer - a device, hardware or software, which watches over the activity of a part of the system; the timer is periodically reset by the occurrence of an event which signifies correct operation and which should occur

periodically; should a specified time interval elapse without a reset, the timer will "time out" initiating some remedial action.

TIP - Terminal Interface message Processor; an IMP with built-in simple Host capabilities which allows users at terminals access to the network, obviating an external Host computer.

TOD - Time Of Day; a modification to a pair of PPB boards to interface a Systron-Donner clock.

trap - either a QUIT or an ILLOP.

very distant Host - a Host connected to the IMP via a communication link, with associated error detection and retransmission protocols.

VDH - Very Distant Host.

VISTAR - refers to an Infoton VISTAR terminal.

watchdog timer - cf. timer.

window - one of the four 4K regions of processor address space which can be mapped onto a page of system address space.

wire-wrap - a technology for making boards wherein connections are made by wrapping a wire around a pin located adjacent to the component.

word - the basic element of data; has 16 bits and two parity bits; there are two bytes in a word.

woven cable - a multiconductor cable constructed by weaving together several twisted pairs with a nylon thread.

write - a bus transaction where the data flow is from master to slave.

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PART 4: INDEX

PLURIBUS DOCUMENT 2: SYSTEM HANDBOOK

PART 5: REPRINTS OF PAPERS

Reprinted from —
AFIPS — Conference Proceedings
Volume 42

Montvale, N. J. 07645

A new minicomputer/multiprocessor for the ARPA network*

by F. E. HEART, S. M. ORNSTEIN, W. R. CROWTHER, and W. B. BARKER

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

Since the early years of the digital computer era, there has been a continuing attempt to gain processing power by organizing hardware processors so as to achieve some form of parallel operation.[1,2] One important thread has been the use of an array of processors to allow a single control stream to operate simultaneously on a multiplicity of data streams; the most ambitious effort in this direction has been the ILLIAC IV project.[3,4] Another important thread has been the partitioning of problems so that several control streams can operate in parallel. Often functions have been unloaded from a central processor onto various specialized processors; examples include data channels, display processors, front-end communication processors, on-line data preprocessors—in fact, I/O processors of all sorts. Similarly, dual processor systems have been used to provide load sharing and increased reliability. Still another thread has been the construction of pipeline systems in which sub-pieces of a single (generally large) processor work in parallel on successive phases of a problem.[5] In some of these pipeline approaches the parallelism is "hidden" and the user considers only a single control stream.

In recent years, as minicomputers have proliferated, groups of identical small machines have been connected together and jobs partitioned quite grossly among them. Most recently, our group and several others have been investigating this avenue further, attempting to reduce the specialization of the processors in order to employ independent processors with independent control streams in a cooperative and "equal" fashion.[6,7,8]

This paper describes a new minicomputer/multiprocessor architecture for which a fourteen-processor prototype is now (February 1973) being constructed. The hardware design and the software organization include many novel features, and the system may offer significant advantages in modularity and cost/performance. The system contains an expandable number of identical processors, each with some "private" memory; an expandable amount of "shared" memory to which all processors have equal access; and an expandable amount of I/O interface equipment, controllable by any processor. The system achieves unusual modularity and reliability by making all processors equivalent, so that any processor may perform any system task; thus systems can be easily configured to meet the throughput requirements of a particular job. The scheme for interconnecting processors, memories, and I/O is also modular, permitting interconnection cost to vary smoothly with system size. There is no "executive" and each processor determines its own task allocation.

A key issue throughout most of the attempts at parallel organization has been the difficulty of partitioning problems in such a way that the resulting computer program(s) can really take advantage of the parallel organization. This issue is raised in its most serious form when the parallel machine is expected to work well on a great diversity of problems as, for example, in a time-sharing system. Our machine design has been developed under the highly favorable circumstances that (1) the initial application, and a prior software implementation in a standard machine, was well understood; (2) the initial application lent itself to fragmentation into parallel structures; and (3) the design would be deemed successful if it handled only that one application in a meritorious fashion. However, we now believe that the design is advantageous for many other important applications as well and that it may herald a broadly useful new way to achieve increased performance and reliability.

The machine has been designed to serve initially as a modular switching node for the ARPA Network[9] and, in the following section, we briefly describe the ARPA Network application and the requirements that the network imposed upon the machine design. In subsequent sections we discuss our choice of minicomputer, describe our system design in some detail, discuss certain of the more interesting characteristics of multiprocessor behavior, and summarize our present status and plans for the near future.

* *This work was sponsored by the Advanced Research Projects Agency under Contracts DAHC15-69-C-0179 and FO8606-73-6-0027.*

ARPA NETWORK REQUIREMENTS

The ARPA Network, a nationwide interconnection of computers and high bandwidth (50 Kb) communication circuits, has grown during the past four years to include over 35 sites, with more than one computer at many sites. The computers at each site, called Hosts, obtain access to the net via a small communications processor known as an Interface Message Processor or IMP.[10] In order to permit groups without their own computer facility to access this powerful set of computer resources, a version of the IMP called a Terminal IMP allows, in addition, attachment of up to 63 local or remote terminals of a wide range of types.[11]

As a considerable simplification, the job to be handled by an IMP is that of a communications processor. Arriving messages must pass through an error control algorithm, be inspected to some degree (e.g., for destination), and generally be directed out onto some other line. Some incoming messages (e.g., routing control messages) must be constructed or digested directly by the IMP. The IMP must also concern itself with flow control, message assembly and sequencing, performance and flow monitoring, Host status, line and interface testing, and many other housekeeping functions. To perform these functions an IMP requires memory both for program and for message buffers, processing power for executing the program, and I/O units of various sorts for connecting to a variety of lines and devices. The original IMP, built around a Honeywell 516 processor with a 1 μs cycle time, could handle approximately three-quarters of a megabit per second of full duplex communications traffic. A later, smaller and cheaper (Honeywell 316) version handles about two-thirds as much traffic.

As the network has grown and as usage has increased, a number of demands for improvement have led to the need for a new "line" of IMP machines. Our intent is to provide a modular arrangement of flexible hardware from which it will be possible to construct both smaller and less expensive IMPs as well as far more powerful IMPs. An important specific objective is to obtain an IMP whose communications bandwidth could be at least an order of magnitude greater than the 516 IMP; such a high speed IMP would permit the direct connection of satellite circuits or land T-carrier circuits operating at approximately 1.3 megabits/second.

It is also desirable to improve the present IMP design in a number of other areas, as follows.

- Expandability of I/O: The present IMPs permit connection to a total of only seven high-speed circuits and/or Host computers. We would like to permit a much greater fanout so that an IMP might be connected to as many as 20 or more Host computers or to hundreds of terminals. This means that the number of interface units should be expandable over a wide range.
- Modularity: A number of groups have wished to make a network connection from a single Host at a considerable distance (miles) from the nearest IMP. We feel that such Hosts should be locally connected to a very small IMP in order to preserve consistency and standardization throughout the network. Therefore, a goal of this new hardware effort is the provision of a small and inexpensive but compatible IMP which could serve to connect a single, distant spur Host.
- Expandability of Memory: The new line of equipment is required for use in connection with satellite links (or longer faster links in general) and must therefore be able to expand its memory easily to provide the much greater buffer storage requirements of such links.
- Reliability: The new line of processors should be more reliable than the existing IMPs and ought to permit better self-diagnosis and simple isolation and replacement of failing units.

Of the requirements posed by the ARPA Network application, the most central was to obtain an order-of-magnitude traffic bandwidth improvement. We first considered meeting this requirement with highly specialized hardware, but the need to allow evolution of the communications algorithms, as well as the "bookkeeping" nature of much of the IMP task, militate against hardwired approaches and require the flexibility of a stored program computer. Thus we need a machine with an effective cycle time of 100 nanoseconds, a factor of ten faster than the present 1 μs IMP. Realizing that a single very fast and powerful machine would be difficult to build and would not give us compatible machines with a wide spectrum of performance, we began to consider the possibility of a minicomputer/multiprocessor in order to achieve the flexibility, reliability, and effective bandwidth required.

With the idea of a multiprocessor in mind we considered the IMP algorithm to determine which parts were inherently serial in nature and which could proceed in parallel. It seemed difficult to process a single message in a parallel fashion: the job was already relatively short and intimately coupled to I/O interfaces. However, there was much less serial coupling between the processing of separate messages from the same phone line and no coupling at all between messages from different phone lines. We thus envisage many processors, each at work on a separate message, with the number of processors carefully matched to the number of messages we expect to encounter in the time it takes one processor to deal with one message. With this simple image there seems to be no inherent limit to the parallelism we can achieve—the ultimate limit would be set by the size of the multiprocessor we can build.

CHOICE OF THE PROCESSOR

In designing a multiprocessor for the IMP application, we found ourselves iteratively exploring two related but distinct issues. First, assuming that the problem of interconnection could be solved, what minicomputer would be

a sensible choice from the price/performance and physical points of view? Second, and much harder: for any specific machine, how did the CPU talk to memory, how would multiple CPUs, memories, and I/O be interconnected to form a system, and how would the program be organized?

Since the program for the existing IMPs was well understood, it was possible to identify key sections of that program which consumed the majority of the processing bandwidth. Then, for each sensible minicomputer choice, we could ask how many CPUs of this type would be needed to provide an effective 100 nanosecond cycle time; and given a price list, physical data, and a modest amount of design effort, we could define the physical structure and the price of the resulting multiprocessor. With this general approach, we examined the internal design of about a dozen machines, and actually wrote the key code in many cases. Using the fastest available minicomputers it was possible to arrive at configurations with only three or four processors; using the slowest choices, systems with 20 CPUs or more were required.

If we defer the interconnection and contention problems for a moment, it is interesting to note that "slow and cheap" may win over "fast and expensive" in this kind of multiprocessor competition to achieve a stated processing bandwidth. This is an especially happy situation if, as in our case, a spectrum of configurations is needed, including a very tiny cheap version.

In considering which minicomputer might be most easily adaptable to a multiprocessor structure, the internal communication between the processor and its memory was of primary concern. Several years ago machines were introduced which combined memory and I/O busses into a single bus. As part of this step, registers within the devices (pointers, status and control registers, and the like) were made to look like memory cells so that they and the memory could be referenced in a homogeneous manner. This structure forms a very clean and attractive architecture in which any unit can bid to become master of the bus in order to communicate with any other desired unit. One of the important features of this structure is that it made memory accessing "public"; the interface to the memory had to become asynchronous, cleanly isolable electrically and mechanically, and well documented and stable. A characteristic of this architecture is that all references between units are time multiplexed onto a single bus. Conflicts for bus usage therefore establish an ultimate upper bound on overall performance, and attempts to speed up the bus eventually run into serious problems in arbitration.[12]

In 1972 a new processor—the Lockheed SUE[13]—was introduced which follows the single bus philosophy but carries it an important step further by removing the bus arbitration logic to a module separate from the processor. This step permits one to consider configurations embodying multiple processors and multiple memories as well as I/O on a single bus. The SUE CPU is a compact, relatively inexpensive (approximately $600 in quantity), quite slow processor with a microcoded inner structure. This slowness can be compensated for by simply doubling or trebling the number of processors on the bus; performance is limited largely by the speed of the bus. With this bus architecture it becomes attractive to visualize multibus systems with a "bus coupling" mechanism to allow devices on one bus to access devices on other busses.

Similar approaches can be implemented with varying degrees of difficulty in systems with other bus structures, and we examined several approaches in some detail for those processors whose cost/performance was attractive. Rather fortuitously, the minicomputer which exhibited the most attractive bus architecture also was extremely attractive in terms of cost/performance and physical characteristics. This machine, the Lockheed SUE, would require fourteen processors to achieve the effective 100 nanosecond cycle time, and we embarked on the detailed design of our multiprocessor on that basis.

SYSTEM DESIGN

Although our design permits systems of widely varying size and performance, in the interest of clarity we will describe that design in terms of the particular prototype now under construction. Our overall design is represented in Figure 1. We require fourteen SUE processors to obtain the necessary processing bandwidth, and we estimate that 32K words of memory will be required for a complete copy of the operational program and the necessary communication buffer storage. The I/O arrangements must allow easy connection of all the communications interfaces, appropriate to the IMP job (modem interfaces, Host interfaces, terminal interfaces) as well as standard peripherals and any special devices appropriate to the multiprocessor nature of the system.

Some of the basic SUE characteristics are listed in Table I. From a physical point of view, the SUE chassis represents the basic construction unit; it incorporates a printed circuit back plane which forms the bus into which 24 cards may be plugged. From a logical point of view this bus simply provides a common connection between all

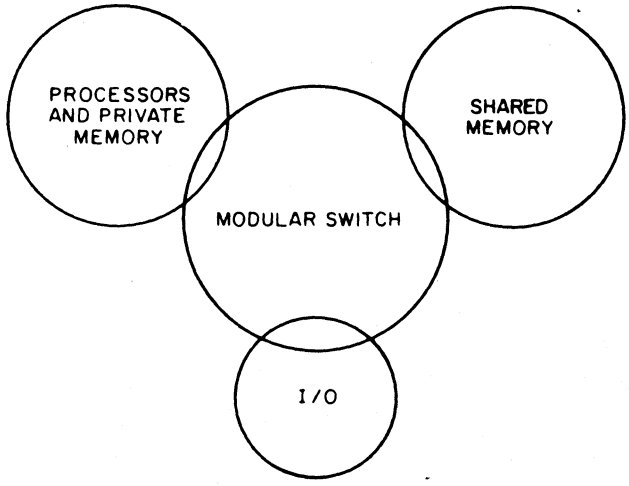


Figure 1—System structure

TABLE I—SUE Characteristics

16-bit word
8 General Registers
△3.7 μs add or load time
Microcoded
Two words/instruction typical
8-½″×19″×18″ chassis
64K bytes addressable by a single instruction
~$3K for: 1 CPU+4K Memory+Power, Rack, etc.
200 ns minimum bus cycle time
850 ns memory cycle time
425 ns memory access time

units plugged into the chassis. We are using these chassis for the entire system: processor, memory, and I/O. All specially designed cards as well as all Lockheed-provided modules plug into these bus chassis. With this hardware, the terms "bus" and "chassis" are used somewhat interchangeably, but we will commonly call this standard building unit a "bus." Each bus requires one card which performs arbitration. A bus can be logically extended (via a bus extender unit) to a second bus if additional card space is required; in such a case, a single bus arbiter controls access to the entire extended bus.

We can build a small multiprocessor just by plugging several processors and memories (and I/O) into a single bus. For larger systems we quickly exceed the bandwidth capability of a single bus and we are forced to multi-bus architecture. Then, from a construction viewpoint, our multiprocessor design involves assigning processors, memories and I/O units to busses in a sensible manner and designing a switching arrangement to permit interconnection of all the busses. Of course, the superficial simplicity of this construction viewpoint completely hides the many difficult problems of multiprocessor system design; we will try to deal with some of those issues in the following sections.

Resources

A central notion in a parallel system is the idea of a "resource," which we define to mean a part of the system needed by more than one of the parallel users and therefore a possible source of contention. The three basic hardware resources are the memories, the I/O, and the processors. It is useful to consider the memories, furthermore, as a collection of resources of quite different character: a program, queues and variables of a global nature, local variables, and large areas of buffer storage.

The basic idea of a multiprocessor is to provide multiple copies of the vital resources in the hope that the algorithm can run faster by using them in parallel. The number of copies of the resource which are required to allow concurrent operation is determined by the speed of the resource and the frequency with which it is used. An additional advantage of multiple copies is reliability: if a system contains a few spare copies of all resources, it can continue to operate when one copy breaks.

It may seem peculiar to think of a processor as a resource, but in fact in our system the parallel parts of the algorithm compete with each other for a processor on which to run. We take the view that all processors shall be identical and equal, and we go to some trouble to insure that this is in fact so. As a consequence no single processor is of vital importance, and we can change the number of processors at will. A later section will describe how the processors coordinate to get the job done without a master of some sort.

Processor busses

A SUE bus can physically and logically support up to four processors. As more processors are added to a bus, the contention for the bus increases, and the performance increment per processor drops; but the effective cost per processor also drops, since the cost for the chassis, power supply, bus arbitration, etc., is amortized over the number of processors.

Roughly speaking, using two processors per bus loses almost nothing in processor performance, using three processors per bus loses significant efficiency, and adding a fourth processor gains less than half an "effective processor." After careful examination of the logical, economic and physical aspects of this choice, we decided to use two processors per processor bus, and we thus require seven processor busses in our initial multiprocessor system.

The next question was how the processors should access the program. In our application, some parts of the program are run very frequently and other parts are run far less frequently. This fact allows a significant advantage to be gained by the use of private memory. When a processor makes access to shared memory via the switching arrangement, that access will incur delays due to contention and delays introduced by the intervening switch. We therefore decided to use a 4K local memory with each processor on its bus to allow faster local access to the frequently run code; these local memories all typically contain the same code. With this configuration and in our application, the ratio of accesses to local versus shared memory is better than three to one. This not only reduces contention delays for access to the shared memory but also cuts the number of accesses which suffer the delays.

The final configuration of a processor bus is shown in Figure 2(a). The units marked "Bus Coupler" have to do with our multiprocessor switching arrangement, which will be discussed below.

*Shared memory busses**

The shared memory of our multiprocessor is intended to contain a copy of the program as well as considerable storage space for message buffering, global variables, etc. Application-dependent considerations led us to select a

* *The terms "I/O bus" and "memory bus" as used here and henceforth are not the same as conventional I/O and memory busses.*

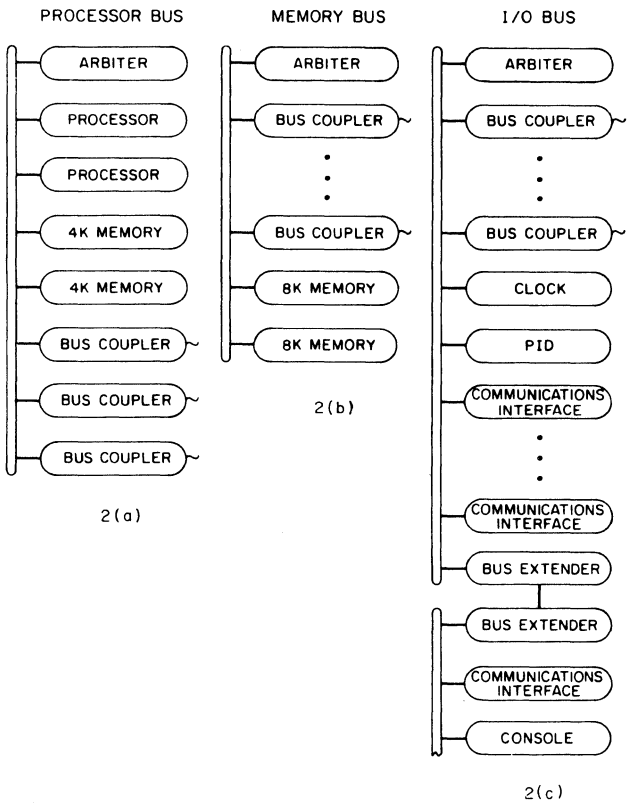


Figure 2—Bus structures

32K memory, but it is possible to configure this memory on a single bus or to divide the memory onto several busses. We first concluded that four logical memory units would be appropriate in order to reduce processor contention to an acceptable level. Then, since the bus is considerably faster than the memories, it is feasible to place two logical memory elements on a single bus with almost no interference. Thus, we are planning two memory busses in the initial multiprocessor; the configuration of a common memory bus is shown in Figure 2(b).

I/O busses

The I/O system of the multiprocessor employs standard SUE busses with standard bus arbitration units on those busses. Into the bus will be plugged cards for each of the various types of I/O interfaces that are required, including interfaces for modems, terminals, Host computers, etc., as well as interfaces for standard peripherals. Our initial system has a single I/O bus and Figure 2(c) shows its configuration; the specialized units shown (a "Clock" and "Pseudo Interrupt Device") are system-wide resources that are used to control the operation of the multiprocessor. The I/O bus will also be the access route for the multiprocessor console; we plan to use a standard alphanumeric display terminal which can be driven by code in any processor, and no conventional consoles will be used.

Interconnection system

Our prototype multiprocessor is now seen to contain seven processor busses, two shared memory busses and an I/O bus. To adhere to our requirement that all processors must be equal and able to perform any system task, these busses must be connected so that all processors can access all shared memory, so that I/O can be fed to and from shared memory, and so that any of the processors may control the operation and sense the status of any I/O unit.

A *distributed* inter-communication scheme was chosen in the interest of expandability, reliability, and design simplicity. The atom of this scheme is called a Bus Coupler, and consists of two cards and an interconnecting cable. In making connections between processors and shared memory, one card plugs into a shared memory bus, where it will request cycles of the memory; the other card plugs into a processor's bus, where it looks like memory. When the processor requests a cycle within the address range which the Bus Coupler recognizes, a request is sent down the cable to the memory end, which then starts contending for the shared memory bus. When selected, it requests the desired cycle of the shared memory. The memory returns the desired information to the Bus Coupler, which then provides it to the requesting processor, which, except for an additional delay, does not know that the memory was not on its own bus. Note that the memory access arbitration inherent in any memory switching arrangement is handled by the SUE Bus Arbiter controlling the shared memory bus, while the Bus Coupler itself is conceptually straightforward.

One additional feature of the Bus Coupler is that it does address mapping. Since a processor can address only 64K bytes (16 bit address), and since we wished to permit multiprocessor configurations with up to 1024K bytes (20 bit address) of shared memory, a mechanism for address expansion is required. The Bus Coupler provides four independent 8K byte windows into shared memory. The processor can load registers in the Bus Coupler which provide the high-order bits of the shared memory address for each of the four windows.

Given a Bus Coupler connecting each processor bus to each shared-memory bus, all processors can access all shared memory. I/O devices which do direct memory transfers must also access these shared memories. These I/O devices are plugged into as many I/O busses as are required to handle the bandwidth involved, and bus couplers then connect each I/O bus to each memory bus. Similarly, I/O devices also need to respond to processor requests for action or information; in this regard, the I/O devices act like memories and Bus Couplers are again used to connect each processor bus to each I/O bus. The path between processor busses and I/O busses is also used in a more sophisticated fashion to allow processors to examine and control other processors; this subject is described in a later section.

The resulting system is shown in Figure 3. One is struck by the number of bus couplers: P*I+I*M+P*M bus couplers are required for a system with P processor bus-

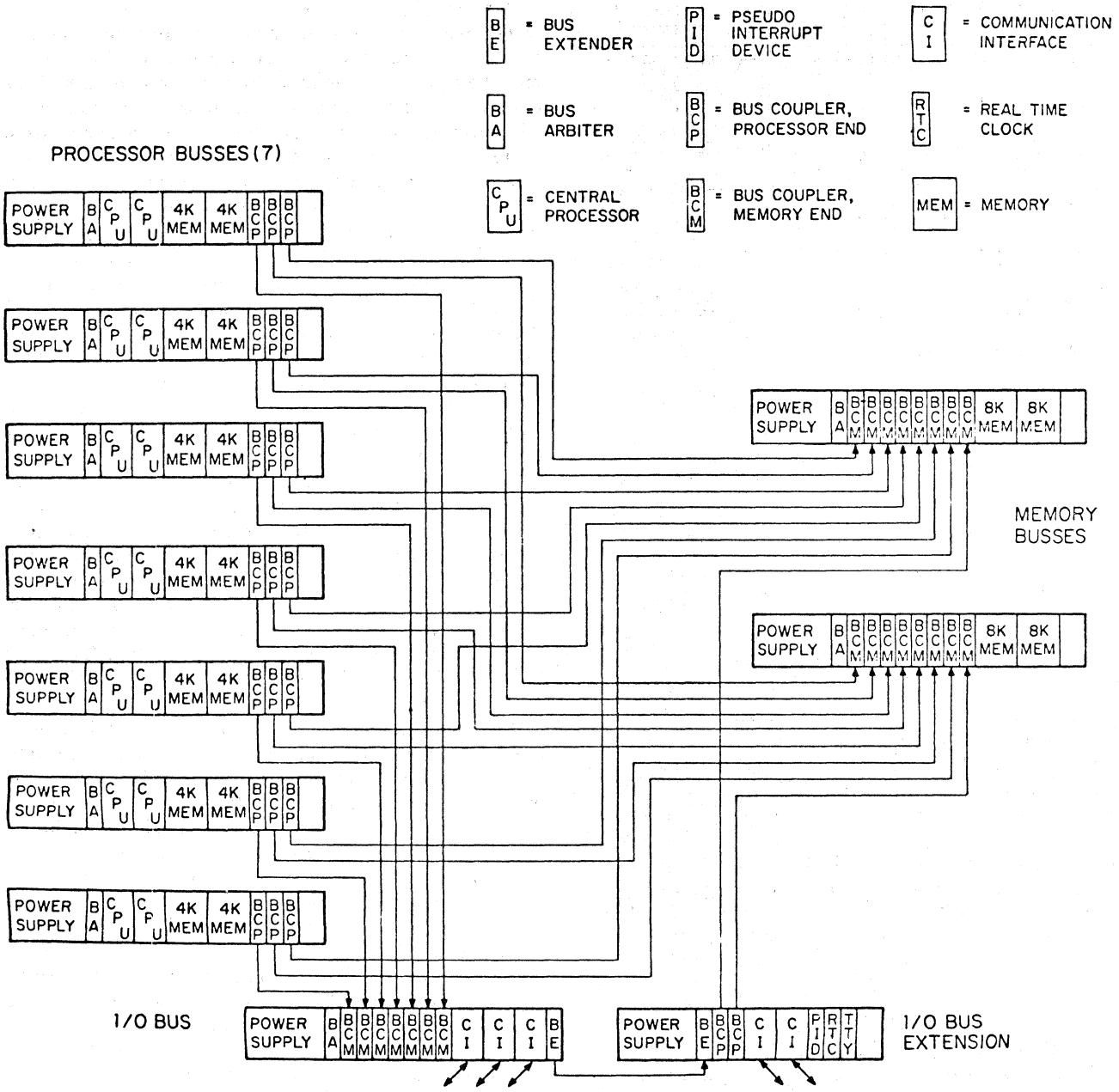


Figure 3—Prototype system

ses, I I/O busses, and M memory busses. In the case of our initial multiprocessor, 23 are needed.

This modular interconnection approach clearly permits great flexibility in the number and configuration of busses, and allows interconnection cost to vary smoothly with system size. We believe that this modular interconnection scheme also permits a complex hierarchical arrangement of busses. Actually the system exhibits a pronounced hierarchical structure already. A processor accesses the local memory when it needs instructions or local variables. Two such processor-memory combinations form a dual processor, which can be regarded as a unit and which needs access to shared resources, such as global variables, free buffers, and I/O interfaces. When one copy of a resource can only support a limited number of users, it seems sensible to provide only the corresponding limited number of connections. If a multiprocessor of this type were to grow larger, the physical number of bus couplers as well as increasing contention problems might not permit the connection of each processor to all of common memory, but might instead require a multi-level structure where groups of processors were connected to an

intermediate level bus which was in turn connected to a centralized common memory. We have not explored this domain but feel it is an interesting area for future work.

MULTIPROCESSOR BEHAVIOR

Until the processors interact, a multiprocessor is a number of independent single processor systems: it is the interaction which poses the conceptual as well as the practical problems. If the various processors spend their time waiting for each other, the system degrades to a single processor equivalent; if they can usefully run concurrently, the processing power is multiplied by the number of processors. If the failure of a single processor takes the system down, the system reliability is only the probability of all processors being up; if working processors can diagnose and heal or amputate faulty processors and proceed with the job, the system reliability approaches the probability of *any* processor being up. We now consider how to keep processors running concurrently, and then how to keep the system running in the case of module failure.

The first problem in making the machines run independently is the allocation of runnable tasks to processors, so that the full requisite power can be quickly brought to bear on high priority tasks. Our scheme for doing this rests on four key ideas: (1) We break the job up into a set of tiny tasks. (2) Our processors are all identical, asynchronous, and capable of doing any task. (3) We keep a queue of pending tasks, ordered by priority, from which each processor at its convenience gets its next task. (4) For speed and efficiency, we use a hardware device to help manage the queue.

By breaking the job up into smaller and smaller tasks until each one runs in under 300 μs, we effectively determine the responsiveness of our system. Once started, a task must run to completion, but there will be a reconsideration of priorities at the beginning of each new task. We have chosen 300 microseconds as the maximum task execution time because this compromise between efficiency and responsiveness is well matched to the execution time of key IMP functions.

By making the processors identical, we can use the same program in systems of widely varying size and throughput capability. Any processor can be added to or removed from a running system with only a slight change in throughput. The power of all processors quickly shifts to that part of the algorithm where it is most needed.

By queuing pending tasks, we keep track of what must be done while focusing on the most important tasks. By using a passive queue in which the processors check for a new task when they are ready, we avoid some nasty timing problems. Tasks may be entered into the queue at any time, either by a processor or by the hardware I/O devices. This approach is an extremely important departure which avoids the use of conventional interrupts and the associated costs of saving and restoring machine state. Further, this approach neatly sidesteps the problem of routing interrupts to the proper processor.

We could not afford a software queue both because it was slow to use and because processors would have been waiting for each other to get access to the queue. Instead we use a special hardware device called a Pseudo Interrupt Device (PID), which keeps in hardware a list of what to do next. A number can be written to the PID at any time and and it will be remembered. When read, the PID returns (and deletes) the highest number it has stored. By coding the numbers to represent tasks, and keeping the parameters of the tasks in memory, a processor can access the PID at the end of each task and determine very rapidly what it should do next.

Contention

Clearly, the PID must give any task to exactly one processor. This is guaranteed because the PID is on a bus that can be accessed by only one processor at a time and because the PID completes each transaction in a single access. This is an example of the more general problem that whenever two users want access to a single resource there must be an interlock to let them take turns. This is true at many levels, from contention for a bus to processor contention for shared software resources such as a free list. When all the appropriate interlocks have been provided, the performance of the multiprocessor will depend rather critically on the time wasted waiting at these interlocks for a resource to become free. As discussed above, whenever conflicts become a serious problem one provides another copy of the resource. We studied our system behavior carefully, noting areas of conflict, in order to know how many additional copies of heavily accessed resources to provide. Table II provides examples of delays due to various conflicts. Practically speaking, the curve of delay vs. number of resources has a rather sharp knee, so that it is meaningful to make such statements as "a memory bus supports eight processors" or "a free list supports eight processors." Of course, these statements are application related and depend on the frequency and duration of accesses required.

With interlocks, deadlocks become possible (in both hardware and software). For example, a deadlock occurs

TABLE II—Expected System Slowdown Due to Contention Delays

Slowdown	Cause
5.5%	Contention for a Processor Bus.
3%	Contention for the Shared Memory Busses.
5%	Contention for the Shared Memories.
10%	Contention for a single system-wide software resource, assuming each processor wants the resource for 6 instructions out of every 120 instructions executed.
1.7%	Contention for one of two copies of a system-wide software resource, as above.
0.15%	Contention for the parameters of a single 1.3 megabit phone line, assuming the parameters will be used for 160 microseconds every 800 microseconds.

when each of two processors has claimed one of two resources needed by both. Each waits indefinitely for the other's resource to become available.[14] Unless there is a careful systematic approach to interlocks, deadlocks interlock, and require that a processor never compete for a resource when it already owns a higher numbered resource. It is not always practical or possible to do this, although we expect to be able to do so with the IMP algorithms.

An interesting example of a deadlock occurs in our bus coupling. To permit processors to access one another, for mutual turn on, turn off, testing, etc., the path connecting each processor bus with the I/O bus is made bi-directional. Thus processors access one another via the I/O bus. In a bi-directional coupler, a deadlock arises when units obtain control of their busses at each end and then request access via the coupler to the bus on the other end. Because the backward path is infrequently used, we simply detect such deadlocks, abort the backward request and try again.

Reliability

We have taken a rather ambitious stand on reliability. We plan to detect a failing module automatically, amputate it, and keep the system running without human intervention if at all possible. Critical to our approach is the fact that there are several processors each with private memory and thus each able to retreat to local operation in the face of system problems. To reduce our vulnerability further, power and cooling are provided on a modular basis so that loss of a single unit does not jeopardize system operation. We are only mildly concerned with the damage done at the time of a failure, because the IMP system includes many checks and recovery procedures throughout the network.

The first sign of a failure may be a single bit wrong somewhere in shared memory, with all units apparently functioning properly. Alternatively, the failure may strike catastrophically, with shared memory in shambles and the processors running protectively in their local memories. Against this spectrum we cannot hope for a systematic defense; instead we have chosen a few defensive strategies.

So long as a module is failing, recovery is meaningless. We must run diagnostics to identify the bad module, or see if cutting a module out at random helps things. We feel that identifying such a solid failure will be relatively easy. Since a processor without couplers is completely harmless, once we identify a malfunctioning processor, we amputate it by turning off its bus couplers. We considered the possibility of a runaway processor turning good processors off. This is unlikely to begin with but we decided to make it even less likely by requiring a particular 16-bit password to be used in turning off a coupler. A runaway processor storing throughout shared memory would need this password in its accumulator to accidentally amputate. Similarly we require a password for one processor to get at another's local memory.

Against intermittents we use a strategy of dynamic reinitialization. Every data structure is periodically checked; every waiting state is timed out; the code is periodically checksummed; memory transfers are hardware parity checked; memory is periodically tested; processors are periodically given standard tests. Whenever anything is found wrong, the offending structure is initialized. Using this scheme we may not know what caused a failure, but its effects will not persist. In the most extreme cases we will need to reload all the program in main memory. Fortunately we have a communications network handy to load from. This technique of reloading has worked remarkably well in the current ARPA Network. Each processor has a copy of the reload program in its local memory, thus making loss of reload capability unlikely.

We might seem to be vulnerable to memory or I/O failures, particularly those involving the PID and the clock. If these modules fail it does indeed hurt us more, but only because we have fewer modules of these types in our system. If we provide redundant modules, the system can reconfigure itself to substitute a spare module for a failed one. Our design allows multiple I/O busses with multiple PIDs and clocks, and we could even have separate backup interfaces to vital communication lines on separate busses.

To summarize, the mainstay of our reliability scheme is a system continually aware of the state of things and quickly responding to unpleasant changes. The second line of defense consists of drastic actions like amputation and reloading. Assuming we can make all this work, we will have quite a reliable system, perhaps even one in which maintenance consists of periodic replacement of those parts which the system itself has rejected.

STATUS AND NEAR FUTURE

In February 1973, as this paper is submitted, we are very much in the middle of our multiprocessor development. Much progress has been made and we are increasingly confident of the design, but much work remains to be done.

The broad design is complete; all Lockheed-provided units (CPUs, memories, busses, etc.) have been delivered; prototype wire-wrapped versions of the crucial special modules have been completed, including the Bus Couplers, Pseudo Interrupt Device, clock, and modem interfaces; and a multi-bus, multi-processor-per-bus assembly has been successfully tried with a test program. A substantial program design effort has been in progress and coding of the first operational program has been started. We are still doing detailed design of some hardware, and we are still learning about detailed organizational issues as the software effort proceeds. An example of such an

Correction (p. 536 of original text, first column, second new sentence):
Unless there is a careful systematic approach to interlocks, deadlocks become almost a certainty. One technique is to assign a unique number to each resource for which there is an interlock, and require that a processor never compete for a resource when it already owns a higher numbered resource.

area is: exactly how is it best for processors to watch each other for signs of failure?

We currently anticipate the parts cost of the prototype fourteen-processor system, without communication interfaces, to be under $100K.

Hopefully, by the time this paper is presented in June 1973, we will be able to report an operational prototype multiprocessor system. Beyond that, our schedule calls for the installation of a machine in the ARPA Network by about the end of 1973. We also plan to construct many variant systems out of this kit of building blocks, and to experiment with systems of varying sizes. As part of this work, we plan to concentrate on the very smallest version that may be sensible, in order to provide a minimum cost IMP for spur applications in the ARPA Network.

As the design has proceeded, our attraction to the general approach has increased (perhaps a common malady), and we now believe that the approach is applicable to many other classes of problems. We expect to explore such other applications as time permits, with initial attention to two areas: (1) certain specialized multi-user systems, and (2) high bandwidth signal processing.

With our presently planned building blocks, although we do not yet know what will limit system size, we do not now see any intrinsic problem in constructing systems with fifty or a hundred processors. As improvements in integrated circuit technology occur, and processors and memories become smaller and cheaper, organization and connection become the paramount questions in multiprocessor design. We expect to see many attempts at multiprocessors, and are hopeful that the ideas embodied in this design will help to steer that technology. Perhaps minicomputer/multiprocessors will soon represent real competition for the various brontosaurus machines that now abound.

ACKNOWLEDGMENTS

Our new machine design is a product of many minds. We gratefully acknowledge the specific design contributions of M. Kraley, A. Michel, M. Thrope, and R. Bressler. Helpful criticism and an important idea about the Pseudo Interrupt Device were contributed by D. Walden. Assistance in planning and in the choice of building blocks was contributed by H. Rising. Helpful ideas and criticism were provided by J. McQuillan, B. Cosell, and A. McKenzie. Assistance with support software was provided by J. Levin.

We also wish to express appreciation for the support and encouragement provided by Dr. L. Roberts of the Advanced Research Projects Agency.

REFERENCES

1. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", *Proc. IEEE*, Vol. 54, No. 12, pp. 1889-1901, December, 1966.
2. Lorin, H., *Parallelism in Hardware & Software - Real and Apparent Concurrency* Prentice-Hall, 1971.
3. Slotnick, J. L., Bork, W. C., McReynolds, R. C., "Solomon", *AFIPS Conference Proceedings*, FJCC 1962.
4. Barnes, G. H., et al, "The Illiac IV Computer", *IEEE Trans.* C-17, Vol. 8, pp. 746-757, August 1968.
5. Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91 - Machine Philosophy and Instruction Handling", *IBM Journal* No. 11, January 1967, pp. 8-24.
6. Cohen, E., "Symmetric Multi-Mini-Processors, A Better Way to Go?" *Computer Decisions*, January 1973.
7. Wulf, W. A., Bell, C. G., "C.mmp - A Multi-Mini Processor", *AFIPS Proceedings*, FJCC, Vol. 41, 1972.
8. Cosserat, D. C., "A Capability Oriented Multi-Processor System for Real-Time Applications", *Computer Communication Proc. ICCC*, pp. 282-289, October 1972.
9. Roberts, L. G., Wessler, B. D., "Computer Network Development to Achieve Resource Sharing" *AFIPS Proceedings*, SJCC, Vol. 36, 1970.
10. Heart, F. E., et al, "The Interface Message Processor for the ARPA Computer Network", *AFIPS Proceedings*, SJCC, Vol. 36, 1970.
11. Ornstein, S. M., et al., "The Terminal IMP for the ARPA Computer Network", *AFIPS Proceedings*, SJCC, Vol. 40, 1972.
12. Chaney, T., Ornstein, S., Littlefield, W., "Beware the Synchronizer", *Proc. COMPCON Conference*, 1972.
13. *SUE Computer Handbook*, Lockheed Electronics Company, Los Angeles, 1972.
14. Holt, R. C., "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

SUPPLEMENTARY BIBLIOGRAPHY

Amdahl, G. M., *Engineering Aspects of Large High-Speed Computer Design - Part II Logical Organization*, IBM Tech. Report TR00.1227, December 1964.

Baskin, H. B., et al, "A Modular Computer Sharing System," *CACM*, Vol. 12, No. 10, October 1969, p. 551.

Bell and Newell, *Computer Structures*, McGraw-Hill, 1971.

Bell, G., et al, *C.mmp the CMU Multiminiprocessor Computer*, Dept. of Computer Science, Carnegie Mellon Univ., August 1971.

Burnett, G. J., et al., "A Distributed PROCESSING System for General Purpose Computing", *AFIPS Proceedings*, FJCC, Vol. 31, 1967.

Dijkstra, E. W., "Cooperating Sequential Processes", in *Programming Languages*, (Gennys, F., ed.), Academic Press, pp. 43-110, 1968.

Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*", Vol. C-21, No. 9, September 1972.

Flynn, M. J., "Very High-Speed Computing Systems", *Proc. IEEE*, Vol. 54, No. 12, pp. 1901-1909, December, 1966.

Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously," *AFIPS Proceedings*, FJCC, pp. 108-113, 1959.

McQuillan, J. M., et al, "Improvements in the Design and Performance of the ARPA Network", *AFIPS Proceedings*, FJCC, Vol. 41, 1972.

Ornstein, S. M., Stucki, M. J., Clark, W. A., "A Functional Description of Macromodules" *AFIPS Proceedings*, SJCC, Vol. 30, 1967.

Pirtle, M., "Intercommunication of Processors & Memory", *AFIPS Proceedings*, FJCC, Vol. 31, 1967.

Randell, B., "Operating Systems - The Problems of Performance and Reliability", *IFIP Congress 71*, Ljubljana, North Holland Pub. Co., 1972, pp. 281-290.

A Description of the Advanced Scientific Computer System, Texas Instruments, Inc., 1972.

Thornton, J. E., "Parallel Operation in the Control Data 6600", *AFIPS Proceedings*, FJCC, Vol. 26, 1964.

Wulf, W., et al, *Hydra— A Kernel Operating System for C.mmp*, Dept. of Computer Science, Carnegie Mellon Univ., 1971.

THE BBN MULTIPROCESSOR

S.M. Ornstein, W.B. Barker, R.D. Bressler,
W.R. Crowther, F.E. Heart, M.F. Kraley,
A. Michel, M.J. Thrope

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

This paper appeared in the Computer Nets Supplement to the Proceedings of the Seventh Hawaii International Conference on System Sciences, January 1974, and is reproduced with the permission of the publisher, Western Periodicals Company, California.

THE BBN MULTIPROCESSOR*

S.M. Ornstein, W.B. Barker, R.D. Bressler, W.R. Crowther
F.E. Heart, M.F. Kraley, A. Michel, M.J. Thrope
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

Abstract

The BBN multiprocessor has gone from conception to prototype over the past year. It is highly modular at several logical and physical levels and will soon be a new IMP in the ARPA Network. It is very flexible both in the range of bandwidths it can handle and the number and type of interfaces it can accommodate.

1. INTRODUCTION

Last year we presented a paper which described the multi-processor we were then setting out to build as a new IMP for the ARPANET [1,2]. Much has been accomplished in this past year and we report here on progress made as well as on some important features of the system that have evolved. Familiarity with the earlier paper is assumed in what follows.

The architecture, as previously described, is highly modular and allows for IMPs of greater or lesser processing power than the present 516/316-based IMPs, as well as for many more and more varied phone line and Host interfaces. The hardware consists of busses joined together by special bus couplers of our design. There are processor busses each of which contains two processors, each in turn with its own "private" 4K memory to store frequently run code. The more processor busses, the greater the system processing power. There are memory busses to house the segments of multiported "common" memory — the more memory busses, the more memory ports. Finally, there are I/O busses which house device and line controllers as well as a special (priority ordered) task disburser which replaces the traditional priority interrupt system. The latter allows equality among the processors so that if some fail the rest can continue to run all system tasks, albeit at reduced capacity.

2. DESIGN ISSUES

In this section we describe features we have designed into the system, some of the more interesting of which relate to reliability issues.

2.1 ADDRESSING & LOCKING

The Lockheed SUE, with a 15-bit word address, can address up to 32K words. A 1.5-megabit line running over a 1/2 sec. round trip satellite channel holds 750,000 bits or about 50,000 words, copies of which must be held in the IMP for possible retransmission. Address expansion is thus inescapable and to allow for several such lines and be reasonably unbound by address space, we have allowed for half a million words. The bus coupler serves as the vehicle for address expansion. 8K of a processor's address space are used for direct references to its private memory. (Although we expect to use only 4K, 8K has been set aside to allow for growth.) Another 8K is used principally for addressing system I/O (on the up to four I/O busses). We assign 8 addresses to each I/O device for pointers and status and control registers; 960 devices can be accommodated in all.

16K of each processor's address space is mapped through the couplers to common memory. At the processor end of each coupler are four program-settable map registers for each possible processor on the bus. (We

*This work was supported by the Advanced Research Projects Agency under Contracts DAHC15-69-C-0179 and F08606-73-C-0027.

expect to use only two processors per bus but up to four are allowed for.) These map registers expand a 15-bit address to a 19-bit system address on the memory busses. By use of the maps, each processor can thus access, at any one time, four 4K pages in system address space. Read accesses through a particular one of these windows are turned by the coupler into read-clear operations, thereby providing the indivisible test-and-modify operation required for program interlocking in a multi-processor. (The processor itself presently lacks such an instruction.)

2.2 ACCESS ENABLING

The coupler paths that connect processor busses into memory and I/O busses have program settable enabling switches at their far (memory and I/O) ends, thus permitting processors to be cut in and out of the system. To allow processors to access one another and to permit reloading as discussed below, we have provided reverse paths in the processor to I/O couplers which also have enabling switches. Normally the forward paths to memory and I/O are turned on and the backward paths are shut off. Since these paths represent a hazard whereby a "sick" processor or device could damage healthy processors, we have arranged that only by storing a password at the proper address can a switch be changed. This greatly reduces the probability that a berserk processor painting memory will affect the path. A processor can neither enable nor disable its own access paths but one processor, deciding that another is sick and should be eliminated from the system, can amputate the bus of the offending processor. It can be similarly reinstated later.

The logic upon which amputation decisions are based is not yet fully understood and will be worked out as experience grows. We expect to require all processors to execute periodic healthiness-proving tasks. A regular system task, performed by any free processor, verifies that all processors have passed their tests and amputates any unhealthy one(s). Protective embellishments easily suggest themselves and we expect to do what seems necessary.

2.3 DISCOVERY

The operational program implements the IMP algorithm with whatever hardware is working at a particular site at a given time. The program discovers the hardware configuration as follows: Memory is found by trying to access it; a failure interrupt results if Memory is not there. Processors are found by accessing a register whose response indicates if the processor is absent, running or halted. I/O Devices are found by reading the 1st word of every possible device in I/O space — a failure interrupt means no Device, a response returns a unique 16-bit device type. Any parameters needed to run the devices are available as status words in the 8-word block. It is somewhat harder to find where the bus boundaries are, but they too can be found by searching for the bus coupler disable switches. In the event that there is some property we cannot otherwise discover, we have set aside 3 registers (associated with the clock device) to hold this information. For example, the IMP number (used for network routing) is contained in 8 bits of these registers.

The Discovery logic is not an initialization phase; rather the program periodically runs through the Discovery logic and reconfigures whenever a change occurs. It thus automatically adapts the IMP algorithm not only to the wide variety of possible configurations but also to those which contain broken components.

2.4 PARITY

At present the memories we are using do not store parity; however, we have built into our system design (and into the hardware) mechanisms to incorporate parity. These mechanisms have been tested with prototype parity memory and we have recently ordered parity memories for our production machines. We use a novel parity computation based not only upon the contents of a word but also on its address. The scheme also detects both "all ones" and "all zeros" failures. For writes to common memory, parity is computed at the processor end and fed, via the coupler, to the memory where it is stored with the word. Reads from memory fetch this stored parity, which is compared to a recomputed parity at the processor end of the coupler, thus checking both the memory and coupler paths in both directions. For units on the I/O bus, in order to check the coupler paths, a special card computes and transmits parity for all words being read from the I/O bus by the processors and checks parity on all words arriving from processor busses.

2.5 RELOADING

At present we use paper tape to load the system. The operator starts a processor which, from tape, loads its own private memory, its map registers and thereby any or all of common memory. It also loads, using backward coupling, the private memories on all other processor busses in the system. After the memory has been loaded, a startup procedure is executed which finally turns on the other processors.

Since all crucial switches, parameters, registers and control flip flops have been made addressable by reads and writes, load-

ing the system and starting it up can be done by externally force feeding it with the right set of addresses and data. Although we presently use paper tape in conjunction with a bootstrap ROM executed by a processor for this purpose, we are planning to construct a means whereby the system can be force fed directly from the network. The mechanism for this is a device on the I/O bus which monitors phone lines from adjacent IMPs looking for a special format which signals arrival of reload information. The card then performs the reload by executing store type bus cycles using the reload data.

This sort of operation, which looks forward to elimination of paper tape, switches, and other operator dependent functions, is appropriate to the IMP job. If a running system fails, as viewed from the net, the first step is to send it a regular "for IMP" message which causes a standard system restart to be attempted. If that seems not to work, the next step is to send another regular message trying to activate the reload-from-the-net code in hopes that it is still intact. Only if that fails would one attempt to force a full restart from scratch, in which case the special card described above is called into play. The first data sent halts the processors in order to stop any interfering activity. Then the reload-from-the-net code is refreshed and finally a processor restarted running that code which then completes reload via the normal packet mechanism.

2.6 MECHANICAL MODULARITY

We have settled on a modular mechanical structure well matched to the modular logical structure of the system. This structure is important in that it allows easy construction of systems of varied size and permits repair of parts of a system while the rest of it continues to operate. The basic unit is a cooling module which houses either 1) a 16-slot bus complete with its own power supply, 2) a 24-slot bus without power, or 3) a power supply for such a 24-slot bus. These units, each with its own set of fans, sit on rails in a vertical tier in a rack, five of them filling a standard height rack. (The 14-processor system requires three racks.) Figure 1 shows how the cooling modules stack. Air flow is from back to front so that racks placed beside one another do not directly heat each other. A tilted pan at the bottom of each module separates the air flow between stacked modules, thus eliminating chimney effects. Cards plug in from the front and all device and coupler cables also connect on that side. An entire unit can be removed to the rear for repair or replacement of the bus, fans, etc. — all without disturbing operation of the remainder of the system.

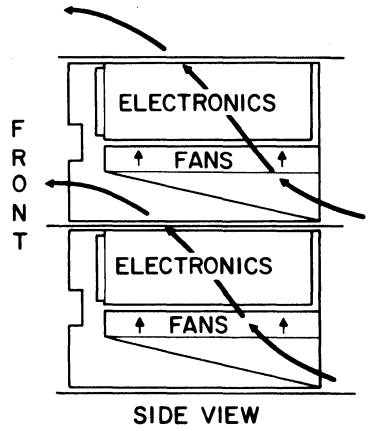


Figure 1
Mechanical Structure

3. THE TEST PROGRAM

The primary design objective of the test program is to exercise all of the hardware as intensively and extensively as possible, detecting all failures and reporting them precisely and comprehensibly. Extensive testing implies a wide variety of test modules; intensive testing implies permitting the entire computational power of the system to be focused on individual components at times. These objectives led to the selection of a system based on processes, analogous to a time-shared system's jobs. Processes are not tied to processors; a given process will switch rapidly from one processor to another. Nor is a process in general tied to a specific copy of code; like time-shared jobs, processes share a single copy of sections of pure procedure.

There are four types of processes: the "system" processes, including the clock, timeout, and type-out processes; the device-specific processes, which are tied to particular I/O devices, two processes per device; the "GART" (Get A Random Test) processes, which select a test at random from a table of tests to be performed; and a dummy process, whose sole purpose is to assure that there is always a runnable process.

Each GART test is designed to test a particular element or feature of the system. These range from standard processor and memory tests (the latter are also useful for checking bus couplers) to exercising the various bus coupler switches and maps. The I/O devices are kept busy by circulating various data through them.

4. WHERE WE STAND

Although the system uses Lockheed SUE processors, busses, memories, etc., we have so far designed and built nine BBN card types for the system: three coupler cards for each of the three bus types, a full-duplex memory channel card, a Host interface card

(which operates at speeds up to 1.5 megabit), transmit and receive modem cards, the pseudo-interrupt card and a clock card. These designs are virtually all finalized and many are in production (printed circuit or similar) form.

We are presently finishing the design of two other cards: the first of these is the parity checking card for the I/O bus described above under the discussion of parity. The second is a checksum/block-transfer card which flows a block of memory through itself computing a checksum as it goes. This is used to checksum critical code from time to time [3], to compute checksums for network end-to-end checking of messages, and other useful checking purposes. A transfer mode can be enabled so that it can also be used to move blocks of information about in memory (checksumming as it goes if desired). In addition we are presently embarking on modifications to the modem transmit and receive cards which will allow them to deal with 1.5 megabit lines and design of the special interface which monitors incoming inter-IMP lines watching for reload information as described above.

At present we are running several systems. Two small systems are being used for testing and debugging of the IMP program. These are sometimes run as separate single bus IMP systems which are connected together with our prototype 516 IMP into a three-node network. At other times the two busses are combined into a single system using a bus coupler. In this case one bus is used as a dual processor bus and the other as a combined memory and I/O bus. This system then works with the 516 IMP to form a 2-node net.

The growing prototype 14-processor system presently consists of three dual processor busses, two memory busses and one I/O bus. We have grown up to this system gradually but it now operates with sufficient reliability under stress (shaking of cables, margining power supplies, shuffling of cards, etc.) that we are presently in the process of building toward the full prototype (i.e., adding the 2nd I/O bus and the remaining four processor busses). By mid-1974 we hope to have two production copies of this large prototype working in the network. During 1974 we plan also to design satellite modem interface cards and to produce and deliver three moderate sized systems with satellite capability [4].

The basic IMP system program is up and running in multi-processor form, that is, with processors picking tasks up via the pseudo-interrupt system and using locks to prevent interfering accesses to resources. So far it has been run only with a two-processor system, but it will shortly be put on the larger prototype. The inner parts of the system, store and forward, Host, task, etc., seem solid. The work that remains is in implementing the system maintenance, monitoring, and debugging functions (i.e., system DDT, periodic status reports, etc.). This coding is about half done and needs finishing as well as debugging. The network error recovery code is ready for debugging. The special reliability code which keeps the system up when parts of the hardware fail is being designed.

Much work must be done in the present network to accommodate the advent of the new line of machines. For example, the whole reloading mechanism must be changed since one's neighbor may now be very different from one's self. The network must therefore be able to pass core load images packet-by-packet to an immediate neighbor of the machine needing reloading.

Our small IMP is built on a single logical bus (consisting of two separate physical busses connected by an extender) which combines memory, processor and I/O. This system embodies none of the special reliability stemming from multiple hardware copies but is the least expensive version available. Small reliable systems are another matter and require, in general, doubling the system to provide complete redundancy of parts to allow for any single failure. Such systems may prove to be one of the more significant outgrowths of this development effort.

REFERENCES

1. Heart, F.E. et al, *The Interface Message Processor for the ARPA Computer Network,* Proceedings AFIPS 1970 SJCC.
2. Heart, F.E. et al, *A New Minicomputer/ Multiprocessor for the ARPA Network,* Proceedings AFIPS 1973 NCC.
3. Crowther, W.R. et al, *Reliability Issues in the ARPA Network,* ACM Data Communications Symposium, Nov. 1973.
4. Butterfield, S.C. et al, *The Satellite IMP for the ARPA Network,* Seventh Hawaii Int. Conf. on System Sciences, Jan. 1974.

PLURIBUS — A RELIABLE MULTIPROCESSOR

by S. M. Ornstein, W. R. Crowther, M. F. Kraley,
R. D. Bressler, A. Michel, and F. E. Heart

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

November 1974

This paper was submitted for review by referees for the 1975
National Computer Conference.

PLURIBUS — A RELIABLE MULTIPROCESSOR*

*This work was supported by the Advanced Research Projects Agency (ARPA) under contracts DAHC15-69-C-0179 and F08606-73-C-0027.

by S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and F. E. Heart

Bolt Beranek and Newman Inc.

Cambridge, Massachusetts

INTRODUCTION

As computer technology has evolved, system architects have continually sought new ways to exploit the decreasing costs of system components. One approach has been to pull together collections of units into multiprocessor systems.[1] Usually the objectives have been to gain increased operating power through parallelism and/or to gain increased system reliability through redundancy.

In 1972, our group at Bolt Beranek and Newman started to design a new machine for use as a switching node (IMP) in the ARPA Network.[2,3] The machine was to be capable of high bandwidth, in order to handle the 1.5-megabaud data circuits which were then planned for the network. It was to have a high fanout to Host computers connected at a node. It was to come in all sizes (of processing power, memory, I/O) so that one could configure an

individual IMP to meet the requirements of its particular location in the network, and change that configuration easily should the requirements change. Most of all, it was to be reliable.

The family of machines we have produced which meets these goals has been named the Pluribus line. The machines are highly modular at several levels and have a minicomputer/multiprocessor architecture. Although the largest configuration we have put together so far contains only 13 processors, we believe there are no inherent problems with considerably larger systems. The structure and details of some of the hardware are described in earlier papers.[4,5] Familiarity with these papers will be helpful in understanding the present paper, which focuses on the issue of reliability. We believe that reliability will become an increasingly common concern as multiprocessors become more commonplace, and we believe that we have gained some interesting insights into the solution of this problem.

THE MULTIPROCESSOR ARCHITECTURE

A novel feature of our design is the consistent treatment of all processors as equal units, both in the hardware and in the software. There is no specialization of processors for particular system functions, and no assignment of priority among the processors, such as designating one as master. We chose to distribute among the processors not only the application job (the IMP job) but also the multiprocessor control and reliability jobs, treating all jobs uniformly. We view the processors as a resource used to

advance our algorithm; the identity of the processor performing a particular task is of no importance. Programs are written as for a single processor except that the algorithm includes interlocks necessary to insure multiprocessor sequentiality when required. The software of our machine consists of a single conventional program run by all processors. Each processor has its own local copy of about one quarter of this program and the remaining three quarters is in commonly accessible memory.

Hardware Structure

Reliability was a main concern in planning the hardware architecture. Although we tried to build the individual pieces solidly, our main goal was to provide hardware which could be exploited by the program to survive the failure of any individual component.

The hardware consists of busses joined together by special bus couplers which allow units on one bus to access those on another. Each bus, together with its own power supply and cooling, is mounted in its own modular unit, permitting flexible variation in the size and structure of systems. There are processor busses each of which contains two processors, each in turn with its own local 4K memory which stores frequently run and recovery-related code. There are memory busses to house the segments of a large memory common tc all the processors. Finally, there are I/O busses which house device controllers as well as

certain central resources such as system clocks and special (priority-ordered) task disbursers which replace the traditional priority interrupt system. About half of the machine consists of standard parts from the Lockheed SUE line; the remainder is of special design.

As emphasized in our initial paper,[4] we were fortunate to have a very specific job in mind as we designed the system. This enabled us to place specific bounds on the problems we sought to solve. For example, the proposed initial setting within a communications network means that outside entities (neighboring communications processors, Hosts, users, etc.) may help to notice that things are going wrong. It also means that recovery assistance is potentially available from the Network Control Center (NCC) through the network.[6,7] The system is designed generally to avoid reliance upon external help, but upon occasion such help is useful and therefore we have provided methods for permitting the system to be forcibly reloaded and restarted via the network.

Software Structure

The problem of building a packet-switching store-and-forward communications processor (the IMP) lends itself especially well to parallel solution since packets of data can be treated independently of one another. Other functions, such as routing computations, can also be performed in parallel.

The program is first divided into small pieces, called *strips*, each of which handles a particular aspect of the job. When a task needs to be performed, the name (number) of the appropriate strip is put on a queue of tasks to be run. Each processor, when it is not running a strip, repeatedly checks this queue. When a strip number appears on the queue, the next available processor will take it off the queue and execute the corresponding strip. We try to break the program into strips in such a way that a minimum of context saving is necessary.

The number assigned to each strip reflects the priority of the task it performs. When a processor checks the task queue, it takes the highest priority waiting job. Since all processors access this queue frequently, contention for it is very high. We therefore built a hardware device called the Pseudo Interrupt Device (PID) which serves as a task queue. A single instruction allows the highest priority task to be fetched and removed from the queue. Another instruction allows a new task to be put onto the queue. All contention is arbitrated by standard bus logic hardware.

The length of strips is governed by how long priority tasks can wait if all the processors are busy. The worst case arises when all processors have just begun the longest strip. In the IMP application, the most urgent tasks can afford to wait a maximum of 400 microseconds. Therefore, strips must not generally be

longer than that.

An inherent part of multiprocessor operation is the locking of critical resources to enforce sequentiality when necessary.[8] A load-and-clear operation provides our primitive locking facility. To avoid deadlocks, we priority-order our resources and arrange that the software not lock one resource when it has already locked another of lower or equal priority.

Status

During the early spring of 1974 a prototype 13-processor system was constructed. As this paper is being written (in the fall of 1974) two production copies have been constructed and are running. Each contains 13 processors, two memory busses, and two I/O busses. These machines have been connected intermittently into the ARPA Network for testing purposes and operational installation in the network is anticipated shortly. A single processor has been running on the network for an extended period in order to validate performance during routine operation. Three Satellite IMP configurations[9] are presently under construction as well as a non-IMP configuration designed to provide highly reliable pre-processing and forwarding of seismic data to processing and storage centers.

RELIABILITY GOALS

Since the term "reliable system" can have many different meanings, it is important to establish clearly just what we are and what we are

not trying to achieve. We are not trying to build a non-failing device (as in [10]); instead, we are trying to build a system which will recuperate automatically within seconds, or at most minutes, following a failure. Furthermore, we want the system to survive not only transient failures but also solid failures of any single component. In many cases (such as the IMP job) it is not necessary to operate continuously and perfectly; it suffices to operate correctly most of the time so long as outages are infrequent, kept brief, and fixed without human intervention.

How one copes with infrequent brief outages depends on what one is trying to do. For tasks not tightly coupled to real-time requirements (e.g., for many large numerical computations), a simple device is to choose checkpoints at which to record the state of the system so that one can always recover by restarting from the checkpoint just preceding an outage.[11,12] The IMP system happens to be embedded in a larger system which is quite forgiving. (This is not an uncommon situation.) Thus brief outages of a few seconds are tolerated easily, and outages of many seconds, while causing the particular node to become temporarily unusable, will not in general jeopardize operation of the network as a whole.

Occasionally, despite all efforts, the system will break so catastrophically that it will be unable to recover. Our goal is to reduce the probability of such total system failure to the probability of a multiple hardware failure. We do not try to

protect against all possible errors; some of our procedures will fail to detect errors whose probability of occurrence is sufficiently low. For example, all code is periodically checksummed using a 16-bit checksum. A failure that does not disturb the validity of the checksum may not be detected. We do not mind if a failure renders large sections of the machine unusable or inaccessible, providing enough remains to run the system. The presence of runnable hardware, however, is not sufficient to guarantee that operation will be resumed; in addition, the software must be able to survive the transients accompanying the failure and adapt to the remaining hardware. This may include combating and overcoming active failures (for example, when an element such as a processor goes berserk and repeatedly writes meaningless data into memory).

All code is presumed to be debugged -- i.e., all frequently occurring problems will have been fixed. On the other hand, we must be able to survive infrequent bugs even when they randomly destroy code, data structures, etc.

In order to avoid complete system failure, a failed component must be repaired or replaced before its backup also breaks. The system must therefore report all failures. The actual repair and/or replacement will of course be performed by humans, but this will generally take place long after the system has noted the failure and reconfigures itself to bypass the failed module. The ratio of mean-time-to-repair to mean-time-between-failures will

determine overall system reliability. It must also be possible to remove and replace any component while the system continued to run. Finally, the system should absorb repaired or newly introduced parts gracefully.

STRATEGIES

In order to understand our system it is convenient to consider the strategies used to achieve our goals in two parts which more or less parallel the traditional division into hardware and software. The first part provides hardware that will survive any single failure, even a solid one, in such a way as to leave a potentially runnable machine intact (potentially in that it may need resetting, reloading, etc.). The second part provides all of the facilities necessary to survive any and all transients stemming from the failure and to adapt to running in the new hardware configuration.

Appropriate Hardware

We have two basic strategies in providing the hardware. The first is to include extra copies of every vital hardware resource. The second is to provide sufficient isolation between the copies so that any single component failure will impair only one copy.

To increase effective bandwidth in multiprocessors, multiple copies of heavily utilized resources are normally provided. For reliability, however, *all* resources critical to running the algorithm are duplicated. Where possible the system utilizes

these extra resources to increase the bandwidth of the system.

It is not sufficient merely to provide duplicate copies of a particular resource; we must also be sure that the copies are not dependent on any common resource. Thus, for example, in addition to providing multiple memories, we also include logically independent, physically modular, multiple busses on which the memories are distributed. Each bus has its own power supply and cooling, and may be disconnected and removed from the racks for servicing while the rest of the machine continues to run.

All central system resources, such as the real time clock and the PIC, are duplicated on at least two separate I/O busses. All connections between bus pairs are provided by separate bus couplers so that a coupler failure can disable at most the two busses it is connecting.

Non-central resources, such as individual I/O interfaces, are generally less critical. Provision has been made, however, to connect important lines to two identical interface units (on separate I/O busses) either of which may be selected for use by the program.

To adapt to different hardware configurations, the software must be able to determine what hardware resources are available to it. We have made it convenient to search for and locate those resources which are present and determine the type and parameters of those which are found.

To allow for active failures, all bus couplers have a program-controllable switch that inhibits transactions via that coupler. Thus, a bus may be effectively "amputated" by turning off all couplers from that bus. This mechanism is protected from capricious use by requiring a particular data word (a password) to be stored in a control register of the bus coupler. Naturally an amputated processor is prevented from accessing these passwords.

Finally, although a common reset line is normally considered essential, we have avoided such a line since a single failure on its driver could jeopardize the entire system. There is thus no central point (not even a single power switch) where one can gain control of the entire system at once. Instead, we rely on resetting a section at a time using passwords.

Software Survival

With the above features, the Pluribus hardware can experience any single component failure and still present a runnable system. One must assume that as a consequence of a failure, the program may have been destroyed, the processors halted, and the hardware put in some hung state needing to be reset. We now investigate the means used to restore the algorithm to operation after a failure. The various techniques for doing this may be classified under three broad strategies: keep it simple, worry about redundancy, and use watchdog timers throughout.

Simplicity

It is always good to keep a system simple, for then one

has a fighting chance to make it work. We describe here three system constraints imposed in the name of simplicity.

First, as mentioned above, we insist that all processors be identical and equal: they are viewed only as resources used to advance the algorithm. Each should be able to do any system task; none should be singled out (except momentarily) for a particular function. The important thing is the algorithm. With this view it is clear that it is simplest if the algorithm is accessible to all processors of the system. A consequence of this is that the full power of the machine can be brought to bear on the part of the algorithm which is busiest at a given time.

One might argue that for some systems it is in fact simpler (or more efficient) to specialize processors to specific tasks. One could, in such a case, then duplicate each different type for reliability. With that approach, however, one must worry about the recovery of several different types of units, and all the possible interactions between them. We consider the recovery problem for a group of identical machines formidable enough.

One consequence of treating all processors equally is that a program can be debugged on a single machine up to the point where the multiple machine interaction matters. Once this has been done, we have found that processor interaction does not present a severe additional debugging problem. On the other hand, finding routine software bugs when a dozen machines are running

is a difficult problem.

A second characteristic of our system which arose from a desire to keep things simple is passivity. We use the terms active and passive to describe communication between subsystems in which the receiver is expected to put aside what it is doing and respond. The quicker the required response, the more active the interaction. In general, the more passive the communication, the simpler the receiver can be, because it can wait until a convenient time to process the communication. On the other hand the slower response may complicate things for the sender. We believe that there is a net gain in using more passive systems. An example of this is our decision to make the task disbursing mechanics (the PID) a passive device. Neither the hardware interfaces nor other processors tell a processor what to do; rather, processors ask the PID what should be done next. There are some costs to such a passive system. The resulting slower responsiveness has necessitated additional buffering in some of our interfaces. In addition, the program must regularly break from tasks being executed to check the PID for more important tasks.

The alternatives, however, are far worse. In a more active system, for example one which uses classical priority interrupts, it is difficult to decide which processor to switch to the new task. Furthermore, it is almost impossible to preserve the context of a processor[13] while making such a switch because of the interaction

with the resource interlocking system. The possibilities for deadlocks are frightening, and the general mechanism to resolve them cumbersome. With a passive system a processor finishes one task before requesting the next, thus guaranteeing that task switching occurs at a time when there is little context, e.g., no resources are locked.

Passive systems are more reliable for another reason: namely, the recovery mechanisms tend to be far simpler than those for active systems.

As a third example of simplicity we introduce the notion of a reliability subsystem. A reliability subsystem is a part of the overall system which is verified as a unit. A subsystem may include a related set of hardware, program, and/or data structures. The boundaries of these reliability subsystems are not necessarily related at all to the boundaries of the hardware subsystems (processors, busses, memories, etc.) described earlier. The entire system is broken into these subsystems, which verify one another in an orderly fashion.

The subsystems are cleanly bounded with well-defined interfaces. They are self-contained in that each includes a self-test mechanism and reset capability. They are isolated in that all communication between subsystems takes place passively via data structures. Complete interlocking is provided at the boundary of every subsystem so that the subsystems can operate asynchronously with respect to one another.

The monitoring of one subsystem by another is performed using timer modules, as discussed below. These timer modules guarantee that the self-test mechanism of each subsystem operates, and this in turn guarantees that the entire subsystem is operating properly.

Redundancy

Redundancy is simultaneously a blessing and a curse. It occurs in the hardware and the software, and in both control and data paths. We deliverately introduce redundancy to provide reliability and to promote efficiency, and it frequently occurs because it is a natural way to build things. On the other hand the mere existence of redundancy implies a possible disagreement between the versions of the information. Such inconsistencies usually lead to erroneous behavior, and often persist for long periods.

It was not until we adopted a strategy of systematically searching out and identifying all the redundancy in every subsystem that we succeeded in making the subsystems reliable. This process therefore constitutes one of our three basic strategies for constructing robust software.

We use the term redundancy here in a somewhat subtle sense, not only for cases in which the same information is stored in two places, but also when two stored pieces of information each imply a common fact although neither is necessarily sufficient to imply the other.

There are several methods of dealing with redundancy. The

first and best is to eliminate it, and always refer to a single copy of the information. When we choose not to eliminate it, we can check the redundancy and explicitly detect and correct any inconsistencies. It does not really matter how this is done as the system is recovering from a failure anyway. What is important is to resolve the inconsistency and keep the algorithm moving. Sometimes it is too difficult to test for inconsistency; then timers can be used as discussed in the next section.

Let us consider a few examples of redundancy to make these ideas more concrete.

- A buffer holding a message to be processed, and a pointer to the buffer on a "to be processed" queue -- if the buffer and queue are inconsistent, the buffer will not be processed. Each buffer has its own timer and if not processed in a reasonable time, it will be replaced on the queue.
- A device requesting a bus cycle, and a request capturing flip-flop in the bus arbiter -- if the arbiter and device disagree, the bus may hang. A timer resets the bus after one second of inactivity.
- One processor seeing a memory word at a particular system address and another seeing the same word at the same address -- The software watches for inconsistencies and when they occur declares the memory or one of the processors

unusable.

- The PID level used by a particular device and the device serviced in response to that level -- The PID level(s) used by each device are program-readable. A process periodically reads them and forces the tables driving the program's response to agree.

Timers

We have adopted a uniform structure for implementing a monitoring function between reliability subsystems based on watchdog timers. Consider a subsystem which is being monitored. We design such a subsystem to cycle with a characteristic time constant and insist that a complete self-consistency check be included within every cycle. Regular passage through this cycle therefore is sufficient indication of correct operation of the subsystem. If excessive time goes by without passage through the cycle, it implies that the subsystem has had a failure from which it has not been able to recover by itself. The mechanism for monitoring the cycle is a timer which is restarted by every passage through the cycle. We have both hardware and software timers ranging from five microseconds to two minutes in duration. Another subsystem can monitor this timer and take corrective action if it ever runs out. To avoid the necessity for subsystems to be aware of one another's internal structure, each subsystem includes a reset mechanism which may be externally activated. Thus corrective action consists

merely of invoking this reset. The reset algorithm is assumed to work although a particular incarnation in code may fail because it gets damaged. In such a case another subsystem (the code checksummer) will shortly repair the damage.

Note that we have introduced an active element into our otherwise totally passive system. These resets constitute the only active elements and furthermore are invoked only after a failure has occurred. This approach seems to provide for the maximum isolation between subsystems.

SYSTEM RELIABILITY STRUCTURE

In the previous section we described a mechanism whereby one subsystem can monitor another. Our system consists of a chain of subsystems in which each subsystem monitors the next member of the chain. Figure 1 and Table I show this structure in the system we have built for the IMP. An efficient way to build
ig. 1 such a chain is to have lower subsystems provide and guarantee some important environmental feature used by higher level systems. For example, a low level in our chain guarantees the integrity of code for higher levels which thus assume the correctness of code. Such a system is vulnerable only at its bottom. (We are assuming here that we have runnable hardware although it may be in a bad state, requiring resetting.) The code tester level (see Figure 1) serves three functions: first, it checksums all low level code (including itself); second, it insures that control

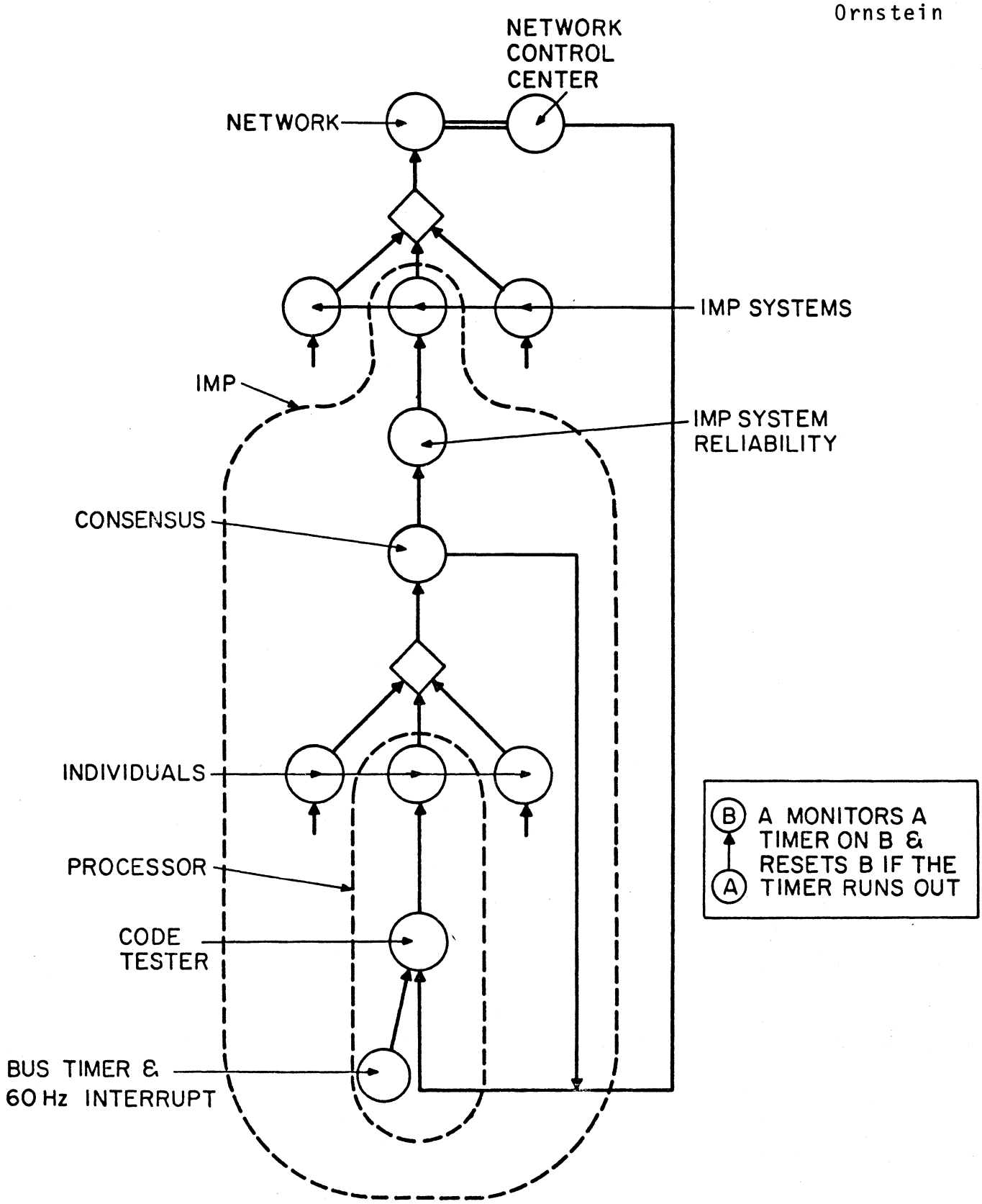


FIGURE 1 RELIABILITY STRUCTURE

is operating properly, i.e., that all subsystems are receiving a share of the processors' attention; third, it guarantees that locks do not hang up. It thus guarantees the most basic features for all higher levels. These will, in turn, provide further environmental features, such as a list of working memory areas, I/O devices, etc., to still higher levels. The method by which the code tester subsystem itself is monitored and reset will be discussed shortly.

Table I

Major Subsystems and their Functions

IMP SYSTEM: Watches network behavior - will not cooperate with irresponsible network behavior.

IMP SYSTEM RELIABILITY: Watches IMP SYSTEM (data structures mostly).

CONSENSUS: Watches IMP SYSTEM RELIABILITY, verifies all Common Memory Code (via checksum), watches each processor, finds all usable hardware resources (interfaces, PIDs, memory, processors, etc.), tests each and creates a table of good ones. Makes spare copies of code.

INDIVIDUAL: Watches CONSENSUS, finds all memory and processors it considers usable, determines where the Consensus is communicating and tries to join it.

CODE TESTER: Watches INDIVIDUAL, verifies all Local Memory Code (via a checksum), guarantees control and lock mechanisms.

BUS TIMER + 60Hz INTERRUPT: Watches CODE TESTER, guarantees bus activity.

The mechanisms we have described ensure that the separate processor subsystems have a satisfactory local environment in which to work. Before they can work together to run the main system it is necessary that a common environment be established for all processors. We call the process of reaching an agreement about this environment "forming a consensus", and we dub the group of agreeing processors the Consensus. The work done by the Consensus is in fact performed by individual processors communicating via common memory, but the coordination and discipline imposed on Consensus members make them behave like a single logical entity. An example of a task requiring consensus is the identification of usable common memory and the assignment of functions (code, variables, buffers, etc.) to particular pages. The members of the Consensus will not in general agree in their view of the environment, as for example when a broken bus coupler blinds one member to a segment of common memory. In this case the Consensus, including the processor with the broken coupler, will agree to run the main system without that processor.

The Consensus maintains a timer for every processor in the system, whether or not the processor is working. The Consensus will count down these timers in order to eliminate uncooperative or dead processors. In order to join the Consensus, a processor need merely register its desire to join by holding off its timer. Within the individual processors it is the code tester subsystem which holds off the timer.

The Consensus, then, acting as a group, provides the monitoring mechanism for the individuals as shown by the feedback monitoring path in Figure 1. This monitoring mechanism run by the Consensus includes the usual reset capability which in this case means reloading the individual's local memory and restarting the processor. Since all of the processors have identical memories, reloading is not difficult. We provide (password protected) paths whereby any processor can reset, reload, and restart any other processor. This reliance on the Consensus is indeed vulnerable to a simultaneous transient failure of all processors. However, the Network Control Center has access to these same reset and reload facilities and these enable it to perform the reload function remotely (a path also shown in the figure).

Thus the Consensus and/or Network Control Center are the ultimate guarantors of the lowest level subsystem. While this process is sufficient it is sometimes slow. For many cases in which the Consensus is disabled (as for example when all of the

processors halt), a simpler reset without reloading will suffice. For this reason we have provided a simpler and more immediate (if redundant) mechanism in each processor for resetting the control and lock systems. We implement this mechanism in software with the assistance of a 60Hz interrupt and a one-second timer on the bus. Together these provide a somewhat vulnerable but much quicker alternative to the more ponderous NCC/Consensus resets.

There is a problem about what area of common memory the processors should use in which to form the Consensus, since failures may make any predetermined system address inaccessible. To allow for this, sufficient communication is maintained in all pages of common memory to reach agreement both as to which processors are in the Consensus and where further communication is to take place.

To protect itself from broken processors, the Consensus amputates all processors which do not succeed in joining it. There is a conflict between this need to protect itself and the need to admit new or healed processors into the Consensus. The amputation barrier is therfore lowered for a brief period each time the Consensus tries to restart a processor. This restart is in fact the reset based on the timer held off by the code tester subsystem, as discussed above. In the case of certain active failures, even this brief relaxation may cause trouble. In these cases the Consensus will decide to keep the barrier up continuously.

Certain active failures may prevent the formation of a consensus. In such a situation each processor will behave as if it were a Consensus (of one) and will try to amputate all other processors. At the point when the actively failing component is amputated, the remaining processors will be able to form a consensus. From this point the system behaves as described above.

Further up in the figure there is another joining of independent units, namely IMPs joining to form the network. The analogy here is incomplete because the ARPA Network was not built with these concepts in mind. There is collective behavior, e.g., routing, and individual behavior which accepts collective decisions only after they pass reasonability tests. However, the reliability features of the network are concentrated in the Network Control Center, which depends on the continual presence of human operators for successful operation. It is correspondingly powerful, resourceful, and erratic in its behavior.

SOME EXAMPLES OF FAILURES

In order to explain in more practical terms some of the reliability mechanisms, we now discuss a number of specific failures and describe the methods which detect and repair the resulting damage. In each case, we focus on the component that failed and the particular mechanism that takes care of that failure. Derivative failures may well take place, and other mechanisms will handle these, since all mechanisms operate all the time.

These examples are set in the context of the IMP application and the severity of their direct consequences rated on the following scale:

1. Momentary slowdown - no data loss
2. Loss of data (a network message)
3. Temporary loss of some IMP function (a network link)
4. Momentary total IMP outage with local self-recovery
5. Outage requiring reloading via the network
6 Failure requiring human insight for debugging.

Example 1. Suppose first that a bus coupler experiences a transient failure on a single reference to common memory, which leaves one word of common memory with the wrong contents but correct parity. Suppose further that the failure is subtle, in the sense that there is no obvious ill effect on processor control, like halting or looping, which will be caught by lower level mechanisms. We will focus first on examples which cause minimal disruption and where detection and gentle recovery are the primary concerns. We consider four examples of transient memory failures:

Example 1.a Suppose that a word of text in one of the messages we are deliverying becomes smashed. There is a checksum on all messages and the network will notice at one of its checkpoints that the message has gone bad. The source will be prompted to send a new copy. (Severity 2)

Example 1.b Near the heart of our system is a queue of unused buffers called the free list. Suppose the failure is in the structure of this queue. The system explicitly tests for both a looped queue and wrong things on the queue. A more subtle form of error occurs when some buffers which should be on the queue are missing from it. Our system is designed so that a buffer should be removed from the free list for at most two minutes at a time. A timer is maintained on each buffer, which is restarted whenever the buffer returns to the free list. Should any timer ever run out, its buffer is forced back onto the free list. The result of this failure will be a degradation of system performance as it attempts to run with fewer buffers for a short while, followed by complete recovery within two minutes. The IMP will stay up and no messages will be lost. (Severity 1)

Example 1.c Suppose that one of the locks on a resource is broken so that it wrongly locks the resource. Any subsystem which tries to use the resource will put a processor into a tight loop waiting for the resource to become free. In about 1/15 sec. this will cause the processor's timer, driven off its 60Hz clock interrupt, to run out. Upon investigation, the program will notice that the subsystem is waiting for a locked resource, and arbitrarily unlock it. Aside from the 1/15 sec. pause, the system will be unaffected by the transient. (Compare the simplicity of this scheme with [14].) (Severity 1)

Example 1.d Suppose now that a failure strikes common memory holding code, and that the trouble is subtle -- either the code is not run often or the change has no immediate drastic effect. In a few seconds the processors will begin to notice that the checksum on that piece of code is bad and stop running it. Shortly the whole Consensus will agree, and will switch over to use the memory holding the spare copy of that code. Unless the broken code has already caused some other trouble, the problem is thereby fixed, with only momentary slowdown. (Severity 1)

Example 2. Suppose a processor fails by suddenly and permanently stopping. The immediate effect will be that some task will be left half done, with a high probability that some resource is locked. This looks to the system like a data failure, as in examples 1.a, 1.b, and 1.c above. The recovery will be identical. In a few seconds the Consensus will notice that the processor has dropped out and processor recovery logic will be invoked. Since the processor is solidly broken the recovery will be unsuccessful, and the system will settle into a mode where every so often recovery is retried. Eventually a repairman will fix the processor, at which time recovery will proceed and the processor will rejoin Consensus. It is hard to predict whether the IMP system will momentarily go down because of the failure; experience indicates that it usually stays up, but our experience is limited to lightly loaded machines. (Severity 2-4)

Example 3. Suppose a power supply for a processor bus breaks. This is similar to the failing processor described above except that both processors on the bus are affected and the processors are given a hardware warning sufficiently far in advance that they can halt cleanly. The system will surely stay up through this failure. (Severity 1)

Example 4. Now consider a case in which some page of common memory ceases to answer when referenced. Each processor will get a hardware trap when it tries to use that memory, forcing it directly to the code which routinely verifies the environment. As a result, the failing memory will be deleted from the memory list by the Consensus and another module will be pressed into service to take its place.

If the failed page contained code, a spare copy will normally be available and a new spare copy will be made if possible. If it contained data, an unused page will be pressed into service. In either case, the system will be reinitialized, momentarily bringing the IMP system down. If the failed page contained the Consensus communication area, a new Consensus must be formed and thus recovery will take a little longer. (Severity 4)

Example 5. Let us now consider a failure of the PID. Suppose that the PID reports a task not previously set. When invoked, each strip checks to make sure that it is reasonable for the strip to be run. If not, another task is sought. Suppose instead that the PID "drops" a task. A special process periodically sets all PID flags independent of what needs to be done. This causes no harm, because

superfluous tasks will be ignored (as described above), and serves to pick up such dropped tasks. Thus we have both a consistency check on redundant information and a timer built into our use of the PID. If a PID fails solidly, another PID will be switched in to operate the system. Transient failures cause slowdown; switch-over may momentarily bring down the IMP system. (Severity 1, 4)

All of this leads to a slightly different image of the PID. Instead of being the central task disburser, with all processors relying on it to tell them what to do, the PID is a guide, suggesting to processors that if they look in a certain place, they will probably find some useful work to do. The system would in fact run without a PID, albeit much more slowly and inefficiently.

Example 6. Suppose a halt instruction somehow gets planted in common memory and that all processors execute it and stop. There is thus no Consensus left to come to the rescue. Furthermore, 60Hz interrupts are ineffective in a halted processor. After one second of inactivity, the bus arbiter timer will reset the processors, making them once more eligible for 60Hz interrupts which will restart them. Before the broken code is run, it will be checksummed, the discrepancy found, and a spare copy used. (Severity 2-4)

Example 7. Let us consider now what happens when, in common memory, an end test for a storing loop is destroyed, causing each processor to wipe out its 60Hz interrupt code in local memory. In this case not only are there no processors left to help, but the 60Hz interrupt will not help either, since the interrupt code itself is

broken. This is a case in which the machine is incapable of rescuing itself and will go off the network as a working node. When the Network Control Center notices that the IMP is no longer up, it will commence an external reload, restoring the IMP to operation. (Severity 5)

Example 8. Consider the case of a processor whose hardware is solidly broken such that it repeatedly stores a zero into a location in common memory. Mechanisms described above will repeatedly fix the changed location, but it is desirable to eliminate the continuing presence of this disrupting influence. The Consensus will notice that one of its number has dropped out and will endeavor to help the errant processor. After a few tries, a longer timer will run out, and the Consensus will take a more drastic action: final amputation. In this case there will be a rather lengthy IMP outage but the system will recover without external help. (Severity 4)

Example 9. One failure from which there is no recovery, either automatic or remote, is a program which impersonates normal behavior but is still somehow incorrect. That is, it holds off the right timers, has a valid checksum, and simulates enough normal behavior so that higher levels (e.g., the NCC) are satisfied. For example, if it were not for the fact that the NCC explicitly checks the version vumber of the program running in each IMP, a previous, compatible, but obsolete version of the program would exhibit this behavior. (Severity 6)

Example 10. Another class of failures which is hard to isolate and deal with is low-frequency intermittents. Consider the case of a single processor which is broken such that its indexed shift instruction performs incorrectly. Since this instruction only occurs in some infrequently executed procedures, the failure only manifests itself, on the average, once every period t. If t is large, for instance one year, then we can safely disregard the error, since its frequency is in the range of other failures over which we have no control. If t is small, say 100 milliseconds, then the Consensus will isolate the bad processor and excise it. At some intermediate frequency, however, the Consensus will fail to correlate successive failures and will instead treat each as a separate transient. The system will repeatedly fail and recover until some human intervenes. (Severity 6)

RESULTS AND CONCLUSIONS

Some strategies and techniques for building a reliable multiprocessor have been described above. We have, in fact, actually built and programmed such a machine using these strategies. We have found this machine straightforward to debug, both in hardware and software. Furthermore, the system continues to operate when individual power supplies are turned off, when memory locations are altered, when cables and cards are torn out, and through a variety of other failures. We have yet to establish field performance (which must be measured both in rate of recoverable

incidents and in rate of unrecoverable failures), but we expect to start gathering this information shortly.

We believe there are many important problems in the world today which could benefit from the principles described here. While we have discussed these principles in terms of a specific application (the IMP), most of the concepts are application independent. We have been able to separate the application code from the reliability subsystems intact in another application of the Pluribus machine.

ACKNOWLEDGEMENTS

Many people in addition to the authors have contributed to the ideas described herein, notably Benjamin Barker, John Robinson, David Walden, John McQuillan, and William Mann. In addition, there is a long list of those who helped to bring these machines into existence. Foremost among these are Martin Thrope, David Katsuki and Steven Jeske. The work reported here would not have been possible without the continued support of the ARPA/IPT office. Finally, a word of thanks to Robert Brooks and Julie Moore, who helped to prepare the manuscript.

REFERENCES

1. W. B. Riley, "Minicomputer Networks -- A Challenge to Maxicomputers?" Electronics, March 29, 1971, pp. 56-62

2. F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 551-567; also in Advances in Computer Communications, W. W.Chu (ed.), Artech House Inc., 1974, pp. 300-316

3. L. G. Roberts and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 543-549.

4. F. E. Heart, S. M. Ornstein, W. R. Crowther, and W. B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 529-537; also in Selected Papers: International Advanced Study Institute, Computer Communication Networks, R. L. Grimsdale and F. F. Kuo (eds.) University of Sussex, Brighton, England, September 1973; also in Advances in Computer Communications, W. W. Chu (ed.), Artech House Inc., 1974, pp. 329-337.

5. S. M. Ornstein, W. B. Barker, R. D. Bressler, W. R. Crowther, F. E. Heart, M. F. Kraley, A. Michel, and M. J. Thrope, "The BBN Multiprocessor," Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu,

Hawaii, January 1974, Computer Nets Supplement, pp. 92-95.

6. W. R. Crowther, J. M. McQuillan, and D. C. Walden, "Reliability Issues in the ARPA Network, " Proceedings of the ACM/IEEE Third Data Communications Symposium, November 1973, pp. 159-160.

7. A. A. McKenzie, B. P. Cosell, J. M. McQuillan, and M. J. Thrope, "The Network Control Center for the ARPA Network," Proceedings of the First International Conference on Computer Communication, Washington, D.C., October 1972, pp. 185-191.

8. E. W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys, Academic Press, London and New York 1968, pp. 43-112.

9. S. C. Butterfield, R. D. Rettberg, and D. C. Walden, "The Satellite IMP for the ARPA Network, " Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 1974, Computer Nets Supplement, pp. 70-73.

10. A. L. Hopkins, Jr., "A Fault-Tolerant Information Processing Concept for Space Vehicles," IEEE Transactions on Computers, Volume C-20, Number 11, November 1971, pp. 1394-1403.

11. A. Avizienes, G. C. Gilley, F. P. Mathur, D. A. Rennels, I. A. Rohr, and D. K. Rubin, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and

Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers, Volume C-20, Number 11, November 1971, pp. 1312-1321.

12. IBM Corporation, "OS Advanced Checkpoint/Restart," IBM Manual GC28-6708.

13. R. J. Gountanis and N. L. Viss, "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System." Proceedings of the IEEE, Vol. 54, No. 12, December 1966, pp. 1812-1819.

14. L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," Communication of the ACM, Volume 17, Number 8, August 1974, pp. 453-455.