AD-A101 412   AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB OH   F/G 9/2
A CONTINUOUSLY RECONFIGURING MULTI-MICROPROCESSOR FLIGHT CONTRO--ETC(U)
MAY 81   S J LARIMER, S L MAHER
UNCLASSIFIED   AFWAL-TR-81-3070   NL

1 OF 2
AD
A101 412

AFWAL-TR-81-3070

A CONTINUOUSLY RECONFIGURING MULTI-MICROPROCESSOR FLIGHT CONTROL SYSTEM

Stanley J. Larimer, Captain, USAF
Scott L. Maher, First Lieutenant, USAF

MAY 1981

Final Report for Period August 1979 to March 1981

Approved for public release; distribution unlimited.

FLIGHT DYNAMICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Scott L Maher

SCOTT L. MAHER, 1Lt, USAF
Project Engineer
Control Data Group
Control Systems Development Branch

Stanley J Larimer

STANLEY J. LARIMER, Capt, USAF
Project Engineer
Control Analysis Group
Control Dynamics Branch

Evard H Flinn

EVARD H. FLINN, Chief
Control Systems Development Branch
Flight Control Division

R.O. Anderson

RONALD O. ANDERSON, Chief
Control Dynamics Branch
Flight Control Division

FOR THE COMMANDER

Robert C Ettinger

ROBERT C. ETTINGER, Colonel, USAF
Chief, Flight Control Division

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify AFWAL/FIGC, W-PAFB, OH 45433 to help us maintain a current mailing list".

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

AIR FORCE/56780/23 June 1981 — 510

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFWAL-TR-81-3070	2. GOVT ACCESSION NO. AD-A102412	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CONTINUOUSLY RECONFIGURING MULTI-MICROPROCESSOR FLIGHT CONTROL SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Final Report 1 Aug 1979 — 30 Apr 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stanley J. Larimer, Captain, USAF Scott L. Maher, First Lieutenant, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Flight Dynamics Laboratory (AFWAL/FIGC) Air Force Wright Aeronautical Laboratories (AFSC) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element: 62201F Project: 2403 Task: 02 Work Unit: 44
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE May 1981
		13. NUMBER OF PAGES 148
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Digital Control Systems
Multi-Microprocessor
Flight Control Systems
Distributed Control
Reconfiguration
Distributed Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Recent research at the US Air Force Wright Aeronautical Laboratories (Flight Dynamics Lab) has resulted in the development of a promising microprocessor-based flight control system design. This system is characterized by a collection of cooperatively autonomous distributed microcomputers interconnected by an arbitrary number of common serial multiplex busses. Each processor in the system independently determines its assignments using a simple algorithm that

(continued)

Cont'd

dynamically redistributes system functions from processor to processor in a never-ending process of reconfiguration. This approach offers several benefits in terms of system reliability, and the architecture in general incorporates many state-of-the-art features which promise improved system throughput, expandability, and above all, ease of programming.

The Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMmFCS) represents a major data point in multiprocessor control system research. Promising ideas from a variety of references have been included and integrated in its design. Its laboratory implementation provides a demonstration of these ideas for improving throughput, reliability, and ease of programming in flight control applications.

FOREWORD

The research described in this report was performed in-house at the AFWAL Flight Dynamics Laboratory during the period from September 1979 to March 1981. It is the result of a joint effort between members of the Control Systems Development Branch (AFWAL/FIGL) and the Control Dynamics Branch (AFWAL/FIGC) of the Flight Control Division.

Work began in this area in late 1978 when the Control Systems Development Branch initiated a work unit called "Multi-Microprocessor Control Elements" (24030244). During this time, Lt. James E. May, Lt. Scott L. Maher, John Houtz, and Capt. Larry Tessler laid the foundation for a study of how growing microprocessor technology could be applied to the problems of modern flight control. It was decided to design and build some form of fully distributed microprocessor-based flight control system in order to explore the potential problems and benefits in great detail.

In September 1979, Lt. Scott Maher took over as Principal Investigator and Capt. Stan Larimer joined the program as the Associate Investigator from the Control Dynamics Branch. In the months that followed, Maher and Larimer developed what has come to be known as the "Continuously Reconfiguring Multi-Microprocessor Flight Control System" architecture. Steve Coates, Richard Gallivan, and Tom Molnar also provided invaluable input to the research during this phase.

During the summer of 1980, Mr. Harry Snowball (Control Data Group Leader) and Mr. Evard Flinn (Control Systems Development Branch Chief) decided to intensify efforts to develop this new architecture. They hired four new engineers including Bill Rollison, Ray Bortner, Mark Mears, and Stan Pruett to work on the project. They also assigned technician SSgt. Jeff Lyons and co-op students Dan Thompson, Russ Blake, and Bob Molnar to assist with the R&D activites. At the same time, Mr. Dave Bowser (Control Analysis Group Leader) and Mr. Ron Anderson (Control Dynamics Branch Chief) increased FIGC support by assigning Lt. Allan Ballenger to act as an architecture and control law consultant on the project. In 1981, Lt. Jack Crotty joined the program as a software designer thereby completing the CRMmFCS team.

The authors would like to thank these individuals for their many contributions to the success of this program. Bill Rollison was responsible for the design, development, construction, and testing of the transmitter and receiver circuitry. Ray Bortner designed the real-world interface processor and helped to develop the CRMmFCS control laws and corresponding aircraft simulation. Mark Mears was responsible for the hardware and software design of the entire data collection and processing network for the CRMmFCS laboratory implementation. Stan Pruett invented a unique five-port RS-232 communications controller which allows all systems in the laboratory to communicate with each other. He also assisted in the development of CRMmFCS processing module software and programmed the TRS-80 to act as controller for the entire laboratory system.

Lt. Allan Ballenger was responsible for development of the real-world plant simulation and overall control law design. He also organized a national workshop on multi-processor flight control architectures which provided a vital link with others working in the field. Tom Molnar served as a technical consultant and work unit monitor for the program. Lt. Jack Crotty was responsible for the development of all CRMmFCS software. Dan Thompson provided the original design for much of the TMS-9900 software used to implement the CRMmFCS operating system.

Rick Gallivan was responsible for the development of a real-world simulation on a Motorola 68000 microprocessor and provided logistics support for the program. Steve Coates handled the construction of the entire laboratory setup and helped to develop a real-time graphics display for the pilot interface. Jeff Lyons designed and constructed the first working bus termination circuit. Russ Blake and Bob Molnar provided drafting and breadboarding support for many of the laboratory components.

The authors would also like to express special thanks to Art Eastman and Dave Dawson for their outstanding work in designing and producing the many figures which appear throughout this document. Carl Weatherholt, Rudy Chapski, and Bill Adams also provided invaluable support in the laboratory. Many thanks also go to Pam Larimer, Jan Robinson, and Dave Bowser for their assistance in the preparation of this report.

Finally, the authors wish to express their appreciation to Mr. Vernon Hoehne, Lt. Allan Ballenger, and Tom Molnar for their considerable efforts in reviewing the technical report.

This report covers work performed from September 1979 through March 1981. It was submitted by the authors in May 1981.

TABLE OF CONTENTS

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (concluded)

SECTION I
INTRODUCTION

The use of microprocessors in flight control applications is a subject which has received much attention in recent years. Microprocessor technology is growing rapidly and there is a strong desire to take advantage of it. This report presents the results of several years of research performed at the Flight Dynamics Laboratory into how microprocessors might best be used for flight control applications.

Microprocessors appear to have two major areas of application to the flight control problem. These may be termed "the low end" and "the high end." In the low end approach, microprocessors are distributed around the system in a dedicated fashion wherever a small amount of processing power is needed. In this mode, microprocessors are relegated to the role of "smart" peripherals to some central computer system. This application has been demonstrated with considerable success in many currently flying aircraft. By performing many of the repetitive, time intensive functions such as keyboard monitoring, sensor preprocessing, display generation, and inner-loop control, microprocessors can relieve the central computer of much of its computational load so that it can concentrate on what it does best: number crunching, system management, and outer-loop control. Since the low end application has been demonstrated in many operational systems and its utility is generally unquestioned, it will not be discussed further in this report.

The "high end" application for microprocessors is a much newer field of research. It is concerned with how to use microprocessors as a distributed, multi-processing replacement for the central computer itself. Since the goal of this research program was to investigate the potential of microprocessors for flight control and not to design a working model for near-term application (where a more conservative approach would be necessary), it was decided to attempt the most ambitious (and most promising) application of microprocessor technology: "a fully decentralized, continuously reconfiguring, self-healing, adaptive, pooled microprocessor-based flight control system." The result was the development of an architecture known as the "Continuously Reconfiguring Multi-Microprocessor Flight Control System" (CRMmFCS).

This report presents the results of research to date in the development of the CRMmFCS. Section II gives an overview of the architecture and the philosophy behind it. Section III presents a detailed discussion of the hardware aspects of the architecture while Section IV does the same for software. Both sections represent virtually stand-alone discussions of their respective topics at a level which is as thorough as possible without sacrificing readability. Technical details which are of use to the reader only after a complete study of the architecture have been removed to the appendicies. Finally, Section V describes the actual laboratory implementation and test procedure which will be used to demonstrate the architecture. The results of these tests will be published in a subsequent technical report.

SECTION II
OVERVIEW

A. OVERVIEW OF THE CRMmFCS ARCHITECTURE

The CRMmFCS design centers around a system of autonomous microprocessors connected by a common set of serial multiplex busses. These processors operate in a pooled configuration where any processor can perform any task at any time. Furthermore, task assignments are continuously redistributed among all processors in a never-ending process of reconfiguration. If a processor fails in the system, it is simply left out of the next reconfiguration cycle and the system continues to operate as if nothing has happened. All of this is accomplished without use of a central controller.

A diagram of the architecture is shown in Figure 1. In the figure, six processing modules are shown connected to a set of four common data busses. Each data bus consists of one clock and one data line and information is transferred between processors using a simple serial multiplexing scheme.

Processors in the system compete for access to the busses without central traffic control using a technique called "transparent contention." Transparent contention is a scheme which allows any processor to talk on any bus at any time. In the event of a "collision" between two transmissions only one message survives while the remaining one is automatically retransmitted as soon as the bus is free. Transparent contention provides one hundred percent efficient bus utilization, eliminates most communications overhead, and completely avoids the need for a central controller. It is discussed in detail in Section III.

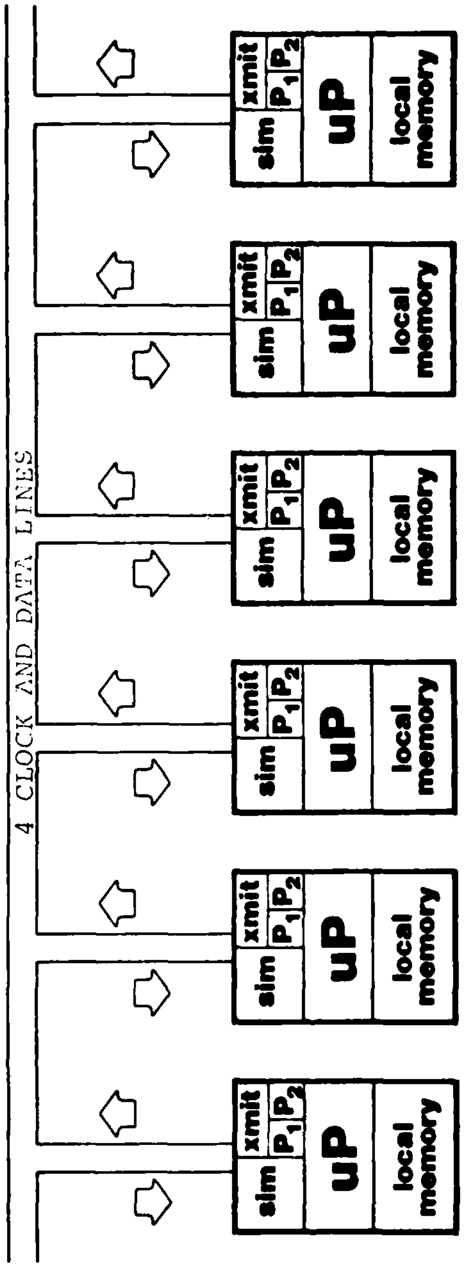


Figure 1. Major CRMmFCS Architecture Components

In addition to competing for the bus, processors also compete for the right to perform tasks in the system. During every time frame all processors "volunteer" for the tasks to be done in the following frame. These tasks are then divided by mutual agreement among all functioning processors in the system, again without needing a central controller.

Because tasks to be performed are redistributed at the beginning of each frame, the task any particular processor performs is changing all the time. The system is said to be "continuously reconfiguring." Continuous reconfiguration has a number of important advantages over other approaches. These include automatic recovery in the event of a failure, constant spare checkout because no unit acts as a spare all the time, latent fault protection, and zero reconfiguration delay. A complete discussion of continuous reconfiguration is presented in Section IV.

Finally, although processors communicate only via simple serial busses, the architecture is configured so that they appear to share a single common memory as shown in Figure 2. This virtual common memory contains all information available about the state and environment of the entire system. Processors obtain the information needed for any task from the virtual common memory and place their results back there for use by other processors. Complete details on how a set of serial busses can be made to act like a common memory are presented in Section III.

B. DESIGN PHILOSOPHY

Before beginning a detailed discussion of the Continuously Reconfiguring Multi-Microprocessor Flight

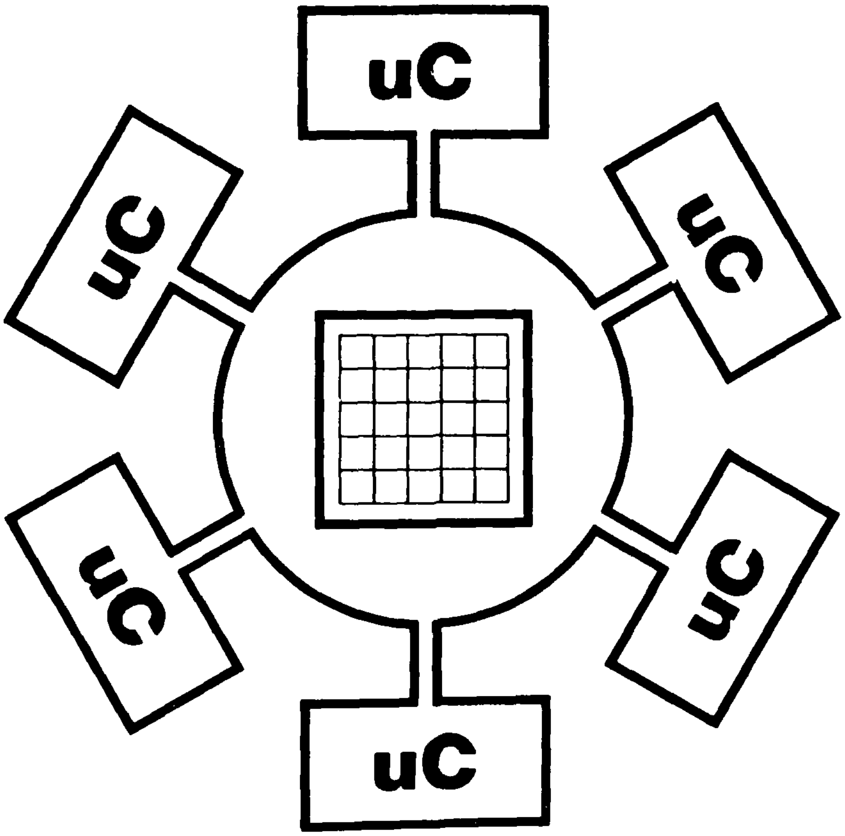


Figure 2. Virtual Common Memory

Control System (CRMmFCS) it is desirable to briefly discuss the design goals and philosophy which lead to this architecture. The original objective of this in-house effort was to develop an Air Force understanding and capability in the area of multi- microprocessor flight control systems. It was determined that a high risk - high payoff approach could be taken in an effort to advance the state-of-the-art while achieving the original objective. The approach taken was to trade off low cost hardware for simplified software and to distribute system control to its extreme in order to study the extent to which its potential advantages could be achieved. Other goals were to reduce overall hardware, software, and life cycle costs of flight control systems while maintaining high reliability and fault tolerance. Design considerations also included expandability for integrated control applications and reconfigurability to meet future self-healing requirements.

C. EVOLUTION OF CONTINUOUS RECONFIGURATION

Figure 3 shows a breakdown of some of the possibilities that exist for implementing digital control systems in general. Starting at the top of the figure, it may be seen that digital control systems can be broken down into either uni-processor of multi-processor systems. The distinction here is not so much whether there is one or more than one processor in the system, but rather whether there is more than one processor performing different functions. A system with, say, four processors performing identical functions for redundancy would, for the purposes of this report, be considered a uni-processor system since its effective throughput is that of a single processor.

In this study, the multi-processor approach was chosen for two reasons. First, since state-of-the-art

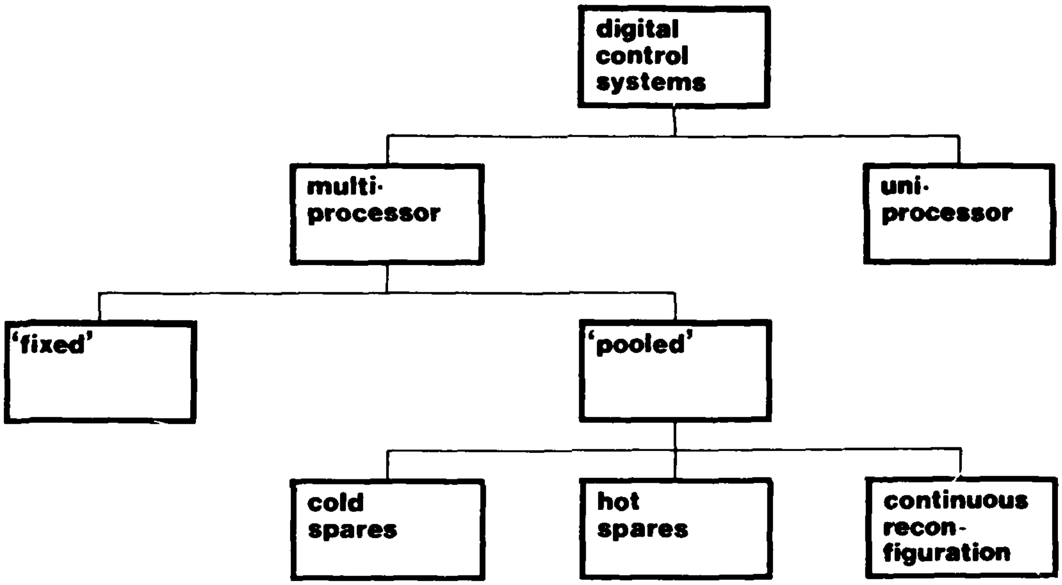


Figure 3. Breakdown of Possible Architectures

microprocessors have somewhat lower throughput than their mini-computer counterparts, it is unlikely that a single microprocessor would be able to handle the workload of a modern flight control system. Since we are constrained to use microprocessors by the goals of this investigation, it would seem that a multiprocessor architecture is mandatory for flight control applications. Of course, with the rate at which the field is developing, there may soon be microprocessors that can handle the required workload in a uni-processor configuration. But with the development of ever more sophisticated estimation, parameter identification, and self-optimization algorithms and increasingly demanding command and control functions, the required workload may go up at an even faster rate. Since a multi-processor architecture will always be able to improve on the throughput of a uniprocessor architecture of the same state-of-the-art, and since the only thing growing faster than computer technology is the size of the problems to be solved, there will always be a need for multi-processor configurations. The need for better methods to construct such architectures is the second reason why the multi-microprocessor approach was selected for this investigation.

Given then that a multi-microprocessor system is to be implemented, there are two possible ways in which their functions can be assigned. As shown in Figure 3, these ways are "fixed" assignment of processing resources, where the function of each processor is permanently assigned, and "pooled" processing resources, where processors are dynamically assigned to each function as the needs arise. The fixed assignment is inherently simpler to implement and is adequate for many applications. However, in systems requiring great reliability and minimum hardware, the pooled approach offers distinct advantages.

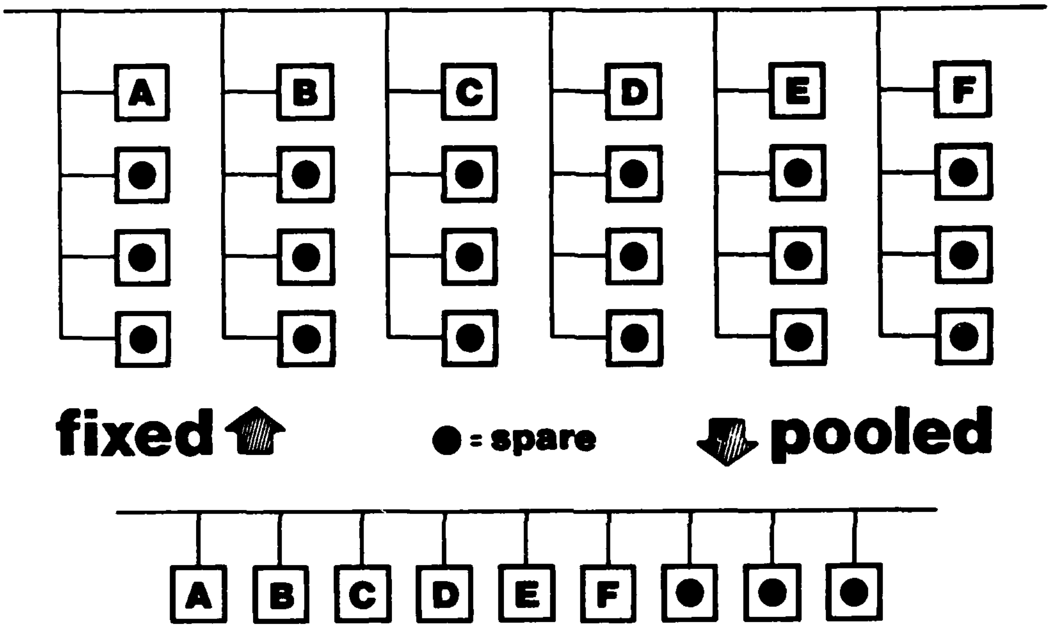


Figure 4. Fixed vs. Pooled Designs with Equal Redundancy

Figure 4 demonstrates the advantages of the pooled approach in systems requiring a large number of processing tasks. The system shown requires six different tasks to be performed concurrently and that a quad level of redundancy be maintained. A fixed assignment implementation of this system (Figure 4a) requires that six processors be permanently assigned to the six tasks and that three spares be permanently assigned to each processor. The net result is a 24 processor system. Figure 4b shows an equivalent system using a pooled architecture. This system still requires at least six processors to perform the six concurrent tasks but the number of spares is substantialy reduced. This is because, since any processor can be dynamically assigned to any task, the three spares are able to cover for any three failures in the system. Thus both systems can tolerate any three random failures but the pooled architecture requires significantly fewer processors. It is clear that, as the number of tasks to be performed increases, this difference becomes even more important.

The argument just presented applies only to systems where failure detection is provided externally and the only requirement is to replace a faulty processor with a spare. For systems which must detect and locate their own failed processors as well as replace them, the distinction is not so much in the number of processors saved as it is in the level of redundancy provided.

For example, Figure 5 shows fixed and pooled architectures each having 24 processors and providing triad voting for fault detection and isolation. They each have a complement of six spares for redundancy purposes, and both detect and correct faults by comparing the results of A1, A2, and A3 or B1, B2, and B3, etc. and replacing the disagreeing processor with a spare. Unfortunately, in the fixed architecture when a failure has occurred in a given

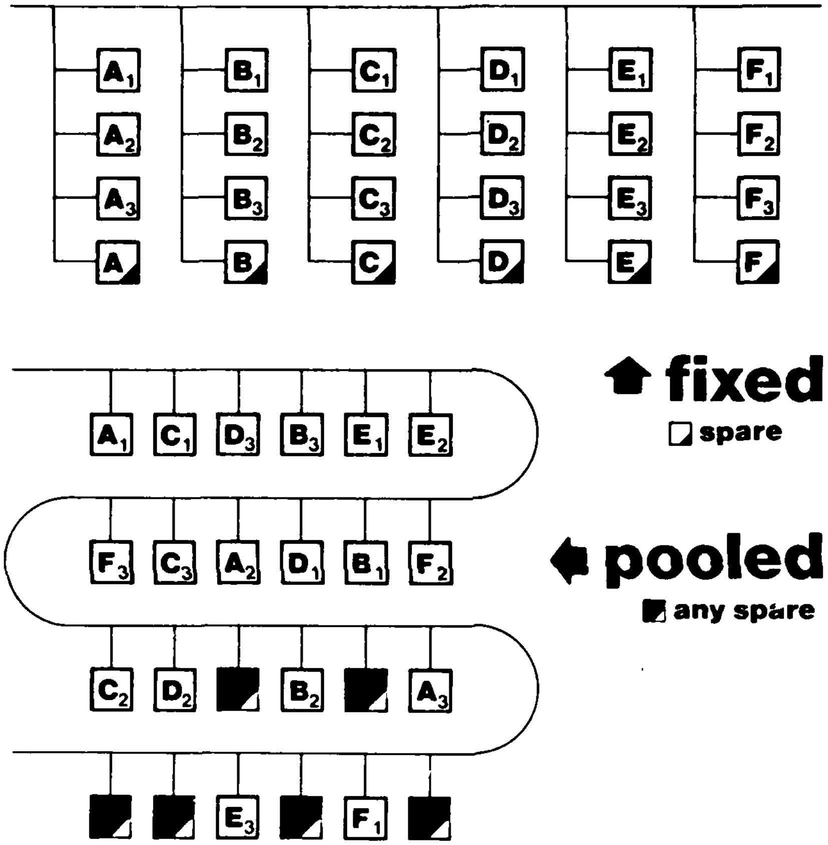


Figure 5. Fixed vs. Pooled Designs
with Equal Numbers

task group, no further failures can be tolerated in that group because its only available spare has been used up. Thus, the entire system can tolerate only one random failure with guarante<d integrity of all six functions. The pooled architecture, on the other hand, can tolerate up to six random failures because all of its spares are free to be assigned wherever they are needed throughout the system.

Because of its many benefits and great versatility, the pooled processor approach was selected for use in this study. Given that decision, and referring again to Figure 3, there are at least three ways of implementing a pooled processor architecture. These three approaches include "cold spares", "hot spares", and "continuous reconfiguration." In each case a pool of spare processors is maintained to replace failed processors. The difference is in the way that the spares are brought on line.

Cold spares are the simplest approach to the problem. A pool of idle spares is maintained and, when a failure occurs, one of the spares is loaded with whatever data and software it needs to perform the missing function and is then brought on line. This is an acceptable method when the system involved is not real-time and a brief interruption during reconfiguration is unimportant. Unfortunately, in real-time systems the delay involved in "warming up" a cold spare is often unacceptable and may even be disasterous.

An obvious solution to the cold spare problem is to maintain a pool of "hot spares." That is, a pool containing spares which are already loaded with all the software and current data needed to come on line immediately after a failure is detected. One hot spare is maintained for each function in the system and the remaining spares are left "cold." When a hot spare switches on line, a cold spare is "warmed up" to replace it so that a hot spare is maintained for every function as long as the supply of cold spares lasts.

The hot spare approach is a great improvement over cold spares and is entirely adequate for most applications. Its chief drawback is the large number of spares required to ensure that every function has its own hot spare. Once a processor has become a hot spare it is essentially dedicated to one function. This is contrary to the goal of truly pooled resources. In addition, although the switch-in time is much improved, reconfiguration is still treated as an emergency requiring special processing and introducing delays and reconfiguration transients. What is needed is an approach which requires a minimum number of spares, produces no reconfiguration delays, and avoids the dedication of spares to specific functions. Continuous reconfiguration is such an approach.

D. THE CONCEPT OF CONTINUOUS RECONFIGURATION

Continuous reconfiguration is defined as a scheme whereby the tasks to be performed in a multi-processor system are dynamically redistributed among all functioning processors at or near the minor frame rate of the overall system. This approach allows continuous spare checkout, latent fault protection, and elimination of failure transients due to reconfiguration delay. By treating reconfiguration as the norm rather than the exception, failures can be handled routinely rather than as emergencies, resulting in predictable failure mode behavior. Using this approach, it is projected that the need for unscheduled system maintenance may be greatly reduced.

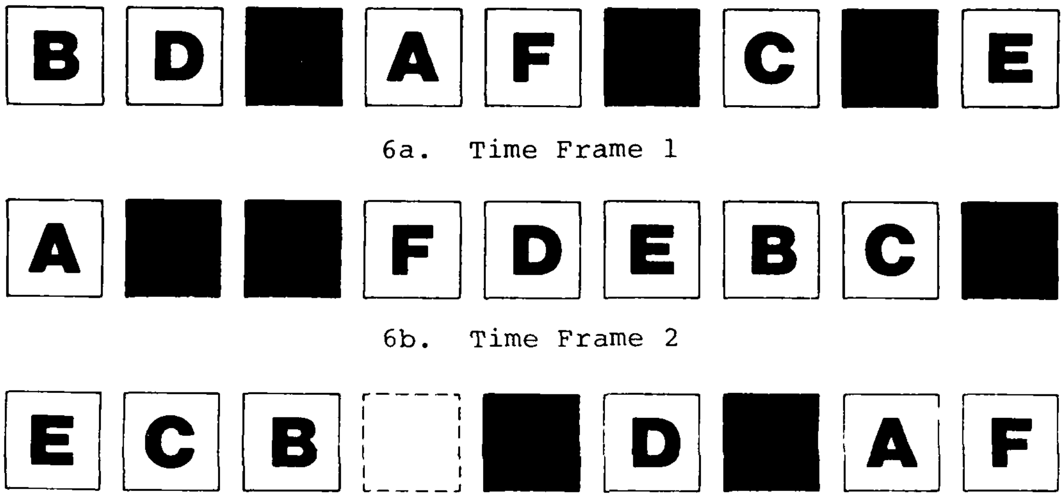
Example Of Continuous Reconfiguration

An example of what is meant by continuous reconfiguration is shown in Figure 6. A system of nine processors is shown performing six different tasks, A thru F, during three consecutive time frames. During the first time frame processor 1 is doing task B, processor 2 task D, processor 3 is a spare, and so on. In continuous reconfiguration the tasks are redistributed among the processors at the beginning of every time frame. For example, in the second time frame , there is an entirely different assignment of tasks to the processors. This reassignment is accomplished by having all of the processors that are currently healthy in the system compete for task assignments. If a processor fails during any time frame, it is no longer able to compete for task assignments and is thereby automatically removed from the system. In Figure 6, if processor 4 failed during the second time frame, then during the next frame, it would not be able to compete for task assignment. The six tasks which need to be done are taken by healthy processors and the two remaining processors become spares (Figure 6c). In other words, a failed processor simply disappears from the system without any other processors being aware that it is gone.

Assuming for the moment that it is possible to implement such a system efficiently, this scheme presents a number of distinct advantages. These advantages will be discussed next.

Advantages of Continuous Reconfiguration

The primary motivation for developing a continuous reconfiguration scheme is to allow a multi-processor system to detect and recover from random failures with no effect on its performance. This is a major problem with



6a. Time Frame 1

6b. Time Frame 2

6c. Time Frame 3

Figure 6. Continuous Reconfiguration

reconfiguration in general since the process of shutting down a failed processor and starting up a spare almost invariably causes a short delay during which the output of the system is incorrect. This period of time is called a reconfiguration delay and the result is a failure transient which can be disasterous at the system output. The main reason for these delays is that most reconfigurable systems treat failures as emergencies requiring special actions which take time. In a continuously reconfiguring system, reconfiguration is the norm, not the exception. Task reassignment is regularly scheduled at frequent intervals so that when a failure does occur the system takes it in stride without missing a beat.

Figure 6c shows how this works. In the figure, processor 4 has failed but the rest of the system doesn't notice it. The task that processor 4 was performing in the previous frame (Figure 6b) has been automatically reassigned to some other processor and the only effect is the net loss of one spare. There has been no reconfiguration delay and therefore no transients due to reconfiguration. This is the first major advantage of continuous reconfiguration.

Eliminating reconfiguration delay by itself can not guarantee that there will be no failure transients at the output. A second source of these transients is failure detection delay. For example, in Figure 6 processor 4 may have been generating bad data for some time before its failure was detected in frame 2. Without continuous reconfiguration this stream of bad data would go to a single output device (rudder, aileron, display, etc.) causing a significant transient in that particular device. With continuous reconfiguration, the bad data goes to a different device every frame depending upon which task the processor is doing at that time. Since real world dynamics are usually much slower that the computer's frame rate, the aircraft will simply not respond to a single sample of bad data. By moving the bad data around from surface to surface, failure effects can be kept insignificant until failure detection occurs. This dispersion of failure effects is the second major advantage of continuous reconfiguration.

A discussion of how to rapidly detect and isolate failures and how to prevent *any* bad data from reaching the system outputs will be presented in Section III when the triad structure is introduced.

A third advantage of the continuous reconfiguration approach is latent fault protection. Latent faults are a class of faults that are inherently undetectable because they produce no noticeable change in system performance or output. This would seem to be no cause for alarm since a fault which produces no error would appear to be harmless. However, a latent fault can be very dangerous if it impairs the system's ability to tolerate subsequent failures. For example, if processor K fails in such a manner that its outputs are correct but it is no longer able to check processor D, then the system will continue to function normally while the fault in K remains unobservable. if processor D should fail, and the system is depending upon K to detect it, a catastrophic system failure may result. The continuous reconfiguration scheme avoids the problem because the processor responsible for checking any other processor changes with every frame. Thus, no dangerous combination of failures is allowed to exist for more than one frame at a time. The possibility of a "deadly embrace" between two partially failed processors is also avoided.

Finally, continuous reconfiguration allows the constant checkout of all processors because no processor serves as a spare for more than one frame at a time. In an ordinary reconfigurable system, where certain processors are always spares, there is a danger that one of the spares will fail before it is needed. If this happens, a disaster may result when the failed spare is used in an emergency. The problem is analogous to changing a tire and discovering that the spare is flat. By constantly "rotating the tires" in a continuously reconfiguring system, failures in any processor can be detected as soon as they occur.

In this section it has been shown that there are four major advantages to the continuous reconfiguration approach. These include (1) zero reconfiguration delay, (2) dispersion of failure effects, (3) latent fault protection, and (4) continuous spare checkout. Because of these advantages and their potential contribution to system reliability, continuous reconfiguration was selected as the method to be used for managing the pooled multi-microprocessor architecture developed in this program. The next section looks at some of the problems involved in implementing a continuously reconfiguring system.

Controlling A Continuously Reconfiguring System

A unique approach was taken for controlling the continuously reconfiguring multi-microprocessor flight control system. The traditional approach would have been to have a central controller in charge of assigning tasks, handling reconfiguration and controlling bus access. Unfortunately, a central controller introduces the possibility of a single point failure in the system requiring redundancy incompatible with the architecture and reducing the reliability of the continuous reconfiguration concept.

An alternative to a central controller is the autonomous control approach. This is a scheme whereby each processor independently determines its own next task based upon the current aircraft state. This can be better understood by using an analogy. Like the traditional centrally controlled computer architecture, a company has a president who has several vice-presidents working for him. The president has access to all information concerning the states of the company and an understanding of how the company should function. He uses this knowledge to allocate tasks to the vice-presidents and arbitrate any disagreements that may arise between them. Autonomous control is analogous to replacing each of the vice-presidents with a clone of the president. The vice-presidents are now capable of making the same decisions that the president would have made under the same circumstances, since each has access to the same data and would go through the same decision making process that he would. The need for the president has been eliminated and he has been replaced by autonomous vice-presidents. This approach is not practical in the human world because no two humans think alike. In the computer world, however, it is a realizable possibility.

Requirements for Continuous Reconfiguration

In order to make continuous reconfiguration of autonomously controlled processors possible, several requirements must be satisfied. These requirements include well-defined task assignment rules, availability of all system state information to all processors, availability of all software to every processor, and an efficient bus contention scheme. The methods used to meet each of these requirements in the laboratory implementation are covered in detail later in this report.

The first requirement is for a set of well-defined task assignment rules. Each of the processors must have an efficient means of determining the next task that it is required to do. There must not be an opportunity for any processor to conflict with other processors in the system and cause system failures. The task assignment rules are a function of the operating system software and are discussed in detail in Section IV.

A second requirement is that all processors must have all software. In order for a processor to be capable of doing any system task at any point in time, it must have the software available to do the task. This may seem unrealistic at first but the trends in memory technology indicate that memory may be expected to double in density several times in the next five years while its cost continues to decrease. This trend makes supplying all software to every processor a reasonable exchange for the benefits offered by the CRMmFCS.

A third requirement of this system is that all processors must have all data. Since any processor must be capable of doing any task at any point in time, each processor must have access to all data concerning the present state of the aircraft. This requirement has been met by the development of a virtual common memory architecture which allows every processor to access any piece of data by what appears to be a simple read from a shared common memory. This concept will be discussed in detail in Section III.

Finally, if all processors are to operate independently and yet share the same set of data busses for communication, some method must be found for them to agree on who can talk on a bus at any given time. Since central control of any kind is not allowed in a fully distributed architecture, this must be done without use of a central bus controller. The scheme selected must also be very efficient since bus bandwidth will be at a premium in systems with many processors. This requirement for an efficient, autonomous bus contention scheme was satisfied through a new approach called "transparent contention." It will also be discussed in Section III.

E. SECTION SUMMARY

This section has presented an overview of the CRMmFCS architecture and the philosophy behind it. The concepts of continuous reconfiguation, autonomous control, transparent contention, and virtual common memory have also been introduced. In the next two sections, these ideas will be discussed thoroughly from both hardware and software points of view. In the process, every major component of the CRMmFCS system will be described in enough detail to give the reader a complete understanding of the overall design. Numerous references to the appendices will be made along the way to aid the interested reader in an even more detailed study of the architecture.

SECTION III
HARDWARE ARCHITECTURE

A. INTRODUCTION

The CRMmFCS architecture consists of a collection of autonomous microcomputers interconnected by a set of serial multiplex busses so that they appear to share one common memory through which they communicate. This section presents the details of how the system was designed from a hardware point of view. Section IV will address the same subject from a software perspective.

One of the main requirements of the CRMmFCS design was the elimination of any form of central control. This meant that processors had to independently determine their own task assignments and that some means was required to manage *communication* without a bus controller. The problem of task selection was solved with software in the CRMmFCS and is discussed in Section IV. Autonomous bus control, on the other hand, had a convienient hardware solution. It is therefore discussed in this section.

The CRMmFCS data bus is fundamental to the entire architecture. The need for an autonomously controlled bus which would act like a common memory influenced the design of every system component. For this reason, the hardware elements of the architecture will be discussed in terms of their relationship to the global bus design.

B. DISTRIBUTED CONTROL OF A MULTIPLEX BUS

One of the most fundamental questions which must be asked when designing a system of autonomous processors is how they will communicate with each other. Direct connection between a large number of processors is clearly impractical since the number of lines required for n processors is n(n-1)/2, (which rapidly becomes very large). A common serial multiplex bus is a more reasonable alternative, but it introduces the problem of bus traffic control: How does one resolve processor contention for the bus without resorting to a central controller? This section presents a promising solution to the problem.

Other Approaches

In order to place this new solution in proper perspective, it is helpful to briefly review several existing bus control schemes. Three such schemes will be discussed including a well known central control approach and two experimental distributed control methods.

The classical central control approach is the MIL-STD-1553 class of busses. Using this scheme, each bus is set up with a central controller and every processor in the system is considered to be a "remote terminal." Any given processor can talk on the bus only when instructed to do so by the central controller. This provides secure and flexible control of global bus resources, but has a number of limitations. First, a processor wishing to transmit must wait until the controller gives it permission, resulting in some inherent throughput delay. Second, there is even more delay involved for one processor to obtain data from another. This is because it must request the data through the controller and wait until the other processor receives the request, looks up the answer, and sends it back. Finally, the system does not make efficient use of bus bandwidth because part of the available transmission time is used up in the overhead of bus control. Thus, the 1553 bus is simple and reliable but somewhat limited in performance. Because it requires a central controller, it also violates the assumed goal of a fully distributed system design.

It should be mentioned in passing that there are 1553-based designs which claim to implement distributed control (Reference 3). Such claims are true only in a limited sense. While potential control of the bus may be distributed in such systems, at any given time there is still only one central controller operating in the traditional command/response mode. For the purposes of this report, distributed control will imply free and open competition for the bus under a set of rules which all participants obey. At no time is such contention arbitrated at any central location, even if the location does move around the system.

There are at least two examples of truly distributed bus control schemes already in existence. The following paragraphs summarize some of the interesting features of each approach but no attempt will be made to cover them comprehensively. The reader may consult the indicated references for further details.

The first approach, found in the University of Hawaii "Aloha" architecture (Reference 4), allows processors to transmit on the bus any time it is available. In the event that more than one processor starts at the same time, a "collision" is said to have occurred and they all stop sending immediately. Each then waits a slightly different interval before attempting to retransmit. The one which "times out" first gains access to the bus and completes its message while all other processors wait until the bus is once again available.

This approach avoids the need for a central controller, but has the limitation that some time is wasted while colliding processors "time out." This becomes serious when demand for the bus is high and collisions are frequent.

Another approach to distributed bus control is used by Honeywell (Reference 5). In this approach, all processors take turns using the bus for a fixed amount of time in a rotating fashion. If a processor has data to transmit, it waits for its turn and then sends it all in a burst of some maximum number of words. If it has nothing to transmit when its turn arrives, it sends a null word and the system moves on to the next transmitter. This is a fairly efficient scheme, although some time is wasted for null messages. The only real difficulty is keeping track of whose turn it is.

Improving Bus Efficiency

While each of the approaches mentioned above has been made to work effectively, there are still a number of things which can be done to improve bus utilization efficiency. These possibilities include:

(1) Scheduling transmissions to avoid periods of bus inactivity or overload.

(2) Forming a queue of data to be transmitted in every processor so that every available microsecond on the bus is in use.

(3) Making sure there is no wasted time between transmissions.

(4) Making sure there are no wasted transmissions due to collisions.

(5) Transmitting data when it is generated instead of waiting until it is needed (thereby reducing access delays).

These five criteria served as design guides for the approach to be presented. In the remainder of this report, it will be shown how these goals have been achieved using the concepts of "transparent contention" and "virtual common memory."

A New Approach to the Problem

This section describes a new approach to autonomous bus control designed to meet the goals listed above. The following is an overview of how the idea works.

Time on the bus is divided into a series of consecutive intervals (slots) that are exactly one transmission word long (32 to 46 bits, depending on word format). At the beginning of each new slot, all processors compete to fill the slot with a word of data. The resulting massive bus collision is then resolved using "transparent contention." Transparent contention is a scheme which allows collisions to occur on the bus in a manner such that only one of the colliding messages survives. All other messages are automatically suppressed without wasting a single bit of transmission time. As a result, the slot is filled with one and only one word and competition moves on to the next available interval.

As long as there is data available to transmit, this approach packs data onto the bus with absolutely maximum density. No time is wasted during transmissions and no time is wasted between them. One hundred percent efficient bus utilization has been achieved.

In order to ensure that there is always data available for transmission, each processor maintains a queue of words to be transmitted. As each new piece of data is generated, the processor places it into a first-in first-out buffer (FIFO) and "forgets about it." A special transmitter circuit then emptys the FIFO onto the bus by competing for time slices with all other transmitters in the system. This frees the processor from transmission considerations and ensures a constant flow of data onto the bus.

There are, of course, potential problems with this approach. If data is not generated fast enough, it is possible for all buffers to become empty resulting in unused time slots on the bus. This is of no concern unless there are other times when too much data is generated resulting in backlogs and throughput delays. It is therefore important to schedule data generation in the system such that an even rate of transmission is maintained. A technique for scheduling data flow is presented later in this section.

All that remains now is to explain the details of transparent contention. The following section discusses how it can render bus collisions harmless. Subsequent sections will present details on how to build and use such a system.

C. TRANSPARENT CONTENTION

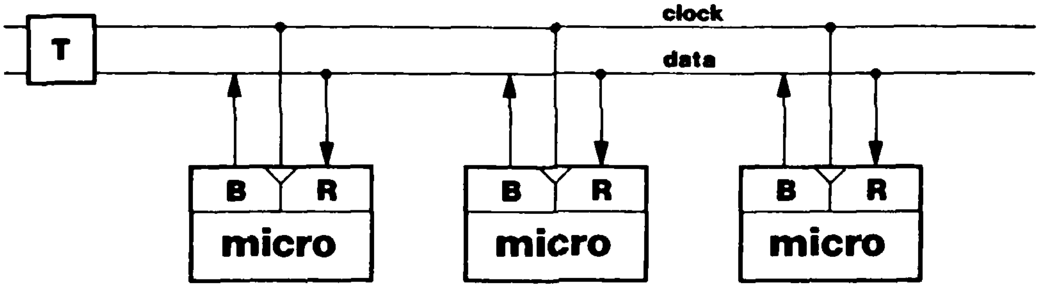
This section presents the theory of operation behind transparent contention. While almost any bus configuration can make use of the idea, one specific design was chosen because of its ease of implementation in the laboratory. This design is covered first in order to clarify subsequent discussion of the transparent contention concept.

The approach selected is nothing new. It amounts to nothing more than clocking data out of one shift register across a serial bus and into another. What *is* unique is the manner in which this process is controlled to prevent conflicts on the bus between contending transmitters. In order to understand this process, it is helpful to review how a single transmitter operates when there is no competition from other transmitters.

Transmission Without Contention

The essential elements of the bus architecture are shown in Figure 7. In the figure, three processing modules are shown interconnected by a common serial bus made up of a data line and a clock line. Each processing module consists of an ordinary microcomputer with two I/O devices including a broadcaster (B) and a receiver (R). These devices use the signal on the clock bus to shift data on to and off of the data bus respectively. The box labeled "T" in the figure is a bus termination circuit which generates the clock signal and terminates the bus properly (See Appendix C).

Using this simple bus structure, a word of data is transmitted in the following manner. The processor wishing to transmit first places its information in its local broadcaster FIFO. If the bus is available (as we assume in this section), the broadcaster immediately latches the FIFO output into a serial shift register and shifts it onto the data bus with each positive-going edge of the bus clock. On each negative-going edge, a bit on the bus is shifted into receiving shift registers in every processor. From there, the complete word is moved into local memory in each processor using direct memory access. This technique will be discussed later in the section on "Virtual Common Memory Design."



Figurc 7. Essential Architecture Elements

Transmission With Contention

The question now arises, "What happens when more than one processor wishes to use the bus at the same time?" The answer is simply, "one of them wins." Exactly which one wins is determined by a special logic circuit in each transmitter which resolves the conflict. Its operation is described below.

In the first place, access to the bus is granted on a first-come, first-served basis. While one processor is actively using the bus, a logical BUSY signal is maintained which prevents any other processor from initiating a broadcast (See Appendix A). This eliminates many conflicts, but sooner or later more than one transmitter will begin using an available bus on the exact same clock pulse. When this happens, some other method is required to resolve the contention.

The solution is found by observing what actually happens when two transmitters put data on the bus at the same time. As shown in Figure 8, each transmitter is connected to the bus by an open collector transistor buffer. When a transmitter wants to send a "zero", it turns on its output transistor shorting the bus to ground. To transmit a "one" the transistor is turned off, allowing the bus to be pulled high by the pull-up resistor. As long as all transistors are turned off, the bus will float at a logic "1". If any transistor turns on, the bus will be pulled to a logic "0" state.

The net result is that logic zeros have an inherent prioriy on the bus. Because a "1" is transmitted by "letting go" of the bus (so it will float high) while a "0" is transmitted by actively pulling the bus low, units

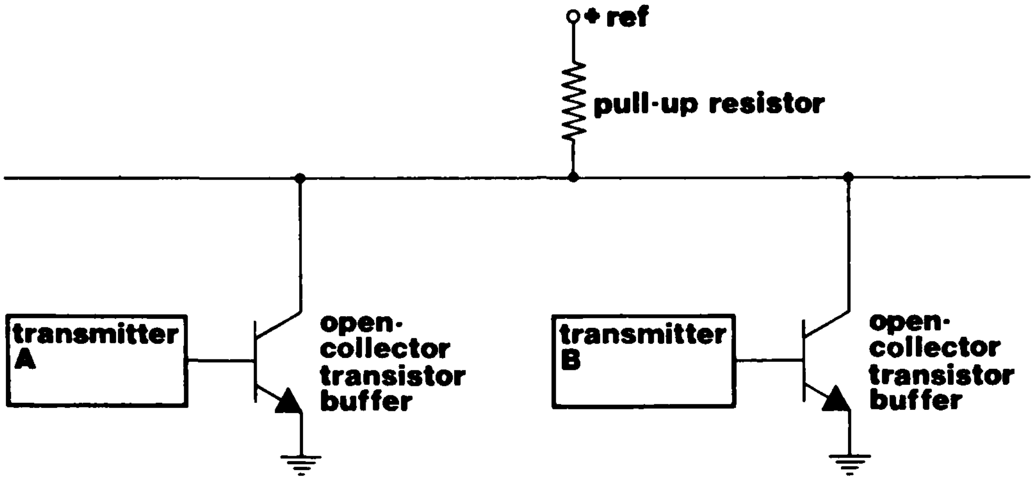

+ ref
pull-up resistor
transmitter A
open-collector transistor buffer
transmitter B
open-collector transistor buffer

Figure 8. Transmitter-Bus Interface

transmitting zeros will always win out over those sending ones. It is this characteristic which allows transparent contention.

The key to the idea is that every transmitter constantly compares what it is trying to put on the bus with what is actually there. In the event of a disagreement, the transmitter simply stops sending and waits for the bus to become available again. What makes this approach work is that when any two transmitters disagree, only one of them notices and drops off. The other one does not notice (because it got its way on the bus) and therefore continues its transmission. No bus time is wasted because one message is finished without interruption.

At this point an example is helpful. Suppose two broadcasters begin to transmit on the same clock pulse as in Figure 9. Transmitter one attempts to send the binary sequence 01001 while transmitter 2 sends 01101. During the first microsecond, both pull the bus low and observe a zero on the bus. Since that is what they wanted, they continue to transmit. During the next microsecond, both transmitters "let go" of the bus allowing it to float high. They each observe a logic 1 and, satisfied, continue to transmit. However, during the third interval, transmitter 2 releases the bus to let it float high while transmitter 1 actively pulls the bus low. They both observe a zero on the bus. Since that is what number 1 wanted, it continues to transmit. Number 2, on the other hand, does not get its desired "one" and concludes that some other transmitter has pulled the bus low. It therefore aborts its transmission and waits for the bus to become available again.

The net result is that transmitter 1 successfully completes its transmission from start to finish without interruption while transmitter 2 aborts as soon as the two

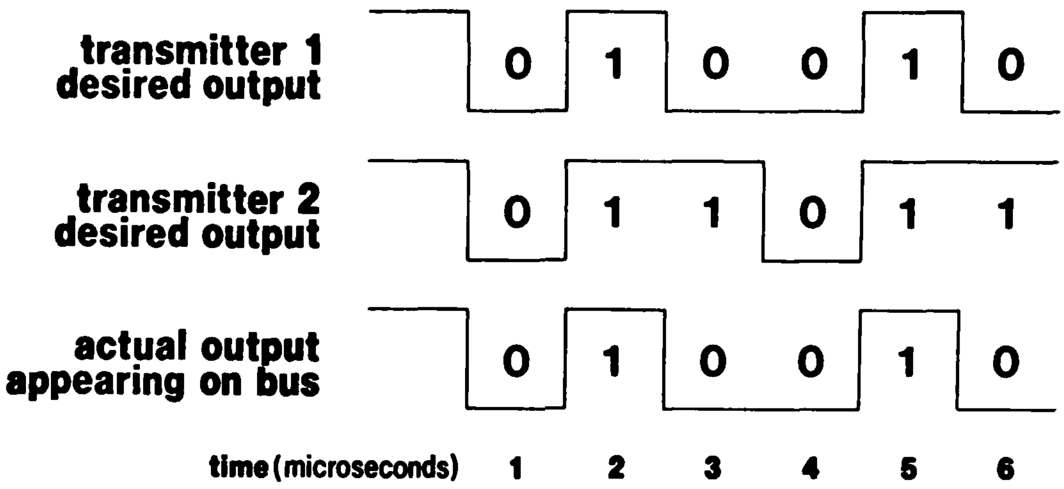

transmitter 1 desired output
0 1 0 0 1 0
transmitter 2 desired output
0 1 1 0 1 1
actual output appearing on bus
0 1 0 0 1 0
time (microseconds)
1 2 3 4 5 6

Figure 9. Bus Contention Arbitration

disagree. No transmission time was lost and, in fact, transmitter 1 was never even aware of its competition. "Transparent contention" has been achieved.

This concept works equally well for any number of transmitters in contention. If ten of them start simultaneously, they all send in parallel until there is a disagreement. At that time those sending zeros win while those sending ones drop off. The remaining transmitters continue until the next conflict at which time still more losers drop off. Eventually, only one transmitter is left and it finishes its transmission, completely unaware of its nine vanquished competitors.

This approach is based upon the assumption that no two transmitters will ever try to send identical words at the same time. If this coincidence should occur, each processor would assume that its own broadcast was successful and only one copy of the word in question would appear on the bus. This may or may not be tolerable depending upon how information appearing on the bus is used.

A more significant consideration is the event in which two words being transmitted fail to disagree until near the end of the word. At that time the losing processor would abort its transmission, but only after having wasted its time sending most of the word. In a system with only one bus this is unimportant since the losing transmitter would have nothing to do but wait for the bus anyway. But in a system with n busses (to be discussed next), it is desirable for a transmitter to find out if it is going to lose as soon as possible so that it can begin searching for another bus.

If these considerations are important, there is a simple solution. Each transmitter adds its own unique identification code to the beginning of each message. In a system with 16 processors, this code would be 4 bits long. Five bits would allow up to 32 processors, and so on. Using this method, two processors are guaranteed to disagree within the first 5 bits freeing the loser to seek another bus. This aproach has the added benefit that it is possible to determine which processor initiated each broadcast for fault isolation purposes.

An Extension to n Busses

The bus structure which has been discussed so far represents a very simple way to interconnect a large number of autonomous processors without need of a central controller. However, a single bus system of any kind is generally unacceptable from a reliability standpoint. At the very least, some form of redundancy is required in order to avoid a potential single point failure node in the system. Also, a single serial bus has only a finite bandwith. A large system of processors exchanging massive amounts of data can quickly saturate such a bus. The approach proposed in this report is ideally suited for expansion to as many busses as are needed to meet the reliability and throughput requirements of nearly any system. The following paragraphs detail the implementation and advantages of an autonomously controlled n-bus design.

Figure 10 shows the transmitter interface of a single processor in a system with four busses (4 sets of clock and data lines). The circuit is controlled by the box labeled "transmission control logic." Upon receiving a "START" signal (from the CPU), this logic instructs the "bus finder" to "SEARCH" for a free bus. When it finds one, it locks two data selectors and a data distributor onto the bus (using its "BUS SELECT" lines) and signals the control logic that it has "FOUND" a bus. The control logic then loads the shift register with data from the CPU output buffer and

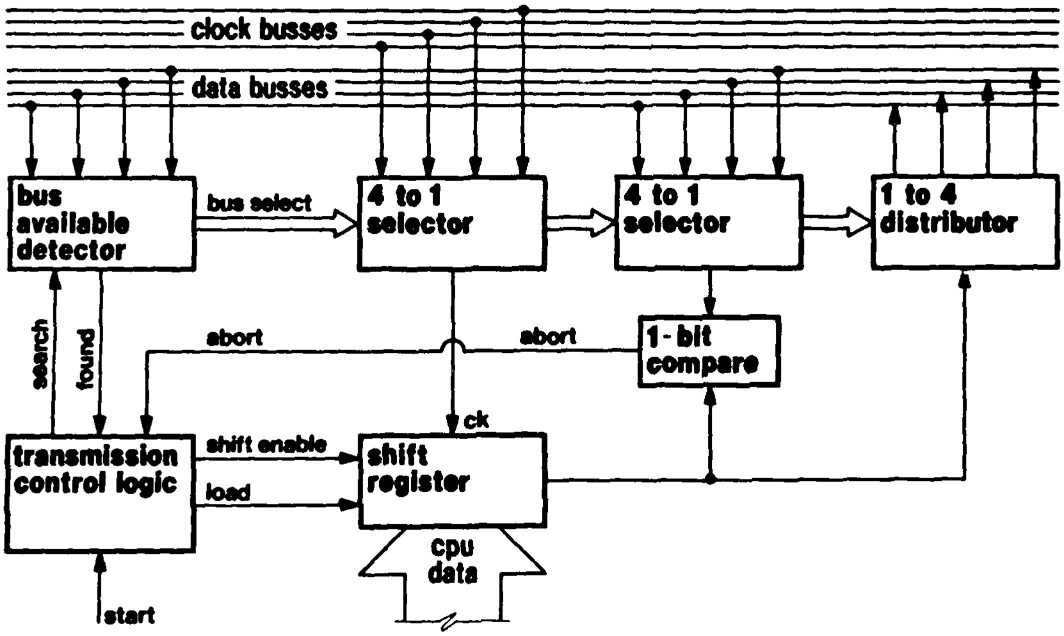


Figure 10. Transmitter Circuit

enables the shift register clock. Data is shifted (using the appropriate bus clock) out through the 1 to 4 distributor onto the selected data bus using the same open-collector transistor buffers shown in Figure 8. As always, a one-bit comparator monitors the difference between the shift register (desired) output and the actual output on the selected bus. If there is ever a miscompare, an "ABORT" is generated and the transmission control logic instructs the bus finder to locate another bus. This process continues until the transmitter is successful at placing its entire word on the bus, at which time another word is obtained from the CPU buffer and the cycle begins again.

This bus design has tremendous flexibility. Its bandwidth is exactly four times that of a single bus and can be expanded still further with additional busses. Reliability is also enhanced. Because processor to bus connections are continuously reconfiguring, selection of an alternate bus in the event of a failure is instantaneous and automatic.

There are, however, a few physical limitations which remain to be resolved. The first is that there is a limit to how many open-collector transistors can be "wire-ored" to one bus before the sum of their leakage currents pulls the bus low even if no transistor is on. This limit can be increased using low leakage transistors, but can never be totally ignored. Noise considerations on such a bus will also require further research. For the present, the triplex data approach described in Section IV will be relied upon to correct for noise-corrupted transmissions.

Another consideration is the effect of propagation delays on the output of each transmitter comparator. The fact that desired and actual outputs match at one location is no guarantee that the same holds true many feet away on the bus. This problem is avoided for reasonable line lengths by the manner in which data is clocked onto the bus. Data is shifted onto the bus on the rising edge of each clock pulse, but the comparators output is not sampled until the falling edge. This allows one half of a microsecond for the data to settle before it is used.

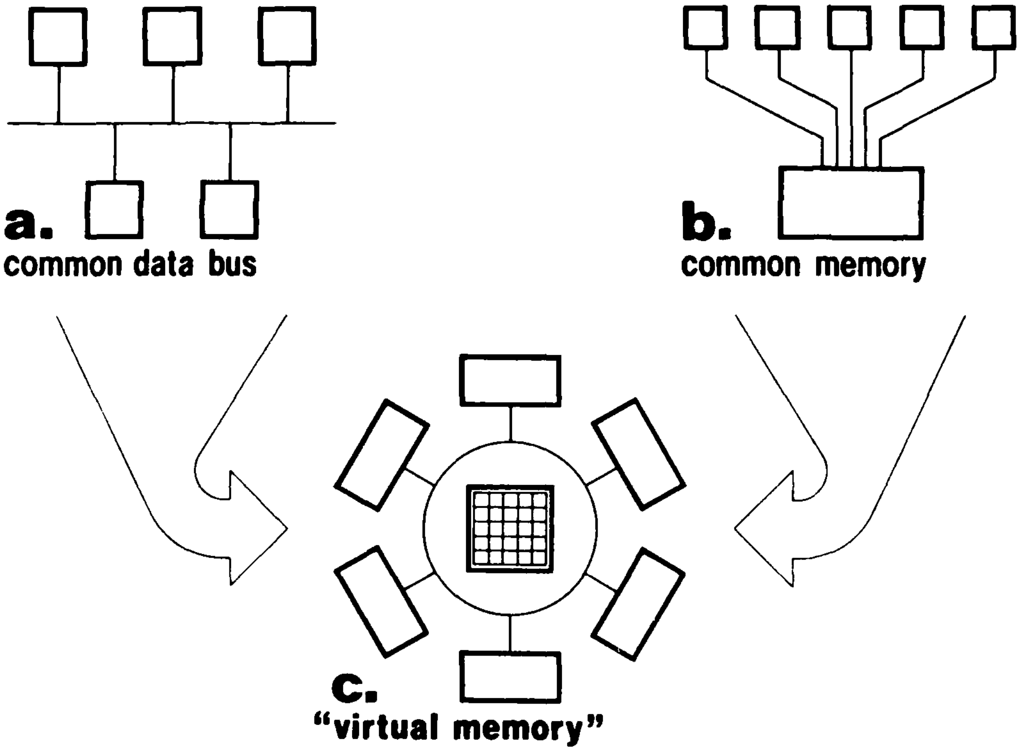
Finally, the open collector transistor implementation is only one approach to the transparent contention concept. Any technique where one logic state wins out over another will work. In the case of fiber optic busses, for example, the presence of light on the bus could be made to win out over its absence, and so on. For the purposes of concept demonstration in the laboratory, the wired-or approach has been shown to work very well.

D. VIRTUAL COMMON MEMORY

Up to this point, discussion has centered around the transparent contention concept and its physical implementation. In this section an actual application is presented, allowing the development of what is called "virtual common memory."

The Virtual Common Memory Concept

One of the main problems that occurs in the design of multiprocessor systems is how to distribute and exchange data efficiently. From a hardware standpoint, the easiest approach is usually to connect all processors to a common serial multiplex bus (Figure 11a). This minimizes hardware complexity and allows expandability, but often involves a large software overhead. This is because processors must



Fiqure 11. Evolution of Virtual Memory

exchange data on a "request" or "broadcast" basis, both of which require special handling by every processor in the system.

In the "request" mode of operation, a processor that needs a piece of information simply asks for it on the bus and receives it from some other processor a short time later. This means that each processor must constantly monitor the bus for data requests rather than concentrating upon the task to which it has been assigned. Because processors must wait for much of their data, processing inevitably takes longer than it would if all data were available in local memory at the start of a given task. Additional processing time is also wasted in responding to requests for data from other system processors.

The "broadcast" method is an alternate approach to data exchange where every piece of data is transmitted as soon as it is produced. Each processor then selects from the bus whatever information it needs to accomplish its current task. This approach also requires a lot of overhead as each processor must now constantly monitor the bus for items of local interest.

Thus, the common serial multiplex bus, while being the most simple and flexible from a hardware point of view, has serious drawbacks in terms of software complexity.

The simplest and most efficient approach to interprocessor communication from a software point of view is a common memory containing all information required by all processors (Figure 11b). In such a system, a processor stores its output in the common memory where it can be instantly accessed by any other processor in the system. When one processor needs information from another, it simply reads it from the common memory without delay.

Unfortunately, what is ideal from a software standpoint is difficult to implement in hardware. Serious contention problems develop when more than one processor attempts to access the same block of memory. Since each processor must be connected to the common memory by a complete set of address and data lines, the hardware complexity is also large. Finally, system expandability is impaired. In a serial bus system more processors can be added by simply connecting them to the bus, but there is a limit to the number of ports available in a common memory. When these have been used, no more can be added without redesigning the system.

So, it appears that what is good for hardware is bad for software and vice-versa. Clearly, a scheme that could combine the best of both approaches is highly desirable. Virtual common memory is such a solution.

Virtual common memory is a method for making a serial multiplex bus look like a single common memory to the system programmer. As such, it combines the hardware simplicity of a serial bus with the software simplicity of a common memory (Figure 11c). In the following paragraphs the physical implementation of a virtual common memory containing all information to be shared among processors will be described.

Virtual Common Memory Design

This section describes the virtual common memory design used in the CRMmFCS architecture. The essential features of this design are shown in Figure 12. In the figure, six processing modules are shown connected by four serial multiplex busses of the kind described above. This represents the system actually being constructed at the

Flight Dynamics Laboratory, although additional busses or processors could have been included. The number shown is considered to be enough to demonstrate the overall concept.

In a true shared memory architecture, the common memory contains all information required by any processor in the system. This information describes the entire state of the aircraft, and the memory which contains it is called the "state information memory" (SIM). When a processor needs a particular state variable, it accesses a well-defined location in the SIM. When it generates a variable, it places it in a specific SIM location where other processors can find it. No other processing is required for complete interprocessor communication. A complete discussion of the SIM concept is presented in Appendix F.

In the virtual common memory design of Figure 12, each processor is given its own copy of the SIM. To access a SIM variable it simply looks up its own copy. To store a variable into the SIM, a processor broadcasts it over the bus and every processor's copy is updated simultaneously. Since reading from a local copy is the same as reading from a common one, and since writing to the bus is the same as writing to a common memory, as far as any processor is concerned there is only one "virtual" common memory being used by everyone.

In order to make bus transmissions appear to be reads and writes on a common memory, two special circuits were designed. These circuits are described next.

The Transmitter. The transmitter circuit (labeled "XMIT" in Figure 12) was shown in detail in Figure 10. It is connected to the microcomputer through a two-page buffer (Pl and P2) which is memory-mapped to the local CPU. These

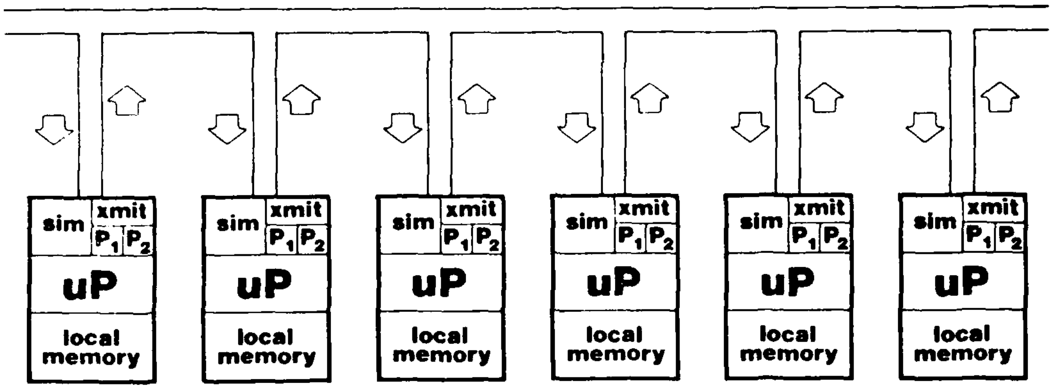


Figure 12. Physical Implementation of Virtual Memory

pages alternate functions every millisecond so that, while one of them is being loaded by the CPU, the other is being unloaded onto the bus. This dual buffer approach ensures that all data is transmitted with no more than one millisecond delay. Its application will be discussed in greater detail later. Complete details on the transmitter circuit are given in Appendix A.

The Receiver. The receiver consists of two major parts including a serial to parallel converting shift register (SIPO) and a block of random access memory (RAM) which contains a complete copy of all state information in the system. This state information memory (SIM) is mapped into the microcomputer's address space as a block of "read only" memory and is accessed by the SIPO outputs as a block of "write only" memory. Figure 13 shows how this works.

In the architecture under discussion, a word of transmitted data is 37 bits long. It consists of four bytes of significant data separated by a zero bit before and after each byte (see Figure 13). A string of more than eight consecutive "1" bits on the bus indicates that it is no longer in use, so zero bits are included between each byte to ensure that the bus continues to "look" busy in the event that more than eight consecutive ones occur in the actual data word.

Once the SIPO has been fully loaded with a 37 bit word from the bus, the 32 significant bits (excluding the 5 separating zeros) are loaded onto the address and data lines of the SIM RAM as follows. The first 5 bits contain the identity code of the sending processor. These bits were used only for quick resolution of bus contention and could be discarded unless it is important to record who sent each word for fault isolation purposes. (In the CRMmFCS design,

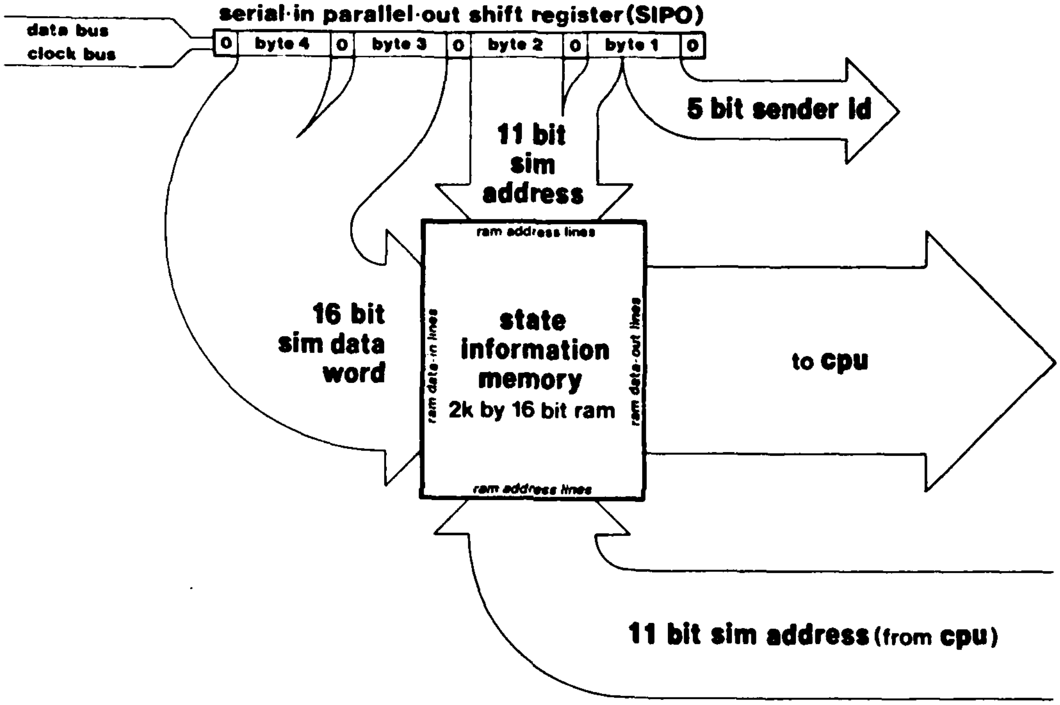


Figure 13. Receiver Block Diagram

these bits are in fact saved in the SIM for use by black-balling algorithms -- see Section IV-D.) The next 11 bits specify the location in the SIM to which this data word is to be stored. They may be thought of as the name of the SIM variable and, in this case, they allow up to 2024 different variables. Finally, the last 16 bits of the transmission contain the value of the variable. These 16 data bits are loaded into the SIM via direct memory access (DMA). The variable is now available for access by the CPU whenever it is needed.

While all of this is happening, the SIPO register is collecting the next word of data as it appears on the bus. This, in general, begins after nine bus clock cycles (during which the bus floats "high"). At this time every other transmitter realizes that the bus is no longer in use (because nine consecutive "ones" have occurred) and contention for the bus begins again. Thirty seven microseconds later the SIPO is again full and another DMA cycle is executed to load its contents into the SIM. Thus, a word is received every 9 + 37 = 46 microseconds.

In a system with n busses, there are n SIPO shift registers requesting direct memory access to the SIM. Since there are 46 microseconds between successive DMA requests from any one bus, and since current high speed RAM can handle as many as four accesses per microsecond, it is theoretically possible for a system to have as many as 4 x 46 = 184 serial busses, each operating at 1 MHz, for a 184 MHz total system bandwidth. Such a system would allow processors to exchange up to 184 x (1 variable/46 microseconds) = 4 million variables per second.

Of course, connecting 184 shift registers to a single block of memory is impractical with today's technology, but if the entire CPU, SIM, transmitter, and receiver were integrated on a single chip (with only the bus lines brought out to the pins), such an approach might well be possible. Until then, a practical limit is about 8 busses with a corresponding bandwith of 8 MHz. Appendix B presents a complete description of the receiver design.

While the potential for high throughput rates is intriguing, the real usefulness of virtual common memory is to allow easier programming of processors connected by serial multiplex busses. The next section shows how a virtual common memory architecture can be used most effectively for this purpose.

E. EFFECTIVE USE OF VIRTUAL COMMON MEMORY

Knowing how to use a virtual common memory architecture can make a big difference in how useful the idea is. This section discusses the application of virtual common memory to the CRMmFCS design. This is not the only way to use virtual common memory, but it does illustrate some important considerations for effective use of the concept.

In the first place, it is important to schedule transmissions carefully in order to take full advantage of the data packing capabilities of the architecture. Figure 14 shows how transmissions are scheduled in the CRMmFCS design.

Time in the system is divided into 1 ms frames and all processors in the system are synchronized to this frame rate. Since each transmission is exactly 46 microseconds long, and since the transmitters pack data onto the bus with no wasted time in between, there is room for 1000/46 or 21 complete transmissions per bus per millisecond. Therefore, when system software is being written, it is modularized into 1 millisecond chunks (called "millimodules") and no more than 4 x 21 = 84 transmissions are scheduled for any one millisecond. Since there are 84 slots available in each frame, this guarantees that every scheduled transmission will be completed sometime within its assigned millisecond.

Each dot in Figure 14 represents a 46 bit word of data appearing on the bus. In this example, 22 of the 84 available slots are scheduled for use. At the beginning of each millisecond, all transmitters compete to place their part of the scheduled variables onto the bus. Transparent contention packs these words into one slot after the other on all four busses until every scheduled word has been transmitted. Then the bus sits idle until the start of the next millisecond.

It is, of course, possible to schedule 84 transmissions per millisecond and never have the bus sit idle. However, for reliability reasons, it is usually wise to leave enough unused slots to "take up the slack" in the event that one of the busses fails. In the CRMmFCS design, 42 slots are left unused in each frame so that the system can tolerate two bus failures without loss of throughput.

This approach makes processor task scheduling a lot easier. Because it is known (to the nearest millisecond) exactly when a variable will appear on the bus, it is also known (to the nearest millisecond) how soon a task can be scheduled using that variable. Figure 15 summarizes this scheduling procedure and illustrates an efficient virtual common memory system in operation.

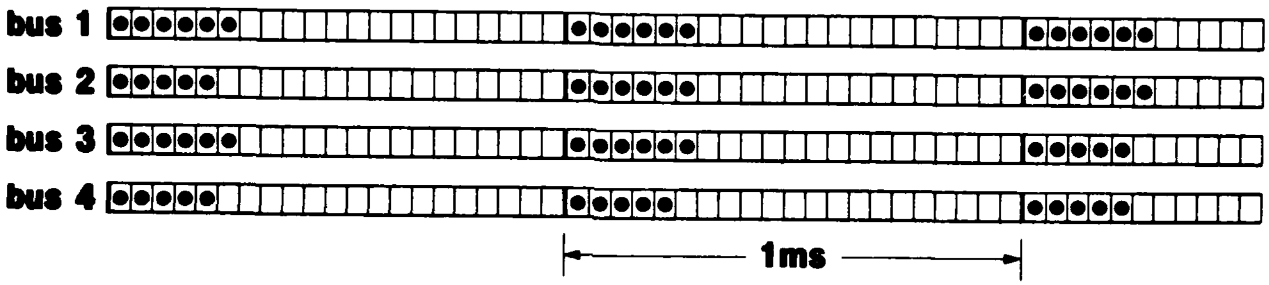


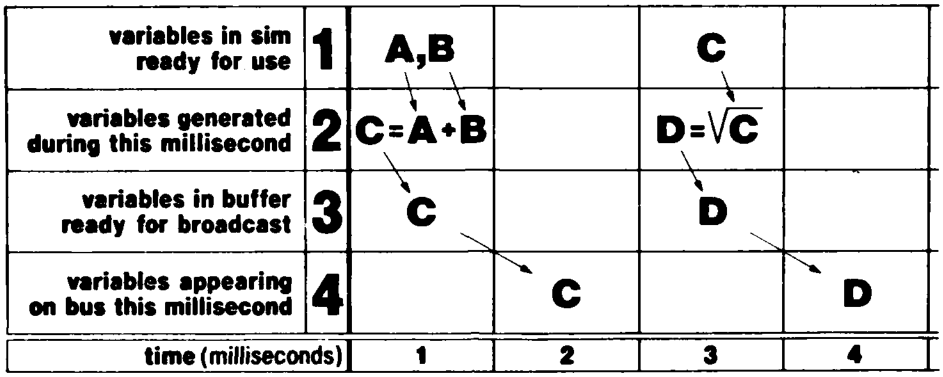
Figure 14. Bus Transmission Scheduling

The figure is divided into four different rows showing where variables A, B, C, and D are scheduled to be over a period of four milliseconds. During the first frame, variables A and B are shown in row 1 indicating that they are currently in the SIM and available for use. As a result, Task 1, which computes C = A + B, can be scheduled for this frame as shown in row 2. Once C has been computed, it is placed in the currently active page of the transmitter buffer (refer to Figure 12) where it remains for the duration of the frame. Let us assume that this was page 1.

At the end of frame 1, the two transmitter pages are switched so that page 1 is connected to the bus and page 2 (which was emptied onto the bus during frame 1) is connected to the CPU to collect any data generated during frame 2. Now that page 1 is connected to the bus, the transmitter circuit broadcasts variable C in the first available slot. This is indicated in row 4 of the figure.

It is not possible to know exactly when during frame 2 variable C actually appears on the bus. This is a random function of when transparent contention allowed the transmitter to gain access. But since there are more slots than there are scheduled variables, sooner or later C will get its chance. It is therefore guaranteed to reach the SIM in every processor by the end of frame 2. This is indicated by showing C in row 1 ready for use during frame 3. Task 2, which computes D = SQRT(C), can now be scheduled for frame 3 and the entire process repeats.

Thus, one of the major goals of the design has been accomplished: elimination of access delay to global variables. Because data is broadcast to every processor as soon as it is generated, it is always available in the SIM by the time a task is scheduled to use it.



variables in sim ready for use	1	A,B		C	
variables generated during this millisecond	2	C=A+B		D=$\sqrt{C}$	
variables in buffer ready for broadcast	3	C		D	
variables appearing on bus this millisecond	4		C		D
time (*milliseconds*)		1	2	3	4

Figure 15. Data Flow Assignment

F. SECTION SUMMARY

This section has presented an overview of the essential hardware elements of the CRMmFCS. It has also introduced two new concepts in interprocessor communications. The first is called "transparent contention" and represents a method for autonomous processors to share a common bus at maximum efficiency without need of a central controller. It has been shown that this approach opens up many new possibilities for improved bandwidth while avoiding the pitfalls of single point failures possible in many central controller designs.

The second concept presented is called "virtual common memory." It represents a method of minimizing the hardware and software involved in interprocessor communications. While not essential to the concept, transparent contention was shown to be an ideal method for implementing virtual common memory in many practical situations.

The next section of this report discusses how the CRMmFCS software structure was designed to utilize the hardware which was developed in this section.

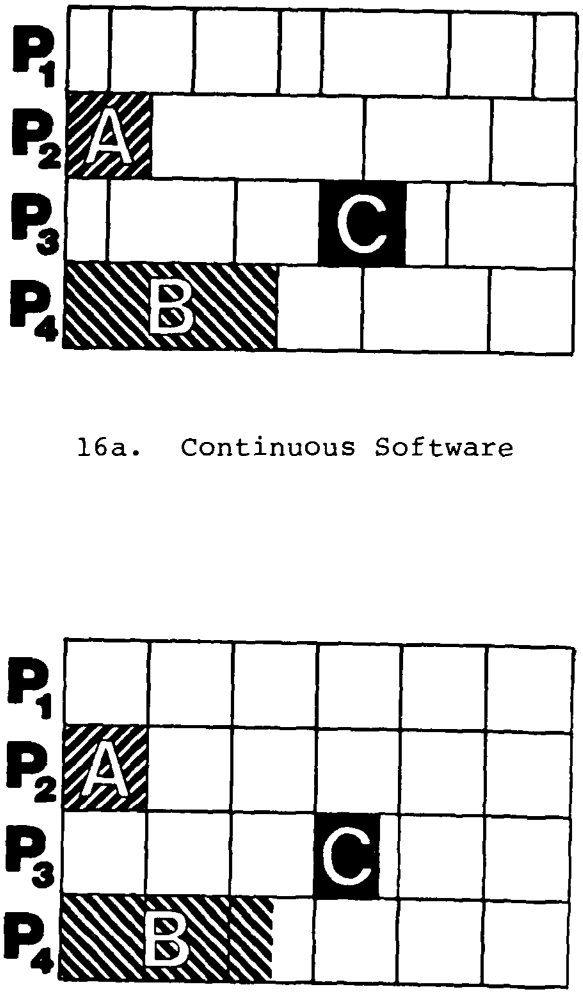
SECTION IV
SOFTWARE STRUCTURE

A. INTRODUCTION

In Section III it was shown how a simple set of serial busses could be made to look like a shared common memory to the system programmer. Now that such a system may be assumed to exist, it is time to show how software can be designed to take advantage of this new architecture. This section presents an approach to organizing software which makes continuous reconfiguration possible and simplifies the task of programming a multi-microprocessor system. This approach is one of the major outgrowths of the CRMmFCS research project.

Programming a system consisting of a large number of processors can become a formidable task. If one is not careful, task scheduling and synchronization problems can make system software a nightmare to modify and maintain. For example, Figure 16a shows how four different processors might be programmed in a multiprocessor system. Each row in the figure represents what one processor is scheduled to do during a given time frame and is broken down into a series of tasks of varying length. As each task is completed a new one is scheduled immediately, thereby packing as many functions into one processor as is physically possible.

This approach maximizes the throughput of every processor in the system but can become very difficult to synchronize. For example, processor 2 does task A in Figure 16a while processor 4 does task B. Processor 3 task C which is supposed to combine the results of tasks A and B. However, if task B is not completed before task C is started, then task C will not have the information needed to



16a. Continuous Software

16b. Quantized Software

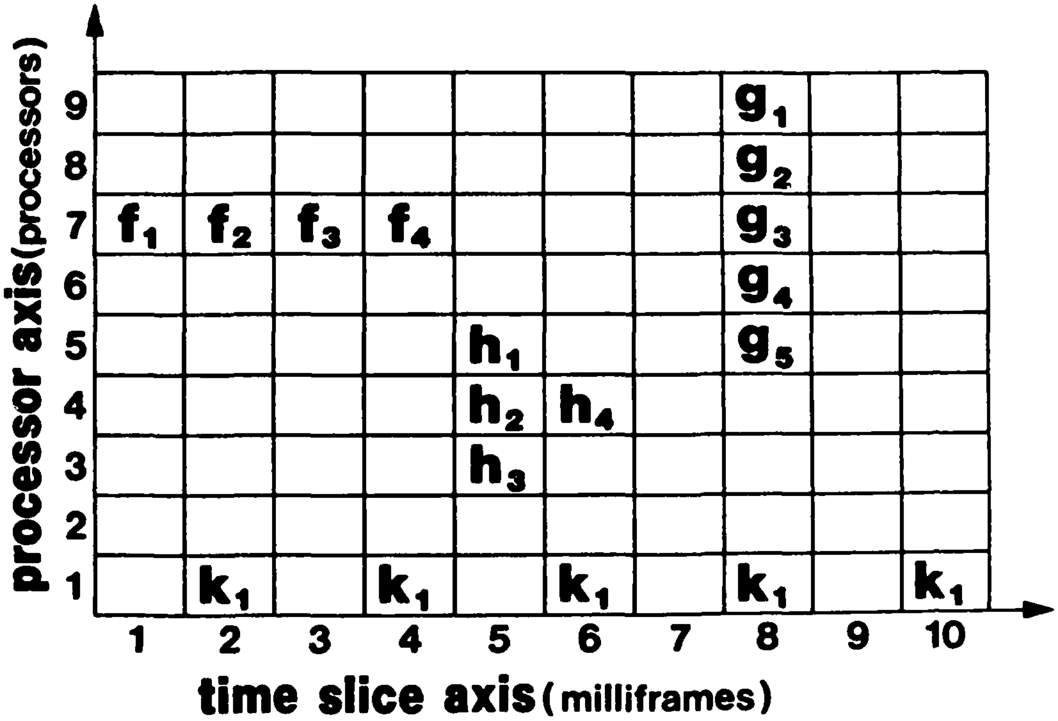
Figure 16. Approaches to Task Scheduling

complete its calculations. This possibility can greatly increase the complexity of the software. A second problem with this programming technique is that it can be difficult to modify. If a block of software requires rewriting or a new algorithm must be added, the timing of the software will be changed. Since synchronization must be maintained between certain tasks, this will require revalidation of all software. One small change can therefore influence the software of the entire system.

The CRMmFCS design takes a different approach to the problem as shown in Figure 16b. In the figure, all software has been divided into a series of standard modules of uniform size. Tasks A, B, and C (and every other task) are then rounded out to an integer number of these modules. This allows control over which tasks are performed during any given interval of time, so strict synchronization of tasks can be maintained. Data is exchanged only on boundaries between modules. As a result, the availability of data for subsequent tasks is known at any given time. Since all software modules are the same size, they can be easily interchanged. The following paragraphs discuss this software structure in greater detail.

B. THE TASK ASSIGNMENT CHART

The approach described above is known as the "quantized software" approach. It divides all system software into separate modules of some standard fixed length (in terms of execution time, not number of instructions). Each of these modules is then placed in a matrix of functions called a task assignment chart (TAC). Figure 17 shows an example of such a chart. In the figure, the horizontal time axis is divided into a sequence of individual time slices of some arbitrary unit length. The vertical axis represents the



	1	2	3	4	5	6	7	8	9	10
9								g_1		
8								g_2		
7	f_1	f_2	f_3	f_4				g_3		
6								g_4		
5					h_1			g_5		
4					h_2	h_4				
3					h_3					
2										
1		k_1		k_1		k_1		k_1		k_1

Figure 17. A Generic Task Assignment Chart

number of processors available in the system with each additional processor representing a unit increment in processing power. Thus, the TAC is made up of a matrix of processing functions one time slice long and one processor wide. These resource units are called "millimodules" and each millimodule is said to occupy one "milliframe" on the time axis. In the CRMmFCS architecture, a milliframe is exactly one millisecond long.

Once a standard millimodule size has been selected, all necessary processing tasks may be assigned to the chart in the following manner. A given task, say F, is first divided into a group of subfunctions, f1, f2, ... fn, each of which require at most one millisecond to execute. Each of these subfunctions is then designated as a millimodule and placed in a convenient location in the task assignment chart. Figure 17 shows a variety of ways in which this assignment can be accomplished.

In the figure, function F (f1, f2, f3, f4) has been assigned to processor 7 and executes in four consecutive time intervals beginning in milliframe 1. Function G (g1, g2, g3, g4, g5) executes entirely in parallel requiring five processors (5, 6, 7, 8, and 9) and only one time slice (milliframe 8). Function H (h1, h2, h3, and h4) first generates intermediate results in parallel and then combines them in processor 4 during milliframe 6.

After milliframe 10 the entire process repeats for the next iteration of each function. Faster iteration rates may be achieved by assigning the same function several times in the same chart as shown for function K (k1) in processor 1.

In a manner similar to the example above, all processing functions may be broken into millimodules and assigned to the TAC. As the space fills up, it may be readily extended by simply adding more processors along the vertical axis. Since all millimodules are one standard size, they are easily manipulated, interchanged, and may even be dynamically reassigned during real time. This provides the system designer with great flexibility in managing his processing resources.

The benefits of this approach are not without cost. Very few functions will fit exactly into an integer number of millimodules so some time must be wasted in rounding them to the next whole module. This requires more processors to make up for the reduced throughput of less densely packed software. Fortunately, such a tradeoff is generally desirable since reduced software costs will usually pay for the small increase in hardware.

Application of Task Assignment Charts

This section describes how the task assignment chart was applied in the CRMmFCS architecture. Figure 18 shows the task assignment chart used for one flight control mode. In this chart, milliframes have been selected to be exactly one millisecond long. Ten consecutive milliframes make up a minor frame and three minor frames form one major frame. This major frame repeats continuously as long as the system is in mode 1. If the mode changes (due to pilot inputs or changes in flight condition), then a new task assignment chart is switched in at a major frame boundry.

The vertical axis has been changed subtly in Figure 18. Each row still represents a particular set of tasks to be performed, but there is no longer a specific processor associated with each row. In the CRMmFCS architecture, processor row assignments change with time. As a result, the vertical axis has been relabeled in terms of task numbers instead of the processor numbers used in Figure 17.

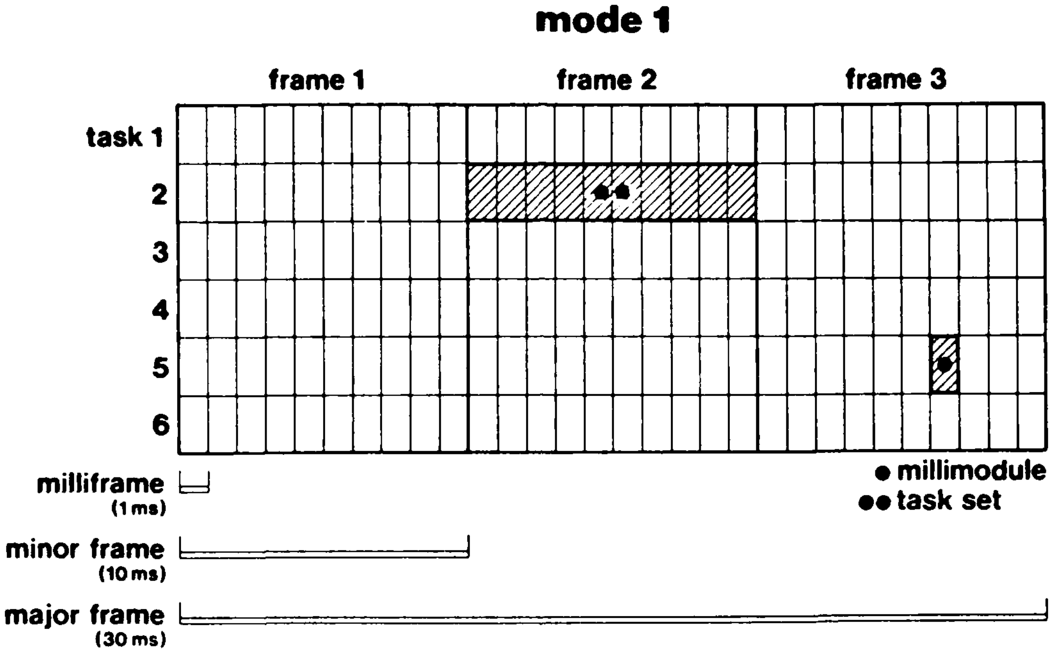


Figure 18. CRMmFCS Task Assignment Chart

Another difference between the generic form of Figure 17 and the application form of Figure 18 is that the latter is actually a compound chart consisting of both major and minor frames. This is to allow reconfiguration to occur every ten milliseconds (at minor frame boundries) while maintaining the flexibility of a chart with a large number of module slots.

Real Time Chart Execution

Once a Task Assignment Chart has been laid out and all task sets defined, some method of translating the chart into executable software is required. This is done by storing the charts in tabular form in every processor's memory. Figure 19 shows the formation of a "task assignment table" from the task assignment charts for ten flight control modes.

In Figure 19, all system software has been divided into ten operating modes and presented in a large task assignment chart. Each mode is further divided into three consecutive frames with each frame divided into enough tasks for every processor in the system. Finally, each task consists of 10 millimodules which are executed consecutively by a single processor assigned to that task. Each task is identified by its row location within a particular mode and frame.

At the beginning of each new frame, all processors "renegotiate" their task numbers and then go to the appropriate mode and frame of the chart to find out which group of ten millimodules to execute. Exactly how processors determine their current mode, frame, and task numbers will be discussed shortly. For now, it will be assumed that these numbers are "given" in order to simplify discussion of how they are used in the system.

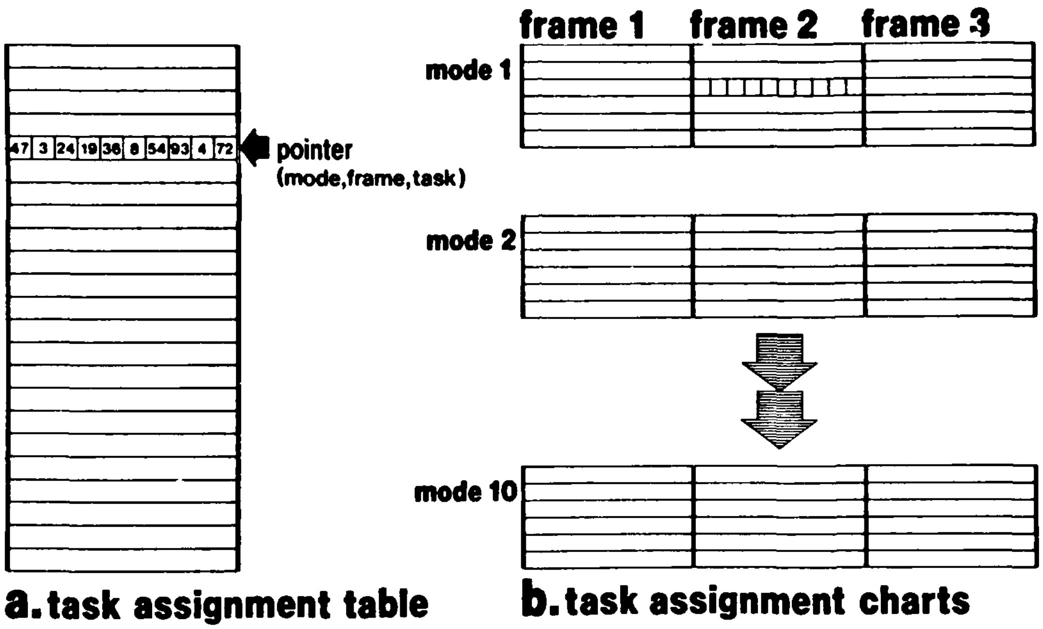


Figure 19. Task Assignment Table Generation

In Figure 19b, task assignment information was presented as a series of charts designed to allow easy manipulation by the system programmer. Once these charts have been completed, they are reorganized into a single task assignment table as shown in Figure 19a. This table is just a four-dimensional array of millimodule call names located within a block of memory in every processor. Each processor maintains a pointer in the table to the next millimodule it is supposed to execute. This pointer is a function of four variables including MODE, FRAME, TASK, and MODULE. These variables point to a single word in the table which contains the call address of the next millimodule to be executed.

Executive software in the system is now very simple. During one of the milliseconds near the end of each frame, every processor "volunteers" for its next task assignment. Using a special autonomous control algorithm (discussed in the next section) each processor determines its next task number (TASK). It then looks up the current MODE and FRAME values in virtual common memory and sets the millimodule count (MODULE) equal to one. This defines a location in the task assignment table, TABLE(MODE, FRAME, TASK, MODULE), which contains the call address of the next millimodule to be executed. The executive then calls this address and one millisecond of software is executed on schedule.

When control returns from the current millimodule, the MODULE count is incremented and the address of the next millimodule is obtained from the table. This address is called and another millisecond of software is executed. This continues until ten modules have been completed at which time all processors volunteer for new task numbers (causing total system reconfiguration) and the process begins again. Figure 20 shows a flow chart which indicates how simple the process becomes when the task assignment chart approach is used.

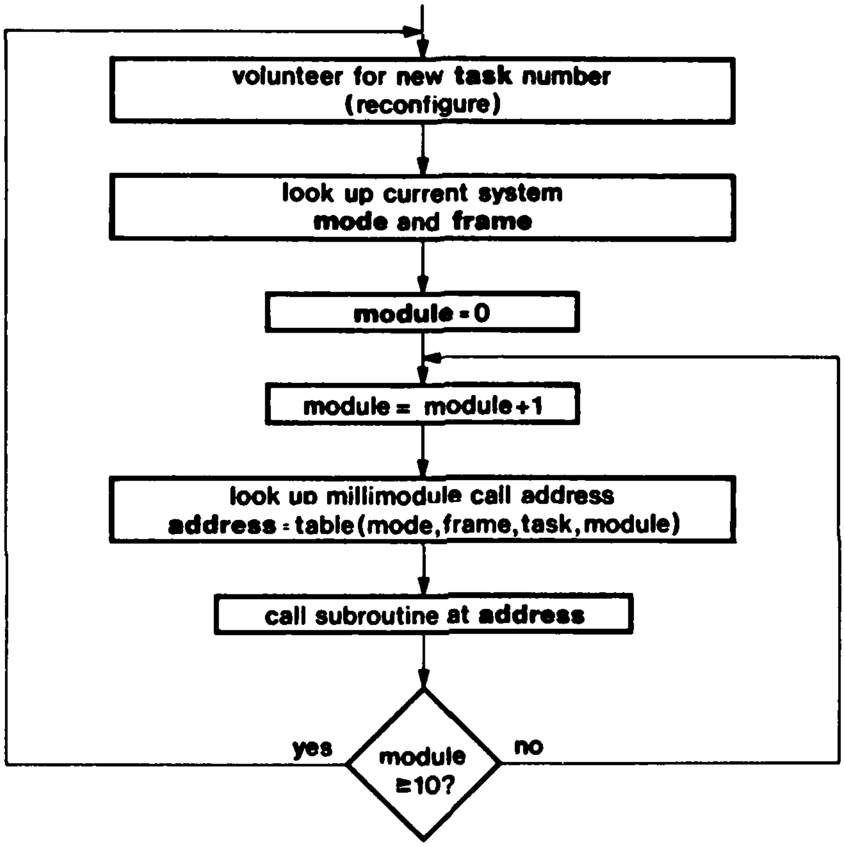

volunteer for new task number
(reconfigure)
look up current system
mode and frame
module = 0
module = module+1
look uo millimodule call address
address = table(mode,frame,task,module)
call subroutine at address
yes
module
≥10?
no

Figure 20. Executive Flow Chart

Multi-Rate Considerations

In the discussion thus far, the task assignment chart has been described as a schedule of tasks to be performed by a set of processors over a period of 30 milliseconds. This schedule is then repeated continuously as long as the system remains in the same mode. As a result of this characteristic, every module scheduled in the chart also repeats at a fixed rate of once every 30 milliseconds. Clearly, some technique is required to allow millimodules to be scheduled at other arbitrary rates.

Figure 17 showed how faster rates could be achieved by placing the same millimodule (k1) in the chart more than once. Unfortunately, there is only a limited number of rates for which this will work. Figure 21 illustrates the problem.

In the figure, millimodule timing over a period of 30 milliseconds is shown for the case where a major frame is only 10 milliframes long. Every 10 milliseconds the entire process repeats as the system makes three passes through the TAC. Ten millimodules labeled A through J are shown repeating at intervals ranging from every millisecond for module A to every 10 milliseconds for module J. Inspection of the chart shows that only modules A, B, E, and J execute at uniform rates over the entire 30 millisecond period. This is because only the periods of 1, 2, 5, and 10 milliseconds divide into the major frame period of 10 milliseconds evenly. All other rates do not divide evenly and therefore lose synchronization at every major frame boundry. Thus, in a 10 milliframe TAC, only rates of 1, 2, 5, and 10 can be scheduled (without resorting to the compound millimodule approach discussed later).

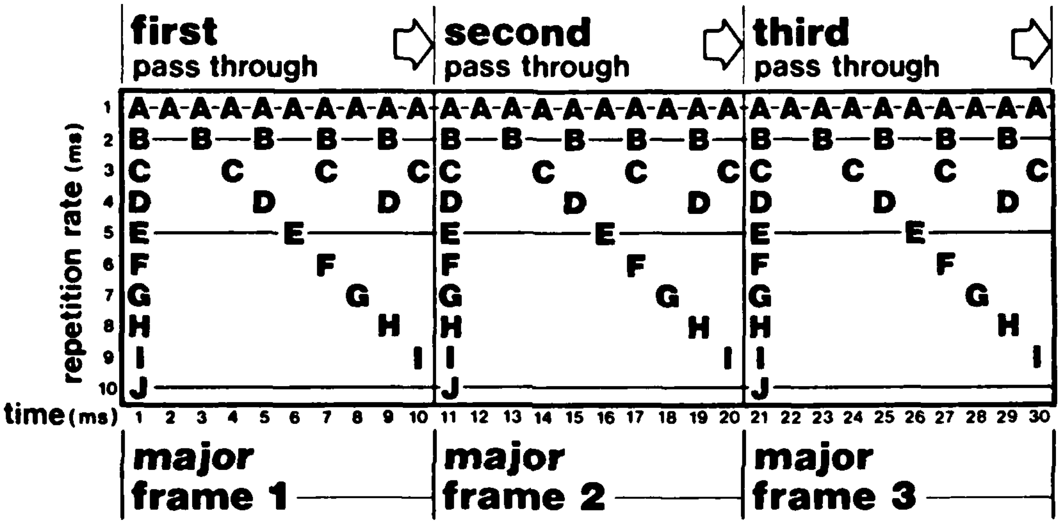


Figure 21. The High-Rate Task Scheduling Problem

If the major frame length is increased to 20 milliframes, a larger number of rates are made possible. These rates include 1, 2, 4, 5, 10, and 20 milliseconds all of which divide into the major frame rate evenly. Similarly, for a 30 milliframe TAC rates of 1, 2, 3, 5, 6, 10, 15, and 30 are allowed. Fourty milliframe TACs allow 1, 2, 4, 5, 8, 10, 20, and 40 millisecond iteration rates, and so on. A 30 milliframe TAC was chosen for the CRMmFCS architecture because it provided the greatest number of rates while minimizing the size of the required task assignment tables. This 30 millisecond chart was then divided into three 10 millisecond minor frames for reconfiguration purposes. Processors volunteer for one minor frame at a time and complete a single pass through the TAC every three minor frames.

Compound Millimodules

To provide rates slower than the major frame rate, the concept of a "compound millimodule" must be introduced. A compound millimodule is simply a module which does not execute every time it is called. A modulo 2 millimodule, for example, executes only every other time it is called. By placing a compound millimodule in the chart at some legal repetition interval (one that will divide evenly into 30), and adjusting its modulus appropriately, absolutely any repetition rate can be obtained. (See Appendix G.)

C. AUTONOMOUS CONTROL ALGORITHMS

Now that the system software has been defined using the task assignment chart, all that is needed is a set of algorithms to implement autonomous control and continuous reconfiguration. This section discusses how these algorithms were designed in the CRMmFCS architecture.

Autonomous Control

Autonomous control is a method for dynamically distributing tasks among a group of processors without using a central controller. It is defined as "a scheme whereby each processor independently determines its own next task based upon the current state of the system". In the CRMmFCS architecture, every processor has access to all system state information through the virtual common memory. All that is needed is an algorithm which uses this information to determine each processor's next assignment.

In order for a processor to know which row of the task assignment table (Figure 19a) to execute during a given frame (reconfiguration cycle), three pieces of information are required. These include the current system mode, MODE, the current frame, FRAME, and each processor's own specific task number, TASK. The values of MODE and FRAME are continuously updated in virtual common memory by an ordinary set of millimodules placed in the task assignment chart for that purpose. In general, the values of MODE and FRAME computed can be complex functions of any variable or set of variables available in virtual common memory. The specific portion of the TAC which is executed at any given time can therefore be made to vary with the instantaneous values of aircraft flight condition, pilot switch settings, configuration, and so on.

Since the values of MODE and FRAME may be looked up in virtual common memory at any time, all that remains is for processors to determine which task they are to perform within the given mode and frame. This is accomplished through a process known as "volunteering."

Volunteering

If system reconfiguration was not required, it would be possible to simply assign a different TASK number to each processor for all time. Each processor would then always execute the same row in the TAC for every frame and mode in every time period. But reconfiguration is required, not only in the event of a failure, but continuously in the case of the CRMmFCS architecture. Thus, a new value of TASK must be computed during every reconfiguration cycle.

The algorithm used to do this computation may be summarized as follows. At the beginning of each reconfiguration cycle, every processor does a brief self-test to determine if it is healthy . If it passes this test, it volunteers for a new assignment by setting a flag in the volunteer status table (VST). The VST consists of a memory location for each processor in the system serving as the volunteer status flag for that processor. If a flag is set to "1", the corresponding processor is known to have volunteered for task assignment. Otherwise, the processor is assumed to have failed and is not included in the next reconfiguration.

After all processors have volunteered, each one examines the VST to determine its next assignment. It counts the number of healthy processors ahead of it in the table, assumes they will each take a task, and selects the next available task number for itself. Figure 22 shows how this works.

Figure 22a shows the volunteer status table for a system of ten microprocessors. In the table it may be seen that processors 1, 2, 4, 5, 7, and 10 have volunteered for duty while the remaining processors (3, 6, 8, and 9) have

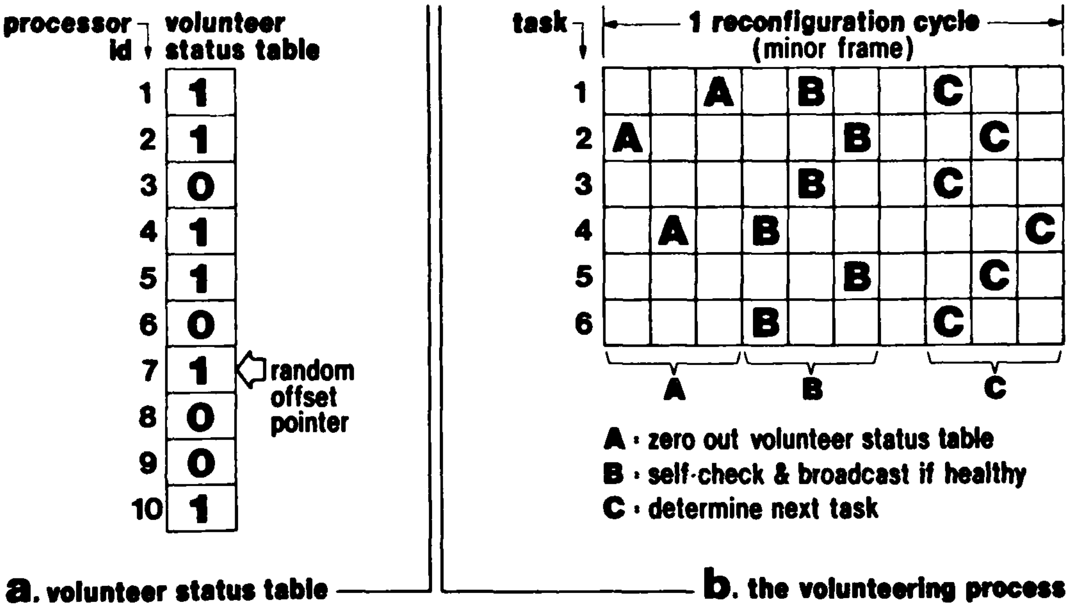

processor id
volunteer status table
1 1
2 1
3 0
4 1
5 1
6 0
7 1
8 0
9 0
10 1
random offset pointer
a. volunteer status table
task
1 reconfiguration cycle (minor frame)
1 A B C
2 A B C
3 B C
4 A B C
5 B C
6 B C
A
B
C
A : zero out volunteer status table
B : self-check & broadcast if healthy
C : determine next task
b. the volunteering process

Figure 22. Volunteering and the Volunteer Status Table

not. Given this information, every processor in the system is capable of independently determining its next assignment. For example, when processor 5 examines the table it finds that processors 1, 2, and 4 have also volunteered and presumably taken the first three tasks. It therefore concludes that it is responsible for task number 4. In a similar manner, processors 7 and 10 select tasks 5 and 6 respectively.

In the task assignment chart of Figure 22b, there are only six tasks to be performed by the 10 processors in this system. If all ten processors were healthy, the first six would perform the tasks and the remaining four would act as spares. In the event that a processor fails to volunteer, all those below it in the table move up one task and the first spare in line takes over task 6. Four processors have failed in Figure 22 leaving just enough to do the required tasks. If one more fails, only the first five tasks in the chart will be taken and task six will not be done. For this reason, tasks are placed in the TAC in order of priority so that the least important ones are dropped first.

Now that the general concept of volunteering has been introduced, the details of its implementation may be presented. Figure 22b shows the specific millimodules involved in making the process work.

Sometime during the first three milliframes of each reconfiguration cycle, a set of type "A" millimodules are executed. Each A module is responsible for storing zeros in all ten locations of the VST to clear it for the next round of volunteering. Three copies of the module are included for redundancy purposes to make sure that the table is fully cleared. During the next three milliframes (whenever it is convenient for the programmer), every processor spends one millisecond doing a type "B" millimodule. The function of a

B module is to do a brief self-check and volunteer if the processor is healthy. It is the type B module which places ones in the volunteer status table. Finally, during one of the last three milliframes in each cycle, all processors execute a type "C" millimodule to determine what task to do during the following frame. Type C modules examine the VST and count ones to determine the value of TASK for each processor. This value and the values of FRAME and MODE from virtual common memory specify exactly which set of ten modules in the task assignment table (Figure 19a) are to be executed during the next frame.

Continuous Reconfiguration

The scheme discussed thus far provides a simple method for achieving autonomous control of task assignment in a system of pooled microprocessors. It does not, however, quite meet the requirements of continuous reconfiguratation. In its current form, task assignments only change when there is a failure in the system. When all processors are healthy, number 1 always gets task 1, number 2 gets task 2, and number 10 is always a spare. Continuous reconfiguration requires that task assignments be randomly redistributed among all functioning processors during each cycle so that spare checkout and latent fault detection can be provided. This section discusses how a simple modification to the current task assignment algorithm can make this possible.

The modification required involves only a slight change to how module C uses the contents of the VST. Instead of counting the number of ones between itself and the start of the table, the count is taken relative to a "random offset pointer" located somewhere in the table. In Figure 22a, the random offset pointer is set at location 7 in the table. All processors then compute their next assignments relative to this location. Processor 5, for example, begins counting healty processors at location 7 and concludes that processors 7, 10, and (wrapping around to the top of the table) processors 1, 2, and 4 are ahead of it during this frame. Processor 5 therefore takes task 6 for its next assignment. If the offset pointer had been at 4, processor 5 would have taken task 2, and so on.

Because the random offset pointer is recomputed every frame, processor task assignments are continuously redistributed in a pseudo-random manner. This pseudo-random redistribution is sufficient to ensure that, over a period of time, every processor gets to perform every task. Continuous reconfiguration has been achieved.

There are a number of ways to generate the random offset pointer. A brute-force approach is to assign three millimodules (for redundancy) to execute a pseudo random number routine. These modules would then store their results in virtual common memory where all processors could compare them and arrive at a common pointer number.

A more elegant method is to simply use the least significant bits of some rapidly changing state variable already located in virtual common memory. Since all processors have access to this variable, it is a simple matter to use it to compute a common offset pointer. This is the approach taken in the CRMmFCS design.

D. RELIABILITY CONSIDERATIONS

Up to this point, very little has been said about how computational errors are eliminated from the system. It has been assumed that the rudimentary self checks performed in the process of volunteering were infallable and that only guaranteed healthy processors are ever given tasks to do. This final section discusses what steps have been taken to ensure that the system operates reliably.

In the first place, every processor is isolated from the bus by a "combination lock" called the bus access gate (BAG). In order to gain access to the bus, a processor must successfuly complete a self check routine ("B" in Figure 22) during which it generates a combination to unlock the BAG. Once unlocked, the BAG remains open for 20 milliseconds until it is relocked by a watchdog timer circuit. If a processor fails to generate a correct combination at periodic intervals, the BAG will lock it off the bus and prevent it from volunteering for further tasks. Complete details on the BAG circuit are present in Appendix A.

A second level of fault tolerance is provided through the use of triplex data and triad millimodules. Three processors are assigned to execute each millimodule by simply including three copies of each module in different rows of the task assignment chart. These in turn generate three copies of each variable which are stored at three locations in virtual common memory. All processors then perform a quick vote on each triplex variable when they obtain it from virtual common memory. In this manner, bad data is suppressed until the processor which generated it can be eliminated from the bus.

The third level of the "fault filter" involves use of a virtual common memory black mark table containing every processor's opinion of every other processor. When a processor finds a piece of bad data in the virtual common memory, it adds a black mark against the processor which generated it. If sufficient black marks accumulate against a processor from more than three of its peers, its combination generating algorithm will be unable to produce the right combination to unlock its BAG. This prevents the processor from volunteering for any more tasks and provides an effective means for a group of processors to "pull the plug" on a bad one.

A final level of protection is continuous reconfiguration itself. Because a processor constantly changes tasks, the errors it may produce never accumulate at any one output. Instead, they are scattered among all outputs where they can be suppressed by triplex voting until the processor is eliminated by black mark accumulation.

E. SECTION SUMMARY

This section has presented a summary of the approach used to manage software in the Continuously Reconfiguring Multi-Microprocessor Flight Control System. It represents a collection of techniques, not all of which are new, that allow systematic implementation of continuous reconfiguration and autonomous control while maintaining as much software simplicity and modularity as possible. The reader is referred to the appendices for additional specific details on the various software aspects of the CRMmFCS design.

SECTION V
LABORATORY IMPLEMENTATION

Up to this point discussion has centered around the theoretical design of the CRMmFCS architecture but very little has been said about the actual laboratory implementation of the system. This section describes the prototype multi-microprocessor flight control system which is currently under construction at the Flight Dynamics Laboratory.

The purpose of the laboratory implementation was to provide a means to test, evaluate, and demonstrate the CRMmFCS concepts discussed in the previous sections. Data gathered from this in-house program will be used to quantify the extent to which expected benefits and limitations of the architecture exist. The laborat^ry model will also be used to validate a detailed software simulaticn of the entire system. This validated simulation will be used to project the throughput, fault tolerence, and other quantifiable characteristics of modifications to the baseline hardware without actually building them.

The in-house facility, shown in Figure 23, has been designed to maximize data gathering, data reduction and programmability of the system. The basic CRMmFCS architecture is represented by the bus termination circuit, real-world interface module, and six processing modules shown in the figure. The remaining blocks represent an aircraft simulator, a cockpit CRT display, and data gathering, reduction, and software development facilities.

A processing module consists of a 16-bit microcomputer, 8K words of memory, and custom engineered transmitter, receiver, and state information memory (SIM). The custom

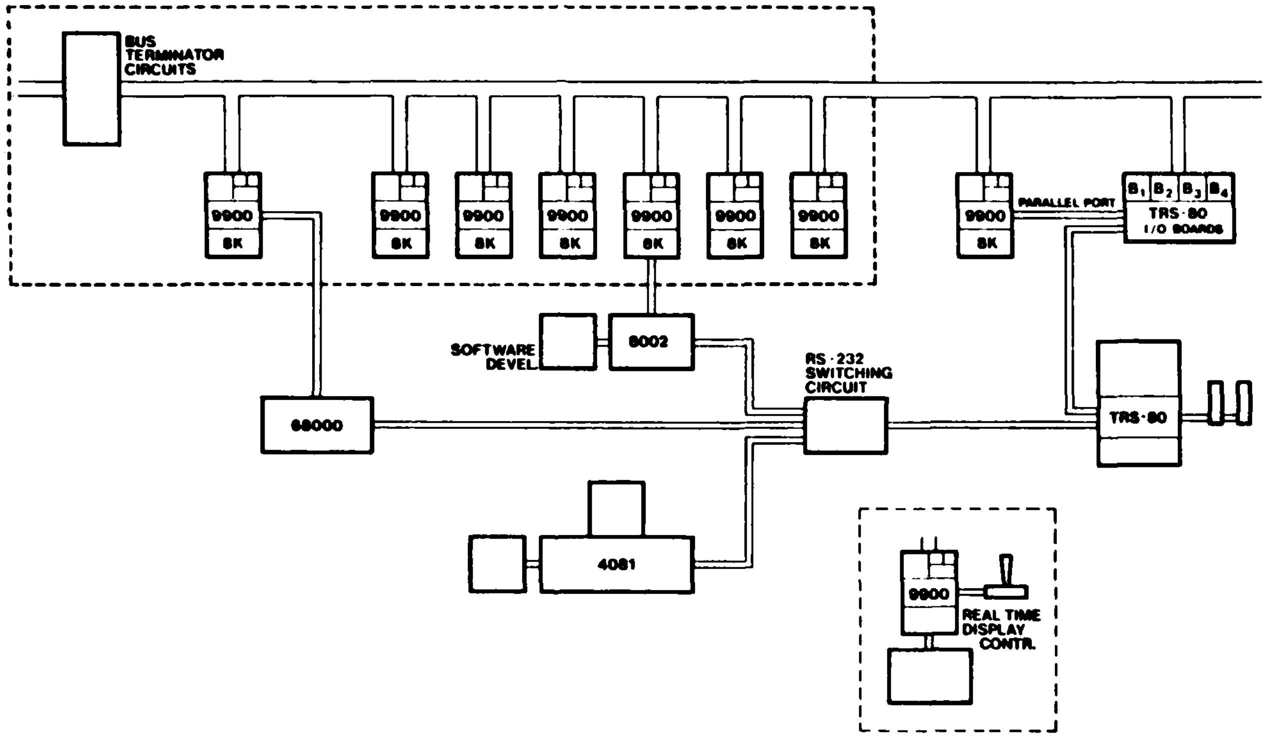


Figure 23. Laboratory Implementation

circuitry uses small and medium scale integrated circuits and is described in detail in Appendices A and B.

The block labeled "68000" is an advanced 16-bit microcomputer which will be used for a single axis digital aircraft simulation. It is interfaced through a dedicated processing module (see Appendix D) to demonstrate one method of accessing external system components such as sensors and actuators. A follow-on effort will use an analog computer to do more complex aircraft simulations.

A Tektronix 8002 microprocessor development system is also shown in Figure 23. It is used for both hardware and software development. The 8002 interfaces to the CRMmFCS hardware by replacing the microprocessor chip in one of the processing modules with its own connecting cable. This allows direct control of that module for debugging and evaluation purposes. It also permits downloading software to the rest of the system through that module's transmitter circuitry. The 8002 transmits software over the global bus to every processing module prior to a simulation run. After the run, it is used to make software modifications based on data gathered during the simulation. The new software can then be rapidly redistributed and the system brought up for another run.

A Radio Shack TRS-80 microcomputer is used in conjunction with a dedicated processing module and custom serial bus interface to gather data during a simulation run. The processing module (or SIM monitor) records the history of specific variables in the SIM during each test run and then transfers the data to the TRS-80 for further processing. The serial bus interface is used to gather raw data from each of the four serial busses which the TRS-80 then processes to pinpoint specific problems and to determine bus utilization and system throughput. The TRS-80 also controls the RS-232 switching circuit which allows data and software to be easily transferred between the major components of the test system. Further details on data collection are available in Appendix E.

Also included in the laboratory setup is a microprocessor-based color graphics display which can be configured as a cockpit instrumentation display or used to monitor the system status in real time. The display controller also has a joy stick input which can be used in more advanced aircraft simulations. The real time display demonstrates the ease with which the architecture can be interfaced to other aircraft subsystems. (See Appendix D.)

The Tektronix 4081 is a stand alone minicomputer with graphics capability and a link to a main frame computer. It is used for further data reduction and display and for the development of complex software for the millimodule compiler and software simulation.

SECTION VI
CONCLUSION

There are three major potential benefits to designing a flight control system using the methods described in this report. The first is expandability as system needs grow. It is a well known fact that from the time the first model of a particular aircraft rolls off the assembly line until the last one lines up in mothballs, there are inumerable changes that occur to the system. This causes excessive increases in cost due to the difficulties of changing hardware and adding new software to the system. The CRMmFCS approach has the potential to greatly reduce these costs. Modularity of both hardware and software should allow much more expandability.

A second potential benefit is the ability to reduce software costs which are the single biggest cost in digital systems today. By designing an architecture that is inherently easier to program, the cost of programming, maintaining, and updating software should be greatly reduced. This contributes to a reduction in life cycle costs.

The third potential benefit is the possibility of avoiding unscheduled maintenence. With the present redundant flight control computers, if any component of the computer has failed the aircraft is not allowed to take off. As digital technology progresses, it will become practical to configure the CRMmFCS with as many as one hundred processors. If only 40 processors are required to accomplish the necessary processing, then there will be 60 spare processors. A requirement that at least 20 spares be available when the aircraft takes off leaves 40 processors that can fail before the aircraft is grounded. When scheduled maintenence occurs, any failed processors can be replaced. Since it is unlikely 40 processors will fail between maintenance periods, the long sought goal of no unscheduled maintenance may actually be achieved.

APPENDIX A

BUS TRANSMITTER DESIGN

In Section III of the report, an overview of the transmission circuit used by each processor to talk on the global bus was presented. This appendix presents further details on the transmitter design as they exist at the time of this writing. Although subsequent research may change some of the specifics of this design, it is expected to remain substantially as described below.

Transmitter Function Review

The purpose of the transmitter is to relieve the processor from the burden of obtaining access to the bus and formatting data for transmission. Figure A-1 shows the main components of the transmitter in block diagram form.

The transmitter works in the following manner. First, the local processor loads one of its two output buffers (Page 1 or Page 2) with data to be transmitted during the next milliframe. When the next milliframe begins, the block labeled "transmission control logic" (TCL) takes control of that page and unloads it onto the global bus. To do this, it must first locate an available bus. It instructs the "bus availability detector" (BAD) to locate a free bus and then unloads one transmission word from the buffer through a shift register onto that bus. It then tells the BAD to find another free bus and the process continues until the output page is empty. After that, the transmitter sits idle until the start of the next milliframe at which time the local processor supplies it with another page of data for transmission. The following paragraphs describe each of these steps in detail.

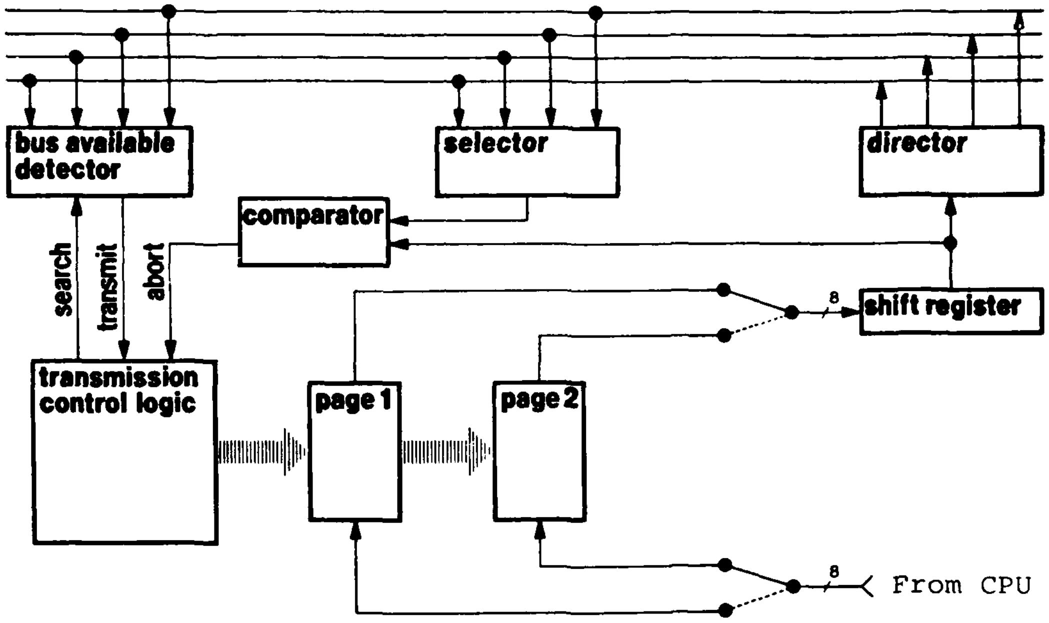


Figure A-1. Transmitter Block Diagram

Transmission Format

In order to understand transmitter operation, it is helpful to first discuss the format of the transmission which it generates. Figure A-2 shows that a single transmission word consists of three parts including (1) a five-bit source identification field, (2) an eleven-bit variable name field, and (3) a sixteen-bit variable value field. Together these three fields add up to 32 bits of meaningful data in each transmission.

The source identification field is different for each transmitter in the system. It serves two functions. First, it identifies which of up to 32 different transmitters originated the message. This is useful for failure isolation and is used by system voting algorithms for blackballing purposes. The second use for the identification field is that it makes messages from different transmitters begin differently. This allows transparent contention to resolve bus collisions in less than five microseconds. (See Section III of the report.)

Each transmitter is given its own unique ID at the time of system assembly. In the CRMmFCS laboratory model, this is accomplished by burning a different number into the EPROM (programmable read-only memory) associated with each processing module. In an operational system, this would more likely be accomplished through hardware jumpers on each circuit board or by coding the connectors into which each board plugs.

The second part of each transmission is the variable name field. This field determines into which SIM location the variable will be stored. Since the name field is 11 bits long, it can specify up to 2048 different state variables.

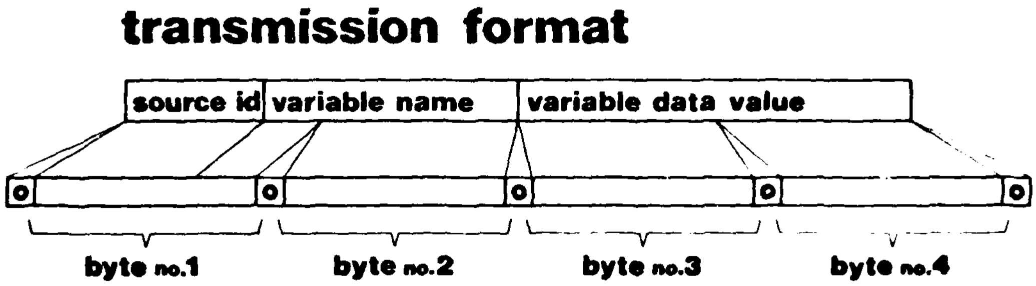


Figure A-2. Transmission Format

Finally, the last 16 bits of meaningful data make up the variable value field. This is the actual value stored in the SIM at the address specified by the address field. It may be in any format (even part of a floating-point number) depending upon its application.

Up to this point, only the 32 meaningful transmission bits have been discussed. The actual format of data appearing on the bus is slightly different. In Figure A-2, the 32-bit message has been divided into four 8-bit bytes and a zero has been inserted between each byte. These zeros are used to make the bus look busy throughout the duration of the transmission. This requirement is clarified in the next section.

The Bus "BUSY" Signal

At the time the transparent contention scheme for autonomous bus control was being developed, a requirement emerged for independent processors to be able to tell if at any time a particular bus was being used. Since the normal state of a wired-or bus when not in use is a continuous stream of logic ones, a natural thing to conclude when a zero is observed was that the bus is currently in use. This fact was used to develop a "BUSY" signal which would allow a transmitter to instantly determine the availability of any particular bus.

The BUSY detector is simply a circuit which counts ones on the data bus and turns off after eight of them have occurred in a row. Each time a zero is detected, BUSY is turned on and the count is reset to zero. As long as zeros occur often enough to keep the ones count from reaching nine, BUSY stays on and no other transmitter will try to use that bus. Once a transmission is over, there are no more

AD-A101 412   AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB OH   F/G 9/2
A CONTINUOUSLY RECONFIGURING MULTI-MICROPROCESSOR FLIGHT CONTRO--ETC(U)
MAY 81   S J LARIMER, S L MAHER

UNCLASSIFIED   AFWAL-TR-81-3070   NL

2 OF 2
AD
A101 412

END
DATE
FILMED
8-81
DTIC

zeros to reset the detector so it counts on up to nine and the BUSY signal shuts off. Every transmitter that is waiting for a bus is then free to try to use it and, through transparent contention, one of them gains control and completes another message.

Most 32-bit transmissions (ID, name, and value) are made up of a fairly well-mixed collection of ones and zeros. This means that zero bits would usually occur often enough so that the BUSY signal would stay on during the entire transmission. Unfortunately, this is not always the case. For example, if transmitter number 11111 sends variable number 111 1111 1111 with a value of 1111 1111 1111 1111, then the transmission would consist of 32 consecutive ones. After the eighth bit, the BUSY signal would disappear and other transmitters would be free to transmit right on top of the last 23 bits of the message. (Actually, the BUSY signal would not even appear during the first 8 bits because there were never any zeros to set it in the first place.) Clearly, some means is needed to set BUSY at the beginning of every transmission and to keep it set until the end. The zeros shown inserted between each byte in Figure A-2 perform this function.

A complete transmission, therefore, consists of 37 total bits including 32 significant bits and 5 zeros spaced throughout. Each transmission is also trailed by nine ones which occur as the bus floats high while the BUSY counter times out in every transmitter. Thus, the effective length of a transmission is actually 46 bits. Using this definition of word length, CRMmFCS transmitters pack 46-bit words onto the bus one after another with no wasted time in between and 100% efficient bus utilization has been achieved.

Of course, it is possible to argue that the 9 trailing ones after each 37-bit message do constitute wasted time on the bus. In one sense this is true, but on the other hand, they constitute a very simple and reliable set of "stop bits" for the transmission. In the future, more sophisticated circuit designs may be able to optimize this format, but the basic concept will still remain valid.

The Processor Interface

Now that the format of the transmitter output has been specified, it is possible to discuss the nature of each transmitter component. This section describes the interface between the transmitter and the processor which uses it.

The purpose of the transmitter is to relieve its processor of the burden of formatting data and obtaining access to the bus. It was designed to look as much as possible like a block of common memory to which the processor can write. For this reason, the interface between them has been implemented as a simple shared memory.

As was shown in Figure A-1, the shared memory is divided into two pages of 256 words each. These pages alternate functions every millisecond at milliframe boundries. While the processor writes its output for the current milliframe to one page, the other page (containing data from the previous milliframe) is unloaded by the transmitter onto the bus. This process is controlled by the transmission control logic (TCL) which maintains counters and pointers to keep track of the data in each page.

Throughout the discussion to follow, the page which is connected to the processor will be called the "input page" and the one connected to the transmitter the "output page."

Loading the Input Page

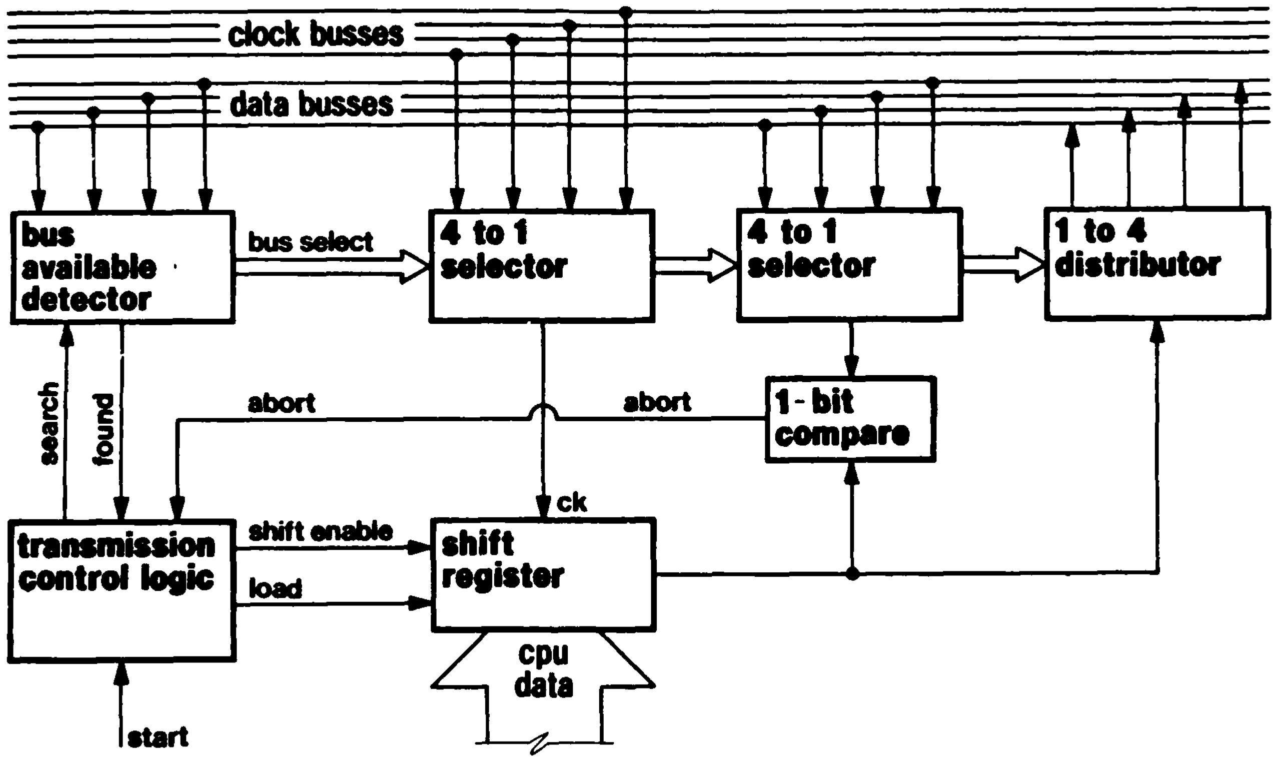
This section describes how a processor fills the input page with data to be transmitted. The procedure requires four consecutive writes to the page, one for each byte in the message (see Figure A-2). The first byte contains the five source ID bits and the first three variable name bits. The second contains the remaining eight name bits. Bytes three and four contain the most significant and least significant eight variable value bits respectively. Since there are 256 bytes of memory in a page, the current implementation allows the processor to generate up to 64 four-byte messages per millisecond.

After all messages for a given milliframe have been loaded, the processor writes one more byte to the input page. This byte consists of all zeros and serves as a signal to the transmitter that all messages have been unloaded. This "end of data" byte is identical to the first byte of a potential transmission by processor 00000 of a variable whose name begins with 000. For this reason, no transmitter in the system is given an ID number of 00000. That way, when five zeros do occur in the first byte of a message it can only mean one thing: the end of data in the output page has been reached.

Unloading the Output Page

While one page is being loaded by the processor, the other one is unloaded by the transmitter onto the bus. This section describes how data is removed from the output page and prepared for broadcast.

Figure A-3 shows a more detailed view of the output portion of the transmitter. The arrow labeled "cpu data" represents the eight output lines from the output page of



Figurc A-3. Transmitter Circuit

the interface buffer. This arrow is shown entering the inputs of a parallel-in/serial-out (PISO) shift register. The shift register is actually nine bits long with its first bit permanently hardwired to a logic zero. The remaining eight bits contain one byte from the output page.

This configuration allows very simple generation of the desired transmission format. When the transmission control logic (TCL) locates a free bus (see next section), it generates a LOAD command which causes the PISO to load nine bits (a hard-wired zero and one data byte from the output page). It then enables the shift function of the register and the bus clock shifts these bits onto the data bus. Nine clock pulses later the shift register is empty and the control logic loads another nine bits from the output page. This process repeats four times resulting in 36 bits (4 bytes and 4 zeros) of the message being placed on the bus. The TCL then allows one final shift from the now empty (full of zeros) PISO register. This places the 37th bit of the transmission (a zero) onto the bus and the transmission is complete.

After the last bit of the message has been placed on the bus, the TCL disables the shift register, increments its pointer to the next byte in the output page, and instructs the bus availability detector (Figure A-3) to find another free bus. This is the subject of the next section.

Bus Availability Detection and Bus Selection

In a previous section of this appendix the bus BUSY signal was introduced as an indicator of whether or not a particular bus was in use. This BUSY signal was turned on by the appearance of a zero on the bus and remained on until eight consecutive ones occurred. There is a separate BUSY indicator circuit for each bus in the system.

The "bus availability detector" (BAD) shown in Figure A-3 is simply a circuit which monitors the BUSY signal from each bus. When it is instructed to SEARCH for a bus by the TCL, it latches its BUS SELECT lines onto the first bus it finds that is not BUSY. It then signals the TCL that it has "FOUND" a bus and the TCL transmits one message from the output page.

The BUS SELECT lines which are set by the BAD circuit are connected to three selector circuits as shown in Figure A-3. These circuits include a 4 to 1 clock selector, a 4 to 1 data selector, and a 1 to 4 data distributer. These circuits are used to connect the transmitter to whichever bus is to be used for the current message.

The bus availability detector is instructed to find a new bus after every message. As a result, the messages from the output page are automatically distributed at random over every available bus in the system. If one bus fails, its termination circuit (Appendix C) pulls it permanently low. This makes it look constantly busy so that the bus availability detector never locks onto it. Every message is automatically routed onto one of the other working busses.

Transmitter Error Protection

Another function of the transmitter is to ensure that only good data is placed on the bus. For this reason, a one-bit comparator constantly monitors what is actually on the bus and compares it to what is supposed to be at the PISO register output. If there is ever a mis-compare (caused by noise, hardware failure, or a contending transmitter) the circuit generates an ABORT command which resets the transmission control logic and causes it to restart the message when it finds another free bus.

In order to prevent false ABORT signals caused by delays between the output of the PISO and actual appearance of a bit on the bus, the output of the comparator is only sampled on the falling edge of the bus clock. Since the register output changes only on the rising edge of the clock, a full clock pulse width (500 ns) is allowed for bus data to settle before it is used.

When an ABORT does occur during a transmission, one of two things can happen to the part of the message which has already appeared on the bus. If the abort was caused by noise or failure, the immediate halt in transmission leaves an incomplete message on the bus. Messages which are less that 37 bits long without all 5 separating zeros in place are automatically discarded by the receiver circuit (Appendix B) and cause no further problems. Aborts which are caused by disagreement with another transmitter sending at the same time are also no problem. Since the part of the message reaching the bus prior to the abort matched that of the contending processor exactly (or an abort would have occurred sooner), it simply becomes part of the contending processor's message and remains transparent to the system. The concept of transparent contention was discussed thoroughly in Section III of this report.

The Bus Access Gate

In any system where there is a large number of transmitters using the same bus, there is always the concern that a failed unit may "babble" on the bus preventing any other unit from using it. This section discusses how this is prevented in the CRMmFCS architecture.

The key to the CRMmFCS approach is a device known as the "bus access gate" (BAG). The bus access gate is the last output buffer between the transmitter and the serial data bus. It has the unique characteristic that it can be disabled by an external logic signal. (Almost any integrated circuit with a chip enable pin can be made to work.) In the CRMmFCS design, this external signal is called UNLOCK. Whenever UNLOCK is true, the BAG is enabled and data may pass through it from the transmitter to the bus. As long as a reliable means is provided to generate the UNLOCK signal, the BAG provides complete protection from one bad transmitter wiping out an entire bus. The next section discusses how this vitally important UNLOCK signal is generated.

Of course, it is possible to argue that the bus access gate itself may fail and wipe out an entire bus. While this possibility exists, its probability is sufficiently small (because of the circuit's simplicity) that it can be tolerated. For the purposes of reliability analysis, the BAGs in every transmitter are considered to be integral parts of the busses to which they are connected. In this sense, failure of a BAG is synonomous with failure of the bus itself. Sufficient spare busses (each with its own independent BAG interfaces to every transmitter) are provided so that such failures can be easily tolerated.

One final note must be made. It is not good design practice to make more than one BAG from a single IC package. Failure of such a package could disable every bus to which it is connected. For this reason, individual discrete components are recommended for construction of each BAG in every transmitter.

Generation of the UNLOCK Signal

Generation of the signal needed to unlock the bus access gate is a relatively simple matter. In every transmitter there is included a resettable 8-bit latch connected to the local processor as an output port. This port is referred to as the "combination register." An 8-bit comparator constantly compares the contents of this register with a hard-wired "correct" combination and its output is used as the required UNLOCK signal. As long as the register contents match the hardwired combination, the UNLOCK signal is true and the BAG is allowed to place transmissions on the bus. If the contents ever fail to match, the transmitter is locked off of every bus until its processor generates the correct combination and places it back in the combination register.

As one final level of protection, a "watchdog timer" circuit is included in every transmitter which periodically resets the combination register to all zeros. When this happens, a processor must successfully generate and store the correct combination back in the register before it can make any more transmissions. In the CRMmFCS, every processor must unlock its BAGs once every reconfiguration cycle (10 ms) in order to "stay in business."

This approach provides protection from a wide variety of possible failures. If a processor itself fails, it will be unable to generate a correct combination periodically and the entire module will be permanently locked off the bus by the watchdog timer. If the transmitter circuit fails, the processor will notice bad data accumulating in the SIM and can deliberately store a wrong combination to lock the BAG. Finally, if the processor fails only partially (a memory fault, for example) such that it can still generate combinations and transmit but some of its results are incorrect, it is possible for other processors to shut it down by destroying information in the SIM which it needs to generate its combination. This procedure is known as "blackballing".

Summary

This appendix has presented the conceptual details of the CRMmFCS transmitter design. It has shown that the transmitter contains circuitry which automatically competes for the bus, formats transmissions, and broadcasts them while checking for errors and rebroadcasting if one occurs. Although a working model has already been demonstrated in the laboratory, detailed schematics have not been included in this report because the final design is still being perfected. This information will be published in a future technical report.

APPENDIX B

BUS RECEIVER DESIGN

Introduction

This appendix contains a detailed description of the hardware design and the operation of the CRMmFCS receiver circuit. This circuit is a part of the flight control computer architecture which is being constructed in-house at the Flight Dynamics Laboratory. It was custom designed for the in-house effort and is implemented with small scale and medium scale integrated circuits.

Receiver Overview

The receiver consists of two major parts including a serial to parallel shift register (SIPO) and a block of random access memory (RAM) which contains a complete copy of all the state information in the system. This state information memory (SIM) is mapped into the microcomputer's address space as a block of "read only" memory and is accessed by the SIPO as a block of "write only" memory. Figure B-1 shows how this works.

The in-house CRMmFCS implementation utilizes four 1 Mhz data busses for data transfer and a TI-9900 microcomputer as the processing element. A block diagram of the receiver circuit is shown in Figure B-2. There are four identical bus receiver and control logic circuits designed to receive information from a data bus using the corresponding clock bus to synchronize data reception. The direct memory access (DMA) controller handles contention for the state information memory (SIM) between the receiver and the TI-9900 microcomputer.

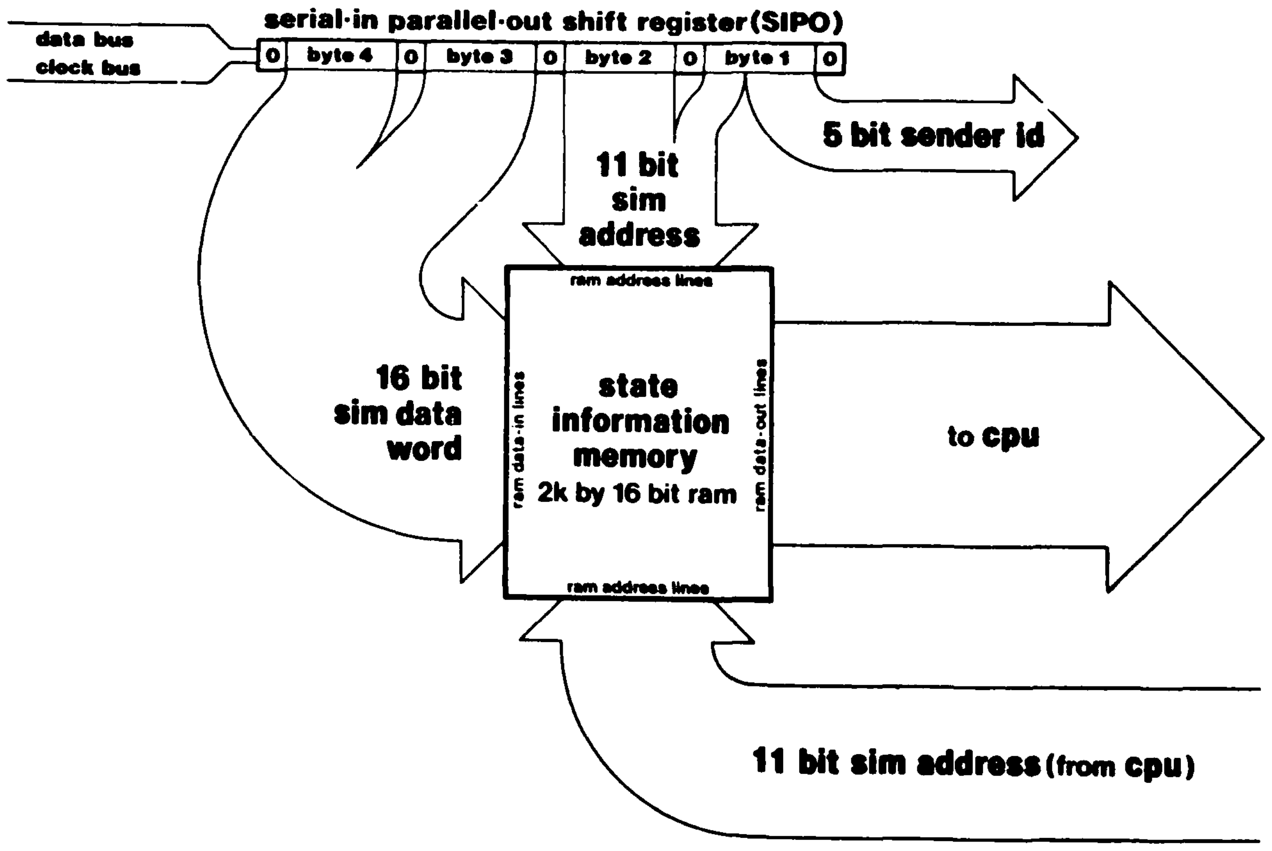


Figure B-1. Receiver Functional Diagram

Data Format

In the architecture under discussion, a word of transmitted data is 37 bits long and consists of four bytes of significant data separated by a zero bit before and after each byte (see Figure B-1). A string of more than eight consecutive "1" bits on the bus indicates that it is no longer in use, so zero bits are included between each byte to ensure that the bus continues to look busy in the event that more than eight consecutive ones occur in the actual data word.

Bytes one and two in Figure B-1 consist of five bits of source identification indicating the processing module which originated the transmission and an 11-bit variable name. The variable name is the address of the actual memory location in the SIM where the data is to be stored. Bytes three and four contain the variable data to be stored in the SIM.

Bus Receiver and Control Logic

The bus receiver (Figure B-2) consists of a 19-bit serial in parallel out (SIPO) shift register, one 11-bit latch for the address, and one 16-bit latch for data. Data is continually shifted into the SIPO by the bus clock.

Byte one and two boundaries (Figure B-1) are detected by monitoring bits 1, 9, and 18 of the incoming data stream. When these bits are simultaneously zero, as discussed in the section on data format, the receiver control logic immediately latches the ID and address information into the 16-bit data latch and the address information into the 11-bit address latch. After the information is latched, bits 1 through 17 of the SIPO are set to ones allowing the next 18 bits of information to be received. While new data

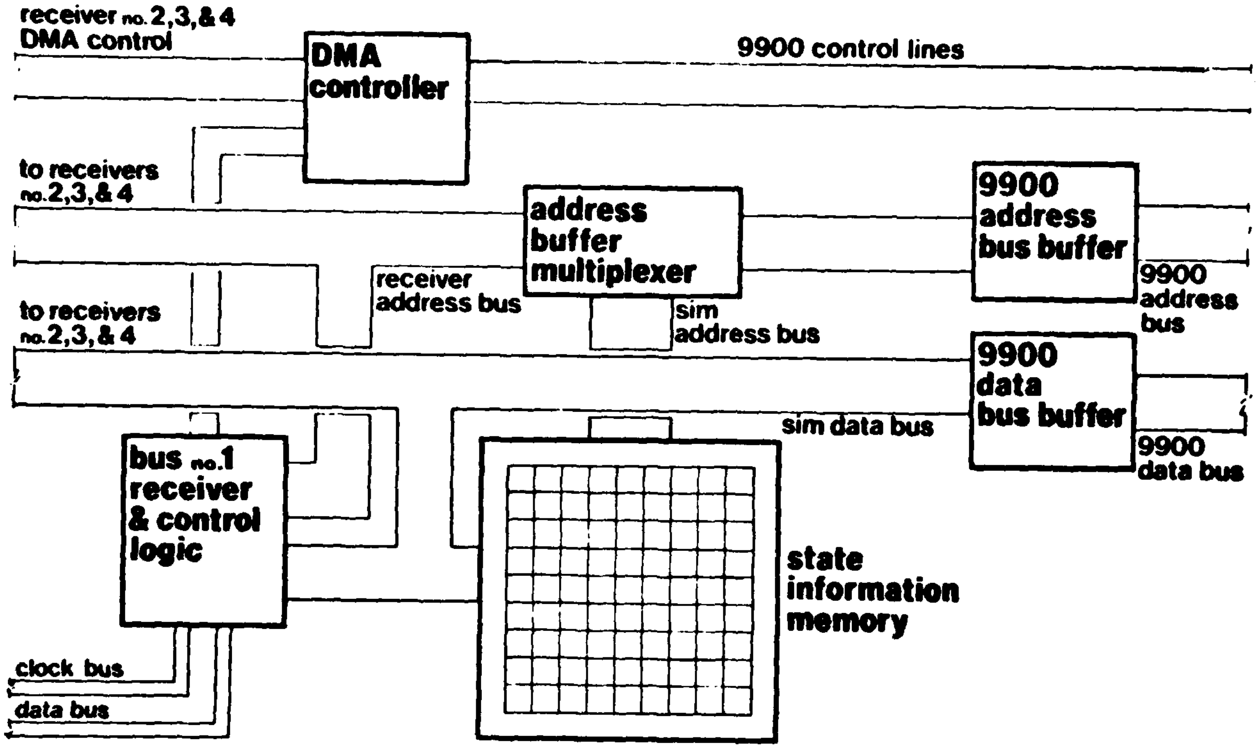


Figure B-2. Receiver Block Diagram

is being received, the DMA controller stores the ID/address information in the designated SIM location and increments the address by one. The DMA controller will be discussed in detail later.

Bytes three and four of the data word (Figure B-1) are shifted into the SIPO while the DMA controller is emptying the data latch. When the control logic again detects zeroes simultaneously at bits 1, 9, and 18 of the incoming data stream the information is latched into the 16-bit data latch. The address latch is unchanged this time. After the information is latched, bits 1 through 17 of the SIPO are set to ones and the receiver is then ready to receive a new ID, address, and data transmission.

Direct Memory Access Controller

The function of the direct memory access (DMA) controller is to transparently store data in the state information memory (SIM) as it becomes available from the receivers. It must also transfer information requested by the processing element (TI-9900 microcomputer) to the TI-9900 data bus at the proper time in the TI-9900 read cycle (see Figure B-2).

A priority encoder circuit is used to start the DMA cycle. The TI-9900 read signal is given the highest priority and the four receiver circuits are arbitrarily assigned the next four priorities. When a receiver buffer becomes full, it sends a logic one control signal to the priority encoder circuit. The priority encoder then activates the output corresponding to the highest priority input and disables further inputs until the end of the DMA cycle. The remaining inputs stay active until they have been serviced. The DMA controller detects the priority encoder output and enables the address and data bus of the indicated receiver and generates a memory write cycle for the SIM. Upon completion of the write cycle, the DMA controller resets the "receiver buffer full" indicator for the receiver which has just been serviced and enables the priority encoder circuit.

The DMA controller is designed to transparently service the TI-9900 no matter when the TI-9900 read request signal becomes active. This capability is achieved by using high speed memory in the SIM. The memory was chosen so that a DMA cycle is less than 300 nanoseconds. The TI-9900 memory read cycle is 660 nanoseconds. Therefore the DMA controller can complete a memory write cycle and still have enough time to service the TI-9900 read cycle.

Conclusion

The receiver circuit and SIM concept allows a large amount of information to be made available to the processing element in a multi-processor system. The processing element can treat the SIM as local memory and access any information it requires immediately. Also, it is conceivable that much higher data rates could be achieved with the receiver as it is presently implemented. Taking into account the maximum access rate of the TI-9900 the maximum total bandwidth of the receiver circuit is nearly 50 Megabaud. Calculating the bandwidth assuming it is not necessary to store the ID/address information and a lower access rate from the TI-9900 indicates a maximum possible bandwidth approaching 184 Megabaud.

APPENDIX C

BUS TERMINATION CIRCUIT

The bus termination circuit is an integral part of the CRMmFCS "smart" bus design. A "bus" actually consists of a data bus, a clock bus, and a bus termination circuit (see Figure C-1). The bus termination circuit serves four purposes. It terminates the data bus, monitors the clock and data busses for faults, generates the clock for the clock bus, and generates a milliframe synchronization pulse every millisecond.

Bus Termination and Fault Detection

Both ends of the data bus are terminated at the bus termination circuit. The wired-or data bus is terminated with pull-up resistors for impedance matching. This minimizes ringing on the bus and helps to suppress noise. A monitor circuit checks both ends of the bus for faults. If a fault occurs the bus termination circuit disables the data bus and corresponding clock bus. A timer circuit is then initiated and after 100 microseconds an attempt is made to restart the bus.

The bus termination circuit drives the clock bus with a one megahertz clock signal. The clock supplies the timing for data transmission over the data bus by the processing modules in the CRMmFCS. Both ends of the clock bus are terminated at the bus termination circuit so that the clock bus can be monitored for faults. As with the data bus, if a fault is detected the bus termination circuit disables both the clock and data bus and initiates a restart attempt after 100 microseconds.

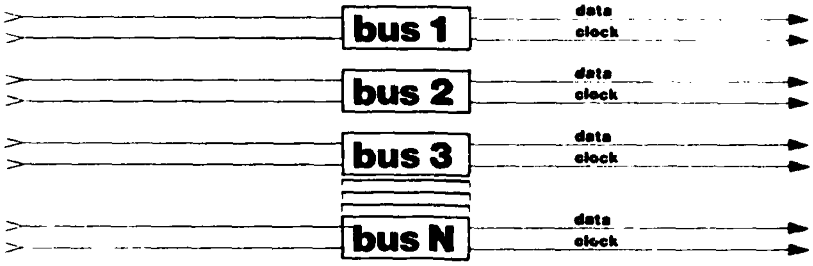


Figure C-1. Smart Bus Configuration

Frame Synchronization

Each bus termination circuit generates a frame synchronization pulse every millisecond. The pulse pulls the clock bus low for 5 microseconds. A missing pulse detector is used by each processing module to detect the frame synchronization pulse. This pulse synchronizes the processing modules to millimodule boundaries.

In the laboratory implementation, four sets of data and clock busses are used. The frame synchronization pulse of each bus termination circuit is internally tied to each of the other three bus termination circuits. Each bus terminator has a voting circuit which forces its synchronization pulse to be generated when two other bus terminator circuits simultaneously generate a frame synchronization pulse. The voting circuit acts as a master reset, synchronizing the frame synchronization circuits. The processing modules use a similar voting circuit, requiring that two frame synchronization pulses occur simultaneously before the millimodule boundary is accepted.

The voting circuits are used to maintain system sychronization while protecting the system from certain faults. For example if one frame synchronization pulse occurs at sporadic intervals the voting circuits prevent it from causing erroneous millimodule sychronization. Also, if the frame synchronization pulses occur at slightly different intervals the voting circuits will prevent the pulses from becoming slewed in time with respect to one another.

APPEՐDIX D

PEAL VORLD IITERFACE

This appendix addresses the problem of how to interface the CRMmFCS system to the outside world. Up to this point, it has more or less been assumed that all data needed by the system processors is somehow already available in the state information matrix (SIM). It has also been assumed that results stored in the SIM would somehow be able to influence the operation of other aircraft systems and control surfaces. In the following paragraphs a plan will be presented for interfacing the CRMmFCS to aircraft sensors, actuators, and displays. A discussion of how this technique will be demonstrated in the laboratory is also included.

Smart Sensors and Actuators

The key to the approach is the assumption that all sensors, actuators, and displays will be "smart." That is, every device has associated with it a processor of sufficient power to perform the necessary conversions of real world signals to and from virtual common memory format.

For example, every sensor has an associated processor which knows how often to sample its output, what signal processing to do, and where to place the resulting information in the state information matrix. For this reason, each sensor processor must have its own transmitter in order to broadcast the information into the virtual common memory where any other processor can access it. It may also be desirable to give each sensor processor a receiver circuit so that it may access the SIM for information on the current aircraft mode, flight condition, blackball table, and other data which may influence the sensor processor's performance.

Similarly, each actuator, display, or other output device has an associated processor which controls it. This processor accesses the SIM for information on what it is supposed to do and uses other SIM information to accomplish it. For this reason, each output processor must have a receiver circuit to access the SIM. It may also be desirable for it to have a transmitter so that it can report on its own health and the state of its output device.

In the most general case, each sensor, actuator, and display could have its own dedicated processor and transceiver circuitry interfacing it to the CRMmFCS global bus. Each device and its processor would then be considered to be a single peripheral unit and failure of either part would result in shutdown of the entire unit. System redundancy and fault tolerance capability would then be designed at the unit level.

This is not at all, however, the only configuration allowed in the CRMmFCS. It is equally possible to have one very large and powerful and reliable "real world interface processor" responsible for sampling all sensors, controlling all actuators, and doing all associated processing. It would then be necessary to use only one transmitter and receiver pair to interface to the real world.

In actual practice the system designed will probably lie somewhere in between the two extremes described above. Each sensor may be connected to several processors and each processor may be able to perform several different I/O (input/output) tasks depending upon system requirements. The important thing is that, no matter how the real world devices are sampled or controlled, they may be easily interfaced to the CRMmFCS through a standard transmitter and receiver pair. Even an external MIL-STD-1553 multiplex bus may be interfaced if one of its remote terminal processors has a CRMmFCS transceiver.

Laboratory Implementation

In the CRMmFCS laboratory prototype, only two I/O processors will actually be implemented. This is enough to demonstrate the concept while minimizing research costs. It is also enough to allow testing of an extention to the task assignment chart approach which will be discussed in the next section.

The two I/O processors which will be constructed in the laboratory will include an aircraft interface unit and a pilot interface unit. The aircraft interface processor (Figure D-1a) will be connected to a real time demonstration of aircraft dynamics and sensor/actuator characteristics. It will be responsible for sampling simulated sensor outputs and providing control commands through simulated digital-to-analog converters connected to simulated aircraft actuators.

The pilot interface processor (Figure D-1b) will serve two functions in the CRMmFCS prototype. First, it will sample a joystick input to allow reseachers to "fly" the system through a variety of maneuvers. This will allow an evaluation of system response and provide a certain amount of "hands-on" capability for demonstration purposes.

The pilot interface processor will also drive a CRT disp'ay, performing all the functions necessary to generate graphic and alphanumeric output. The display will operate in two modes. The first allows real-time readout of any combinaton of SIM variables. This is expected to be very

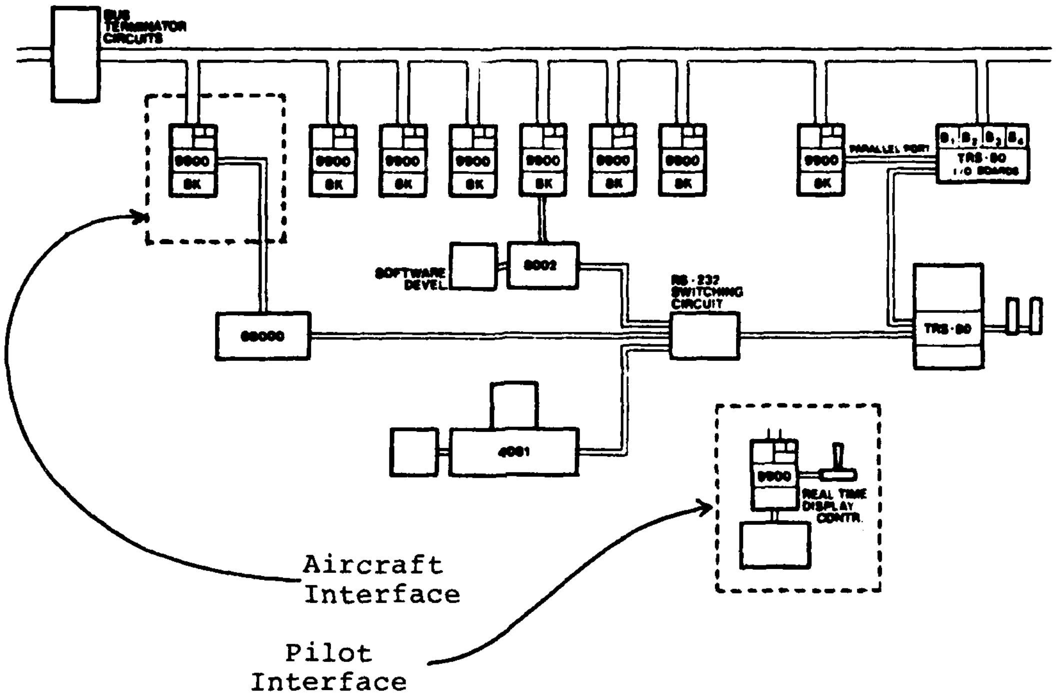


Figure D-1. Real World Interface Processors useful for debugging, testing, and evaluating system performance during actual operation rather than waiting until the run is over to study the collected information. The second display mode will generate a simple vertical situation display for hands-on demonstration purposes.

I/O Processor Software

Software for the CRMmFCS I/O processors will be managed in the same manner as for the rest of the system. Tasks will be divided into millimodules and a separate task assignment chart (TAC) will be used to schedule them. When a particular sensor is sampled will depend upon where its millimodule is placed in the TAC. Actuator and display updates will also be controlled by TAC scheduling.

Operating system software will change slightly for I/O processors because continuous reconfiguration will not be required. Since I/O processors may differ widely in capability (from powerful minicomputers for complex displays to simple 4-bit microprocessors for solonoid and relay control) and since different processors may be connected to different devices, constant redistribution of tasks among them will be impractical. For this reason, each processor will execute only one fixed row of the I/O TAC for each system MODE.

Of course, it is possible to visualize a system where every real-world signal is connected to every processing module (perhaps through a common motherboard). If this were true, then the control TAC and the I/O TAC could be combined and any processor in the system could perform absolutely any task. For now, however, separate task assignment charts and specialized I/O processors appear to be the more practical approach.

Data Flow Scheduling

The single biggest reason for taking a task assignment chart approach to I/O processing is to allow control over the scheduling of data flow on the bus. In Section III-E, it was shown how a "data flow assignment chart" could be used to prevent bus overloads and maximize bus utilization efficiency. It was also shown that there was a definite bandwidth limitation which allowed a maximum of 88 variables (in the current four 1 MHz bus configuration) to be placed on the bus per millisecond. If there is to be a slot on the bus for every variable generated, then this limit must not be exceeded. For this reason, I/O processor transmissions must be included in the overall system data flow assignment chart. The use of an I/O task assignment chart with synchronized milliframes greatly simplifies this scheduling problem.

APPENDIX E

DATA COLLECTION CIRCUITS

Introduction

In the design of any laboratory experiment, some means must be provided for collecting the necessary data. This is especially true if the experiment is a new micro-electronic system design where most of the data of interest consists of voltage levels which change in the sub-microsecond range. If anything other than the most simple input/output observations are to be made, some means must be provided for automatically collecting and processing selected data wherever it occurs. This appendix describes the provisions which have been made for data collection in the CRMmFCS laboratory implementation.

Data Collection Requirements

There are three major requirements for data collection in the CRMmFCS. The first is the ability to monitor what is going on inside the processing module itself. Some means must be provided to observe the processor's data and address lines continuously as it executes instructions and transfers information to and from memory. This function is provided by a Tektronix 8002 Microprocessor Development System which will be referred to as the "PM monitor."

A second requirement is some means to monitor traffic on the global data bus. A major part of the CRMmFCS research involves the precise timing and scheduling of large amounts of data on the bus by a great many free-running independent transmitters. Multi-trace storage oscilloscopes and high speed logic analyzers are helpful for part of this requirement. They can be used to measure voltage vs. time relationships and even many bit patterns but they fall short in one important area. There is a need to be able to record every bit that appears on the bus over an extended period of time and then process that information to put it in a form more usable to the human researcher. To meet this requirement, a special "bus monitor" circuit was developed. It will be discussed later in this appendix.

Finally, there is a very important need to be able to monitor all of the state variables in the system as they change with time. The state information matrix (SIM) stored in virtual common memory contains this data which changes constantly as information surges back and forth across the global bus. The "SIM monitor" circuit allows the sampling of any state variable or group of variables over very long periods of time at a variety of rates. It is data from the SIM monitor which is used to plot the time response of every variable in the CRMmFCS.

The PM Monitor

The Tektronix 8002 is a piece of laboratory test equipment designed to help develop and debug microprocessor-based systems. It plugs into the system under test using a ribbon cable in place of the system's own microprocesor chip and then emulates that chip to the extent that the system is totally unaware of the replacement. Everything operates as usual except that now the 8002 is capable of displaying every instruction executed and every bit of data manipulated by the particular module to which it is connected.

The 8002 allows the user to take over complete control of any processing module in the CRMmFCS system. Data collected may be sent to a CRT screen, to a line printer, or stored on disk for future reference. Since the 8002 meets every requirement specified for a PM monitor, no custom designed circuits were necessary for this part of the data collection system. All that was required was a simple RS-232 data link to send the collected data to the post-processor for analysis. The post-processor will also be discussed later in this appendix.

The Global Bus Monitor

The purpose of the bus monitor is to record every bit of data appearing on each of the four global busses implemented in the laboratory design. Figure E-1 shows its essential features.

Data Collection. The bus monitor consists of four 32K memory boards (Bl, B2, B3, and B4) which are connected to the four global busses via four custom serial to parallel conversion circuits (SIPOs). Each circuit collects data from a bus eight bits at a time and stores it using direct memory access (DMA) into its corresponding 32K memory bank. A counter is incremented after each store operaton to point to the next available location in memory. When it reaches 32K (32768), the counter overflows to zero and the circuit either stops collecting or continues to count overwriting previously stored values until it is halted by an external command. (Which option occurs is user selectable.) In either case, the net result is that each bank of memory is loaded with 32768 consecutive eight-bit samples of what appeared on its corresponding bus during the last run. This amounts to 262,114 bits of bus data or just over 0.26 seconds worth of transmission time.

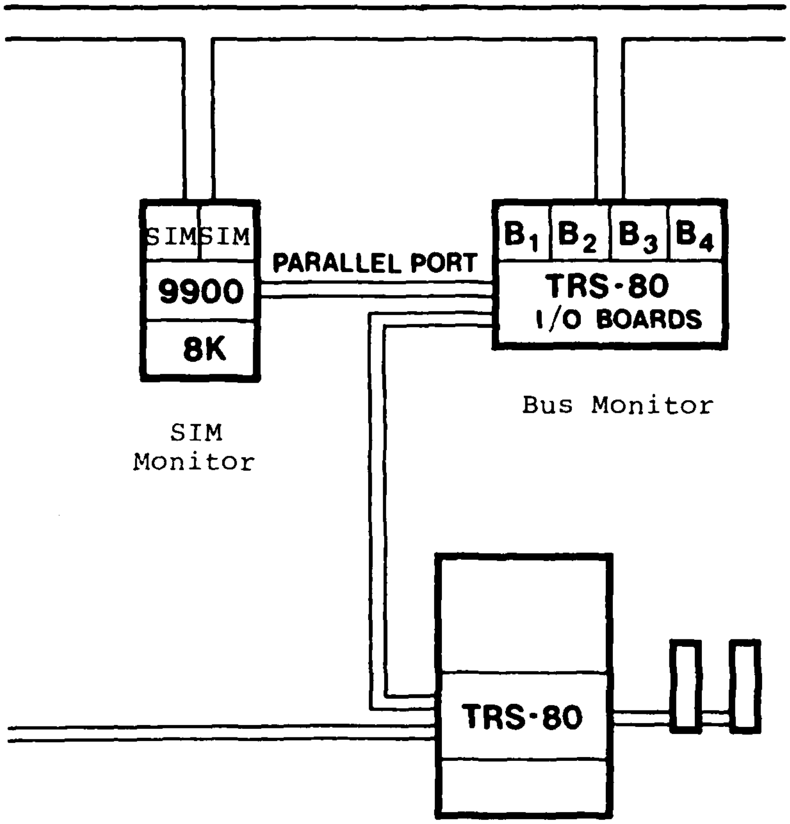


Figure E-1. Bus and SIM Monitors

While on the surface 0.26 seconds does not seem to be very much time, it actually allows the collection of an enormous amount of data. During that interval, the CRMmFCS has completed nearly 9 iterations of its control laws and has reconfigured over 26 times. This is more than long enough to determine what the bus monitor was designed to find out: when and where data appeared on the bus, if any faulty or incomplete transmissions occurred, and the efficiency with which the bus is being used.

Data Analysis. Once a 0.26 second "picture" has been taken of the bus by the SIPO circuits, some means is required to access and analyze the data stored in each memory bank. A Radio Shack TRS-80 microcomputer is used for this purpose.

Referring again to Figure E-1, each 32K buffer (B1, B2, B3, and B4) is memory-mapped into the upper half of the TRS-80's address space. By storing the appropriate number in an output port, the TRS-80 can access any buffer in a bank-select fashion. The information in each buffer may then be processed, displayed, stored on local five-inch floppy disks or transferred to a Tektronix 4081 mini-computer for further processing, mass storage, or high-resolution plotting. This post-processing of data will be discussed later.

The SIM Monitor

The third major piece of data collection equipment is also a custom circuit called the SIM monitor. It is responsible for recording the time histories of selected system variables over the duration of each test run.

The SIM monitor is just a standard processing module which has been modified to contain two receiver circuits and no transmitter. Its sole function is to listen to the bus and transfer variables of interest from the SIM to its own local memory. Two receivers have been provided so that one can listen to the bus while the other is being copied into local memory. Every millisecond, the receivers alternate functions allowing the local processor that much time to examine what the contents of the SIM were at the end of the previous millisecond.

The SIM monitor may be programmed by the TRS-80 to sample any number of variables in the system at any number of rates. The only limitation is the number of words a 9900 microprocessor can transfer per millisecond and the size of its local memory.

The SIM monitor is programmed from the TRS-80 keyboard. Prior to a run, the operator uses an interactive program (written in TRS-80 Basic) to select which variables are to be recorded and the rate at which they are to be sampled. After a run, the processing module dumps its local memory through a parallel port to the TRS-80 for processing, storage, and display. The final section of this appendix discusses how this data is processed.

Post-Processing

The TRS-80 microcomputer is the heart of the CRMmFCS data collection system. It is connected to the bus monitor through memory mapping, to the SIM monitor through a parallel port, and to the 8002 PM monitor via an RS-232 interface (Figure E-2). Unfortunately, the TRS-80 has only limited processing, storage, and graphics capability. For

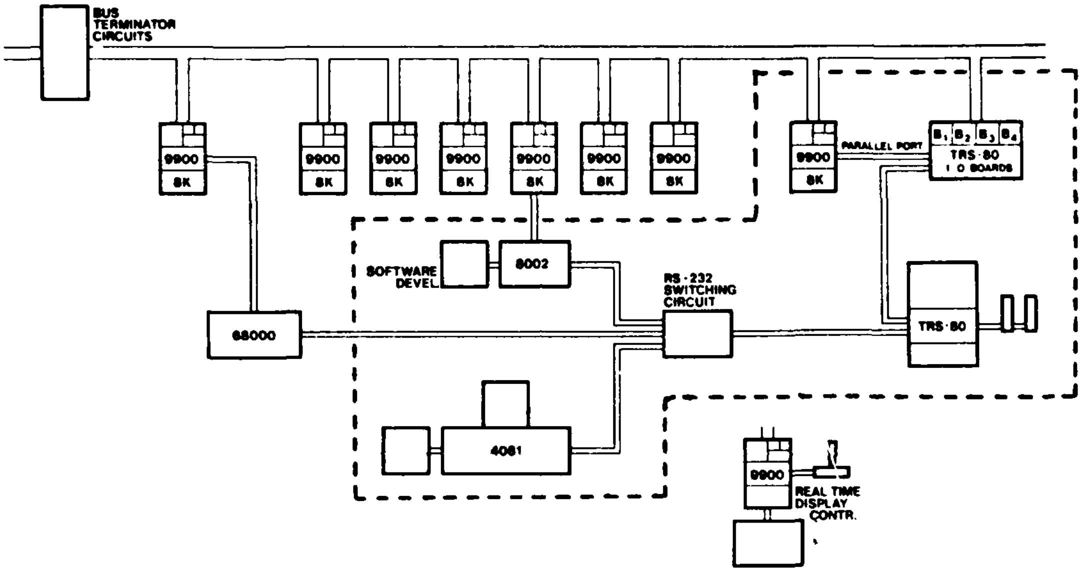


Figure E-2. Data Collection and Post-Processing Circuitry this reason, it was relegated to the role of data collection system manager and operator interface console. Through the TRS-80 and its custom designed RS-232 switching circuit, the entire CRMmFCS laboratory system can be controlled.

One of the devices connected to the TRS-80 by the RS-232 circuit is a Tektronix 4081 minicomputer. The 4081 is a stand-alone 32-bit computer with hardware floating point, hard disk storage, and superb graphics capability. All data collected in the system is routed to the 4081 for reduction, storage, and hard copy plot generation. It may also be routed from there to a CDC Cyber mainframe computer for even more processing and the generation of report-quality Calcomp plots.

Summary

This appendix has described the major components of the CRMmFCS data collecton and reduction system. This system consists of three data collecton devices (including PM, Bus, and SIM monitors), a TRS-80 system manager, a custom RS-232 switching circuit, and a 4081 post processor. All of these devices have been integrated into a single system designed to maximize the amount of information which can be obtained in the laboratory. Since the main reason for building a working model of the CRMmFCS was to generate this data, the development of an effective means to collect and analyze it is every bit as important as the construction of the flight control system itself.

APPENDIX F

STATE INFORMATION MATRIX THEORY

This appendix discusses the state information matrix (SIM), a concept which is fundamental to the design of the entire CRMmFCS architecture. The SIM is defined as "an n-dimensional array containing the current best estimate of all information available about the state and environment of an aircraft." It is designed to help manage the wealth of real time information available to a modern flight control system.

The SIM contains information about traditional state variables (rates, velocities, positions), cockpit switch settings, target states, air data, telemetry data, aircraft model parameters, and raw sensor data. It also includes a sufficient number of their past values to implement any required difference equations.

The state information matrix may be visualized as a set of post office "pigeon holes." As shown in Figure F-1, there is a separate cell for every piece of information known about the aircraft and its environment. Some cells contain only static information which never changes (such as aircraft dimensions and numerical constants). Others contain discrete information which only changes when a switch is thrown or a new mode is selected. Still others contain information that is continuously updated at a variety of rates.

There is nothing particularly revolutionary about the state information matrix. The concept of a state vector has been around for a long time. The state information matrix simply extends the concept of state (traditionally reserved

	u	v	w	p	q	r	ψ	θ	ϕ		
	$\mathring{u}$	$\mathring{v}$	$\mathring{w}$	$\mathring{p}$	$\mathring{q}$	$\mathring{r}$	$\mathring{\psi}$	$\mathring{\theta}$	$\mathring{\phi}$		
		M	ρ_o	ρ_s	$\bar{q}$			a_x	a_y	a_z	
								$\mathring{a}_x$	$\mathring{a}_y$	$\mathring{a}_z$	
C_{m_α}	$C_{m_{\mathring{\alpha}}}$	C_{m_q}	$C_{m_{\delta e}}$	C_{n_r}	C_{n_β}	$C_{n_{\delta a}}$					
C_{ℓ_α}	$C_{\ell_{\mathring{\alpha}}}$	C_{ℓ_q}	$C_{\ell_{\delta e}}$	C_{ℓ_r}	C_{ℓ_β}	$C_{\ell_{\delta a}}$			A MODE	B MODE	C MODE
									X NAV	Y NAV	Z NAV
$x(k)$	$x(k-1)$	$x(k-2)$	$x(k-3)$	$x(k-4)$							
$y(k)$	$y(k-1)$	$y(k-2)$					DIS x_1	DIS y_1	DIS x_2	DIS y_2	
$z(k)$	$z(k-1)$	$z(k-2)$	$z(k-3)$	$z(k-4)$	$z(k-5)$						

Figure F-1. The State Information Matrix for positions, rates, and accelerations) to include all information about the system. It also extends the dimensionality of the representation from a one-dimensional vector to an n-dimensional array.

A special case of the SIM, called the state-time form, is useful for introducing the concept. The state-time form is just a two-dimensional array of past and present state vectors. Figure F-2 is a generic example of the state-time form.

In the figure, the vertical axis contains each state variable of interest and the horizontal axis contains the past n values of each variable. Conceptually, this matrix is of infinite dimension along both axes. However, due to hardware limitations and actual requirements, it is necessary to implement only enough of it to keep track of variables of practical interest.

Processing the State Information Matrix

Now that all available information has been collected in one conceptual place, all that is necessary is to develop an effective way to process, distribute, and use it. This is the topic to be addressed next.

The processing which is required for a typical flight control system may be broken down into three general types:

Type 1: Processing raw sensor signals to obtain usable input data. (Signal conditioning, scaling, filtering, etc.)

Type 2: Processing current input data and past state information to obtain a best estimate of current state information.

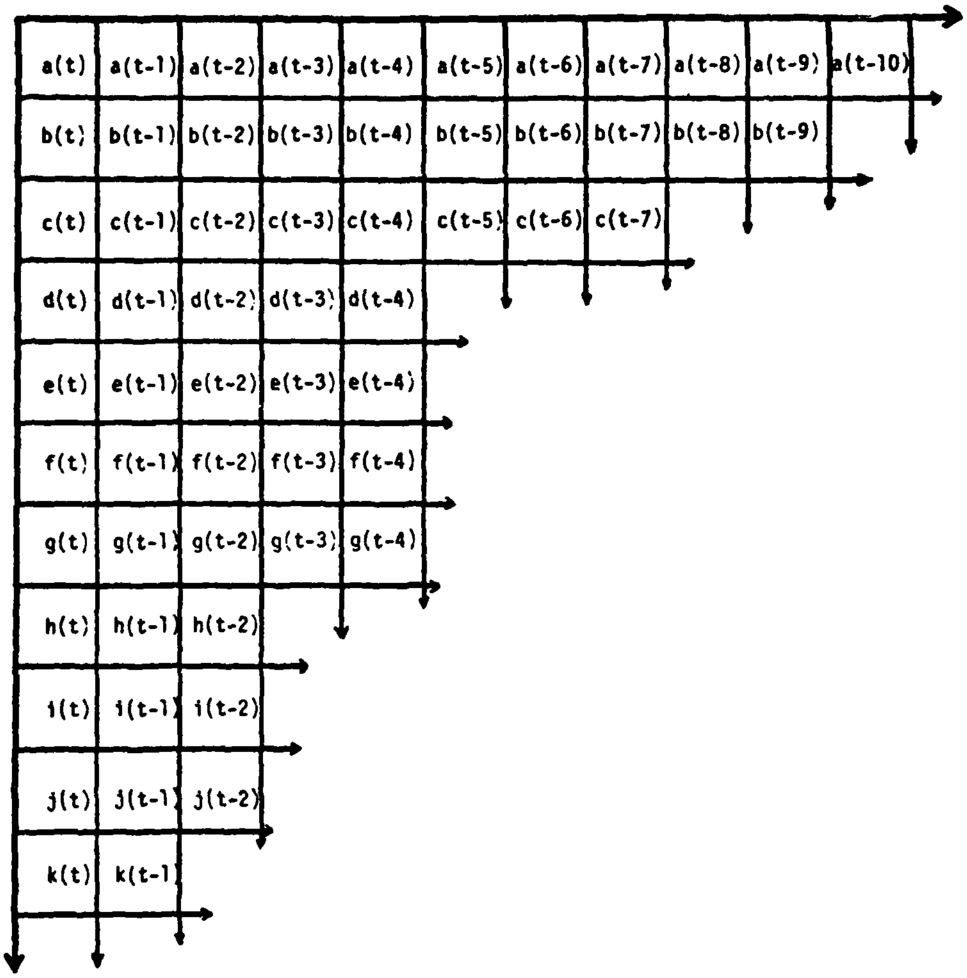

a(t) a(t-1) a(t-2) a(t-3) a(t-4) a(t-5) a(t-6) a(t-7) a(t-8) a(t-9) a(t-10)
b(t) b(t-1) b(t-2) b(t-3) b(t-4) b(t-5) b(t-6) b(t-7) b(t-8) b(t-9)
c(t) c(t-1) c(t-2) c(t-3) c(t-4) c(t-5) c(t-6) c(t-7)
d(t) d(t-1) d(t-2) d(t-3) d(t-4)
e(t) e(t-1) e(t-2) e(t-3) e(t-4)
f(t) f(t-1) f(t-2) f(t-3) f(t-4)
g(t) g(t-1) g(t-2) g(t-3) g(t-4)
h(t) h(t-1) h(t-2)
i(t) i(t-1) i(t-2)
j(t) j(t-1) j(t-2)
k(t) k(t-1)

Figure F-2. The State-Time Form

(Estimation, parameter identification, analytic redundancy, observers, etc.)

Type 3: Processing past and present state information to obtain the required control signals for all onboard systems. (Aero surfaces, displays, telemetry, etc.)

This partition divides the required processing into three sets of independent functions that (1) obtain data from all available sources, (2) squeeze all possible information out of that data, and (3) use this information to generate the "best possible" control signal for every onboard system. If these functions are shown in their relationship to the state information matrix, what results is a processing structure of the form shown in Figure J-3.

In the figure, $\underline{f}$ is a set of vector functions of Type 1. These functions process row sensor data to generate the elements, s, of the SIM that are functions of current sensor inputs only. The set of functions, $\underline{h}$, (of Type 2) generate the remaining variable elements, x, of the SIM as functions of s and past values of x. The entire SIM is then used by a set of Type 3 functions, $\underline{g}$, to generate the control output signals required by all aircraft systems.

This simple partition is important because it allows the separation of all processing into functions that are independent of each other and that require only simple interactions with the state information matrix. Using this approach, all functions in each of the three sets are (by construction) completely decoupled and may be designed independently by experts in each area. For example, sensor designers can be asigned the task of developing the individual $\underline{f}$ functions for each particular sensor subject

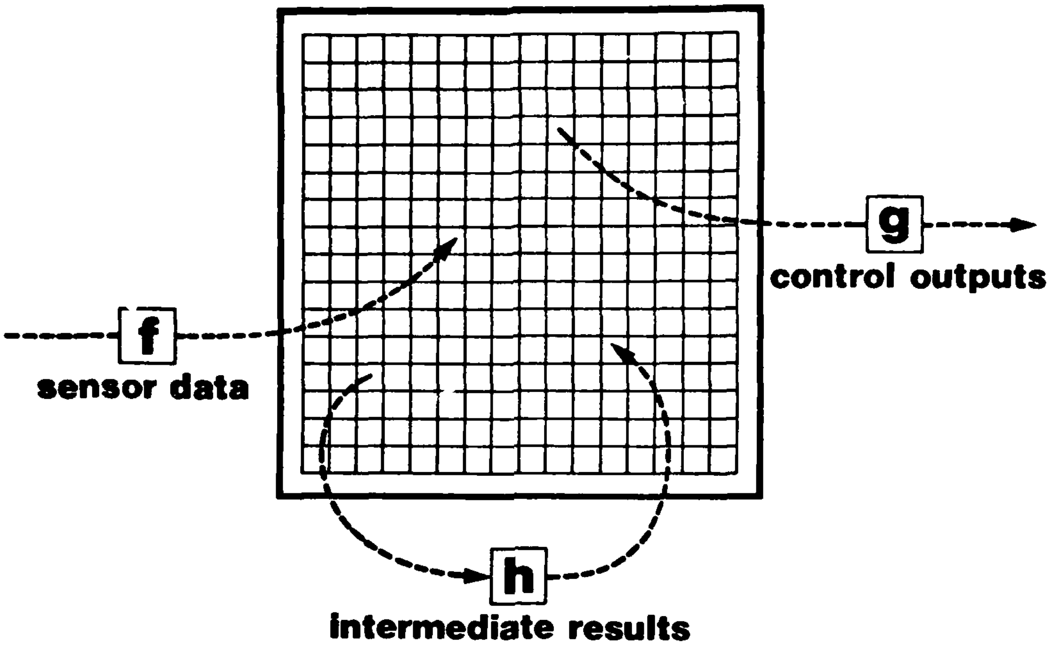


Figure F-3. State Information Matrix Processing compatible with the SIM. Similarly, control experts can design their algorithms knowing that the best possible estimate of every required state variable is readily available at a well-defined location in the state information matrix.

This approach can also have a high payoff in terms of maintenance and modification throughout the life cycle of the system. If a new and better estimation algorithm is developed, for example, it can simply replace the old algorithm and use the same inputs (or any other information in the SIM for that matter) to generate the same outputs for deposit to the same locations in the SIM. No other functions are affected or even notice the change, except that now some of the variables they are getting from the SIM are of higher quality.

This modularity, as always, is the key to conquering any large-scale design task. Modularity of software is a well-known and proven concept. Partitioning of processing into $\underline{f}$, $\underline{g}$, and $\underline{h}$ functions operating on a common set of data may enable a designer to take advantage of this fact.

Distributing the State Information Matrix

Up to this point it has more or less been assumed that the state information matrix is a set of "pigeon holes" somewhere with a label for every piece of information and the $\underline{f}$, $\underline{g}$, and $\underline{h}$ functions busily updating the contents of every hole. This is strictly true only for a standard uni-processor architecture where a section of memory may be set aside for exactly this purpose. For multi-processor architectures like the CRMmFCS, some other approach is necessary. The following paragraphs discuss some of the ideas that were evaluated in the development of the CRMmFCS architecture.

Single Copy / Multi-Access. It is possible to implement a multiprocessor architecture in which all processors share a common memory which contains the state information matrix (Figure F-4). While this approach may be the most straightforward, it has serious problems in terms of bus contention and throughput. This is because only one processor can access the memory at any one time.

Multi-Copy / Multi-Access. A brute force solution is to replace the single common memory with n common memories where any memory can be used by any processor (Figure F-5). This solves the contention problem and even provides a measure of inherent redundancy. However, it introduces the new problem of how to ensure that all memories contain the same current information. The cost, weight, and complexity of this method are also large.

Distributed Data Shared on Request. This is a common method used in many existing multiprocessor systems. Each processor has a portion of the SIM stored in its own local memory (Figure F-6). If a processor requires a piece of information not in its own memory, it simply asks for it and receives it over the bus a short time later. This method is simple and works well for many applications, but it runs into difficulty when large numbers of processors make bus contention a problem.

Broadcast Data Vector. Using this technique, the entire state information matrix is transmitted, one value at a time in a specific order, over the bus at periodic intervals (Figure F-7). Processors listening to the bus collect the variables they need as they come by and store them in local memory for use when needed. This makes maximum use of the bus bandwidth because no variable names need to be transmitted. Each variable is identified simply by its order of occurrence in the transmission.

Virtual Memory Emulation. This architecture is the one which eventually evolved into the CRMmFCS design. Each processor is given its own copy of the SIM which it accesses for all required data (Figure F-8). When a processor needs to update the SIM with a new value, it does not store it to the SIM directly. Instead, it broadcasts the value onto the global bus where it is received simultaneously by all SIM copies (including its own). Thus, storing to one SIM is the same as storing to all of them and reading from a local copy is the same as reading from any SIM (because they all contain the same data). This means that, as far as any one processor is concerned, there is only one SIM that is being transparently accessed by every processor in the system.

Summary

Regardless of how the state information matrix is distributed, it is important that it remain transparent to the application software. As far as anything but the operating system is concerned, the SIM is simply stored in a virtual common memory somewhere and accessed directly by a specific address or variable name (Figure F-9). This makes programming applications software very simple and allows the power of modular top-down programming techniques to be employed. It may be seen from the discussion above that the SIM concept is applicable in any application from conventional to the most exotic.

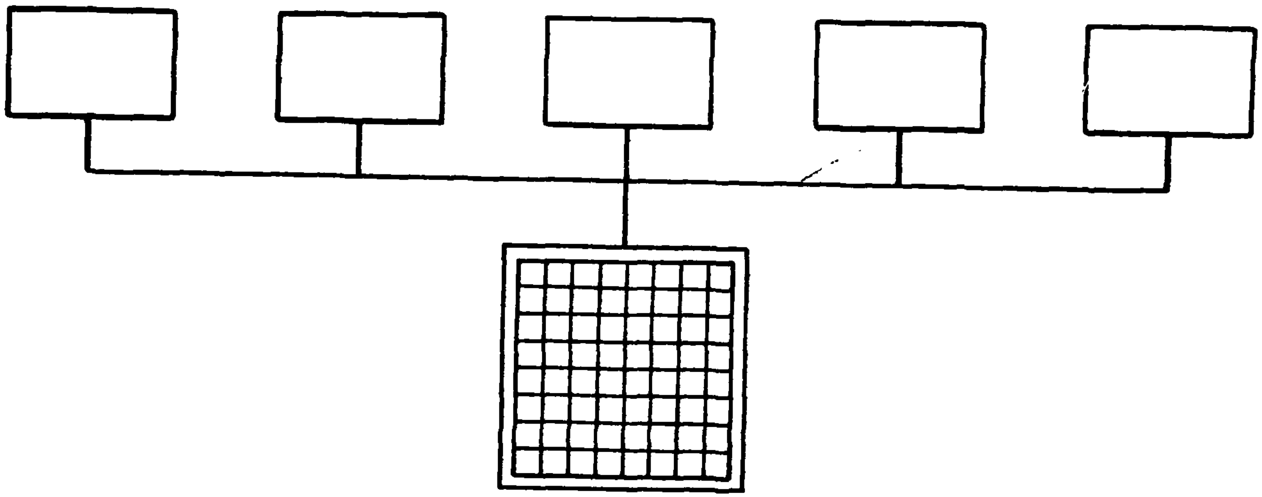


Figure F-4. Single Copy / Multi-Access Architecture

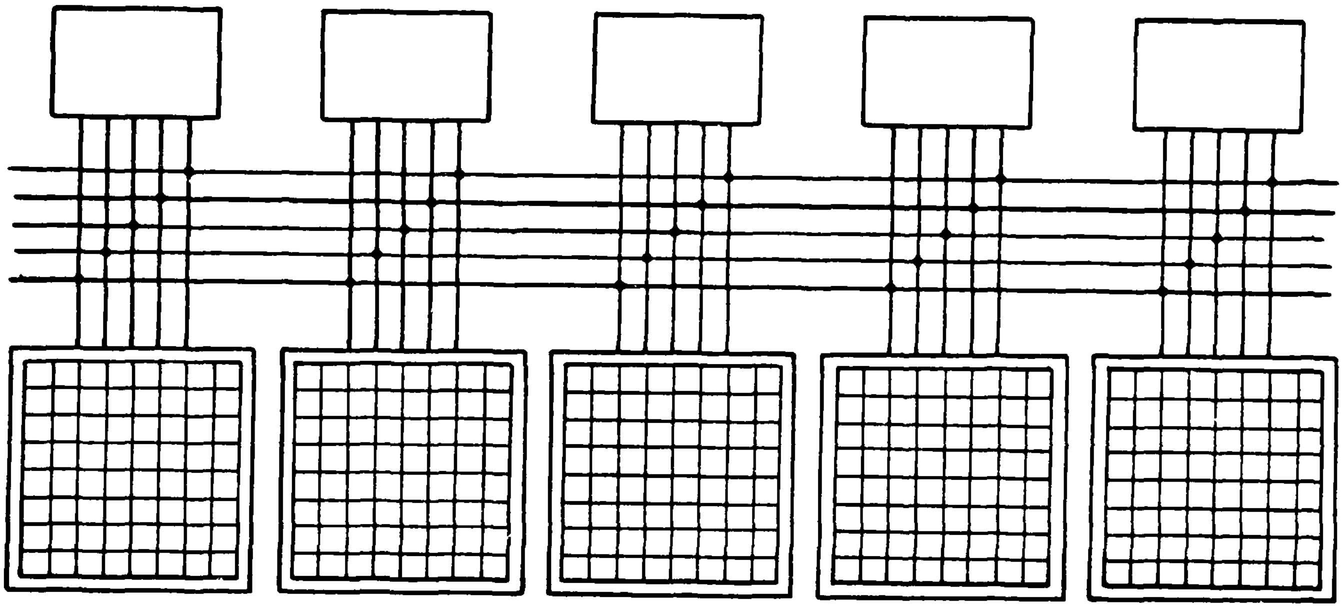


Figure F-5. Multi-Copy / Multi-Access Architecture

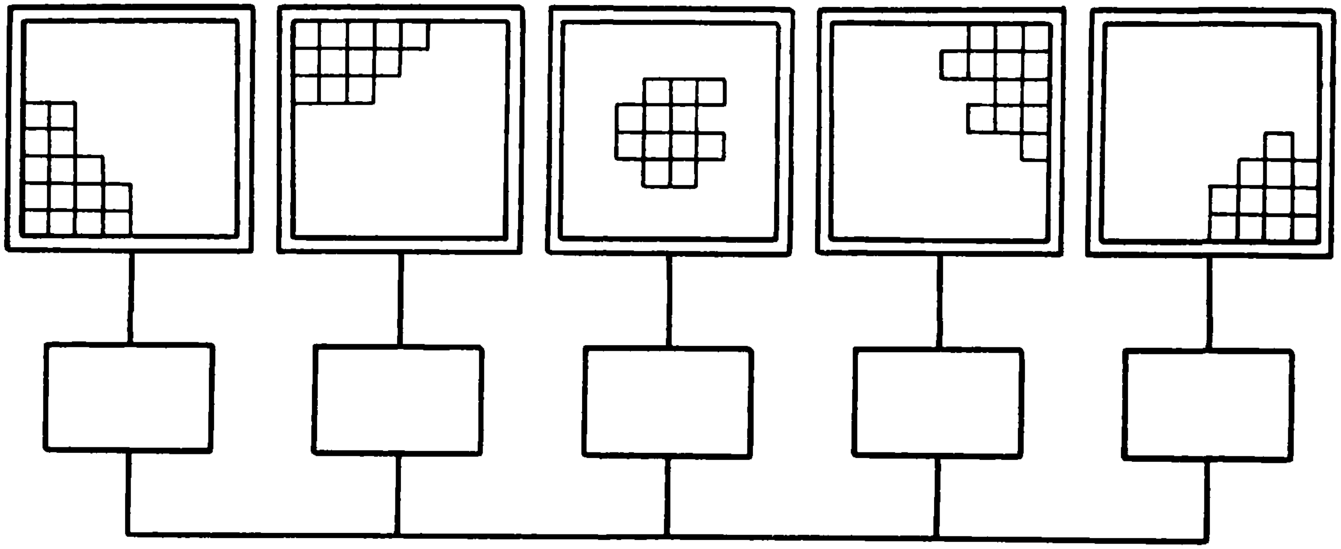


Figure F-6. Distributed / Shared Information Architecture

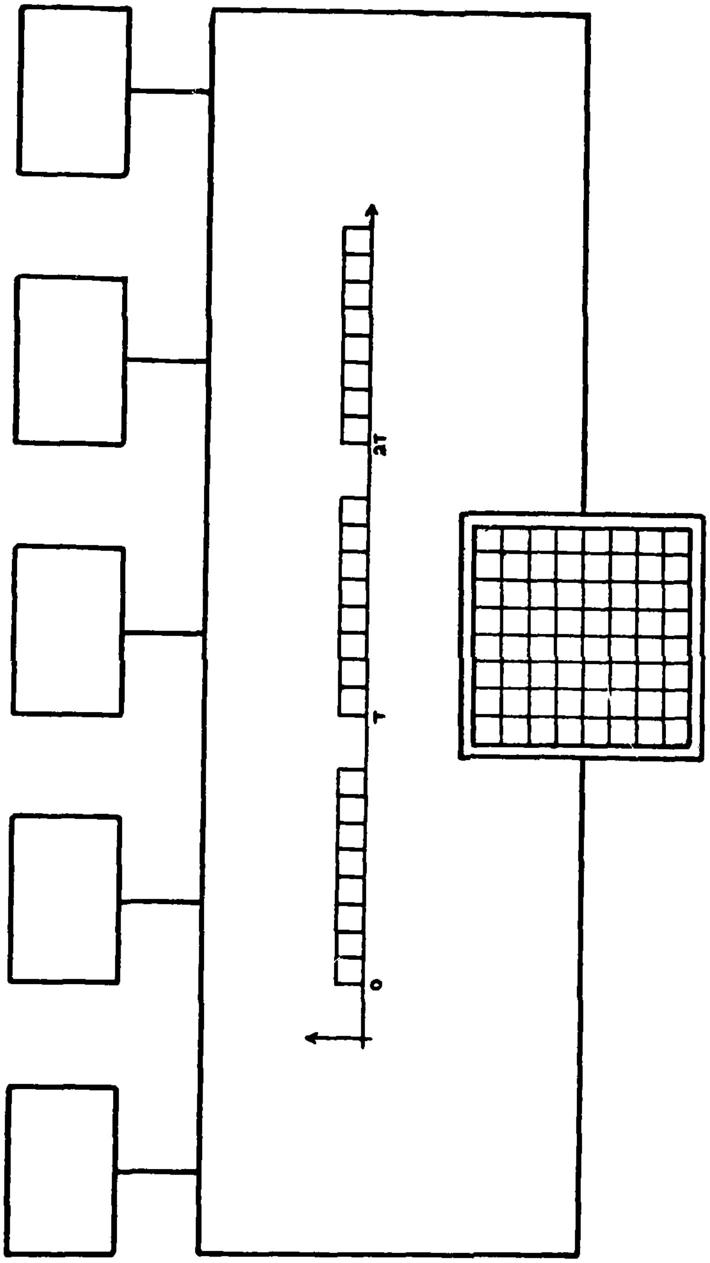


Figure F-7. Broadcast Data Vector Architecture

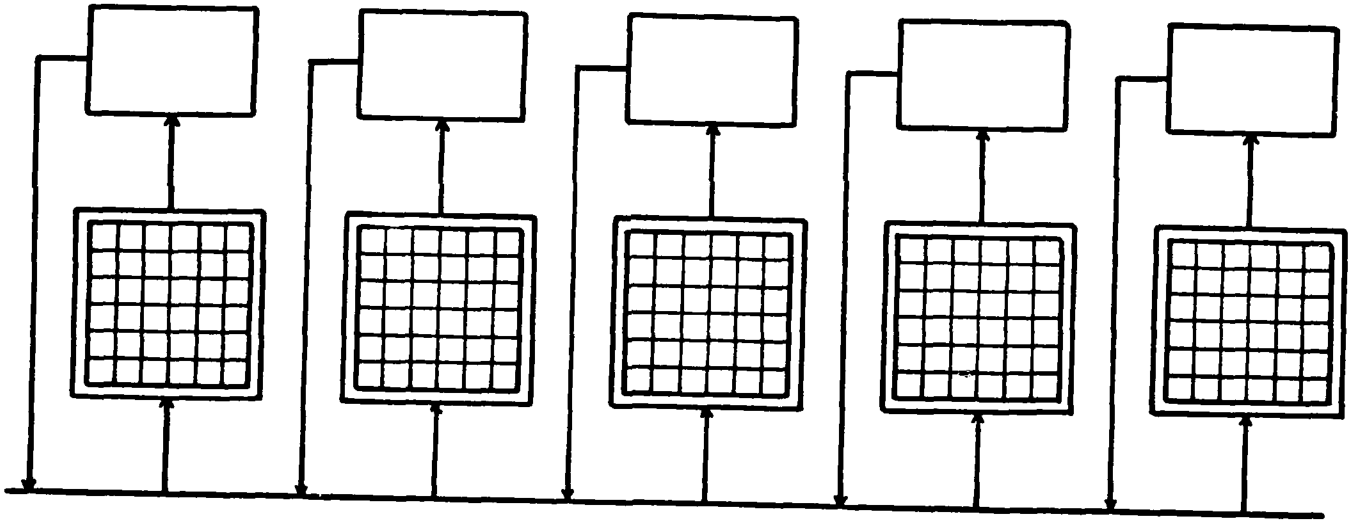


Figure F-8. Virtual Memory Emulation

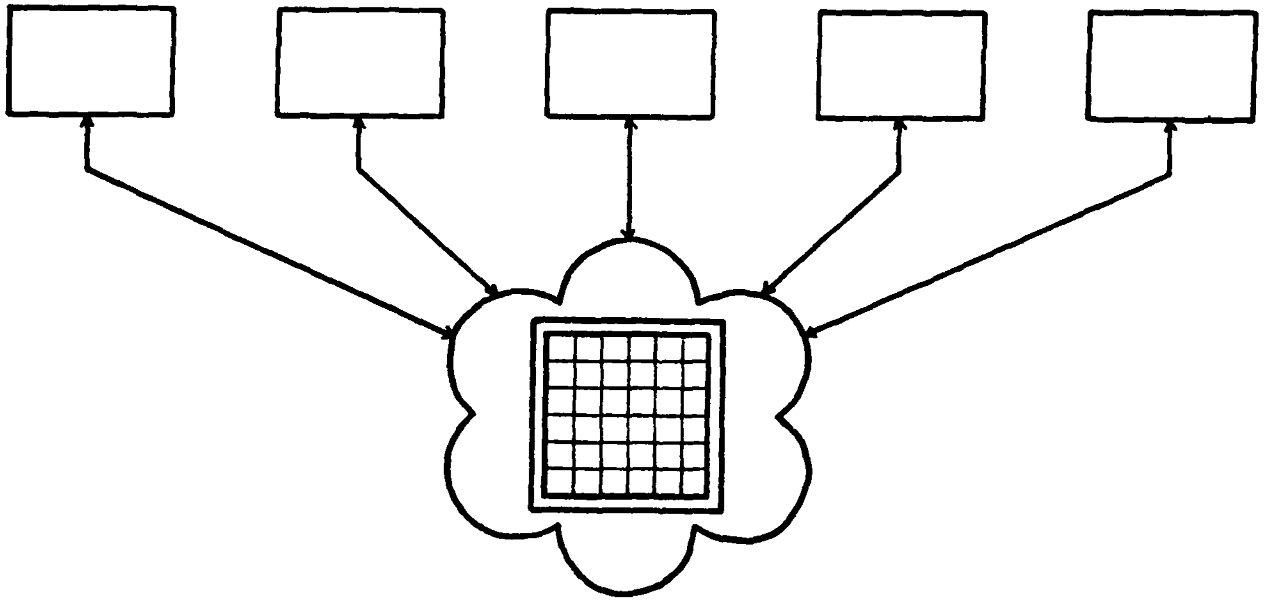


Figure F-9. Virtual Memory Equivalent

APPENDIX G

MULTI-RATE MILLIMODULE DESIGN

In Section IV-B of this report the concept of the task assignment chart (TAC) was introduced and it was shown how tasks could be scheduled to execute at a variety of rates. This appendix provides further elaboration on how this is done.

For the task assignment chart to be generally useful, it should allow millimodules to be executed at any arbitrary repetition rate. Since execution of the TAC repeats after every pass through it, every millimodule in the chart is guaranteed to repeat at the major frame rate of the system. If the major frame is only one minor frame long (10 milliframes), then a given millimodule in the chart will repeat once every 10 milliframes (a rate of 100 Hz in the CRMmFCS architecture).

Figure 17 showed how faster rates could be achieved by placing the same millimodule (kl) in the chart more than once a periodic intervals. Unfortunately, there is only a limited number of rates for which this will work. Figure G-1 illustrates the problem.

In the figure, millimodule timing over a period of 30 milliseconds is shown for the case where a major frame is only 10 milliframes long. Every 10 milliseconds the entire process repeats as the system makes three passes through the TAC. Ten millimodules labeled A through J are shown repeating at intervals ranging from every millisecond for module A to every 10 milliseconds for module J. Inspection of the chart will also show that only modules A, B, E, and J

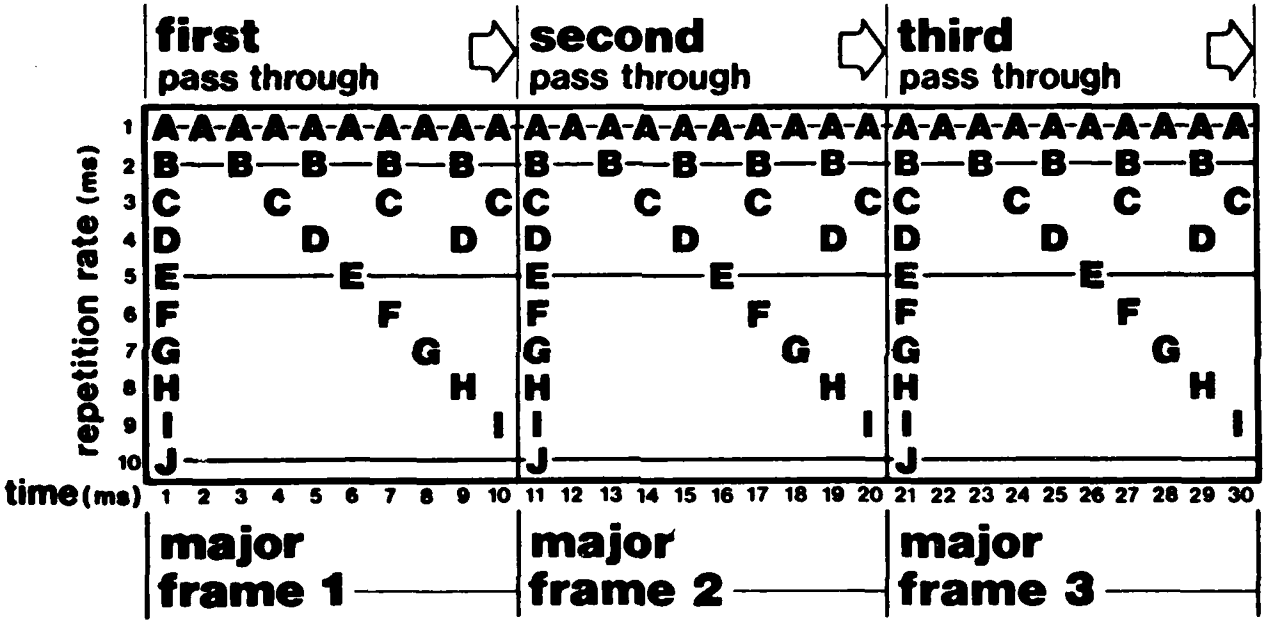


Figure G-1. Possible Rates for a Ten-Milliframe TAC execute at uniform rates over the entire 30 millisecond period. This is because only the periods of 1, 2, 5, and 10 milliseconds divide into the major frame period of 10 milliseconds evenly. All other rates do not divide evenly and therefore lose synchronization at every major frame boundry.

For example, module C correctly executes in milliframes 1, 4, 7, and 10. However, when the second pass through the chart begins at milliframe 11, module C is forced to execute immediately (because it is always located in the first column of the chart) and fails to execute at its correct time during milliframe 13. A similar problem exists for modules D, F, G, H, and I. Thus, in a 10 milliframe TAC, only rates of 1, 2, 5, and 10 can be scheduled (without resorting to the compound millimodule approach discussed later).

If the major frame length is increased to 20 milliframes, a larger number of rates are made possible. These rates include 1, 2, 4, 5, 10, and 20 milliseconds all of which divide into the major frame rate evenly. Similarly, for a 30 milliframe TAC rates of 1, 2, 3, 5, 6, 10, 15, and 30 are allowed. A 40 milliframe TAC allows 1, 2, 4, 5, 8, 10, 20, and 40 millisecond iteration rates, and so on. A 30 milliframe TAC was chosen for the CRMmFCS architecture because it provided the greatest number of rates while minimizing the size of the required task assignment tables. This 30 millisecond chart was then divided into three 10 millisecond minor frames for reconfiguration purposes. Processors volunteer for one minor frame at a time and complete a single pass through the TAC every three minor frames.

Compound Millimodules

Up to this point it has been shown how repetition intervals of 1, 2, 3, 5, 10, 15, and 30 can be obtained by simple placement of millimodules in a 30 milliframe TAC. This section briefly discusses how absolutely any repetition rate can be achieved using the technique of "compound millimodules."

A compound millimodule is simply a module which does not execute every time it is called. A modulo-2 millimodule, for example, executes only every other time it is called. Modulo-3 modules execute every third call, and so on. By placing a compound millimodule in the chart at some legal repetition interval and adjusting its modulus appropriately, absolutely any repetition rate can be achieved.

For example, if a repetition rate of six milliseconds is desired for a particular millimodule, it may be placed in the TAC every 3 milliseconds and given a modulus of two. It will then execute every second time it is called at the desired six millisecond rate. Alternatively, the module could have been scheduled every 2 milliseconds with a modulus of 3 or a modulo-6 module could be scheduled every millisecond. In general, it is desirable to minimize the modulus of every millimodule because each time a module is called but not executed an entire millisecond is wasted in the chart. A modulo-19 module wastes 18 milliseconds for every 1 millisecond of productive execution time.

Constructing Compound Millimodules

Once it has been decided to create a compound millimodule in order to achieve some special repetition rate, some method is needed to tell the module that it is compound. This may be accomplished through the use of a "modulus count variable" (MCV). There is a separate MCV associated with every compound millimodule. This variable tells the module how many times it has been called since the last time it executed its low-rate task. It is located in the SIM where it can be accessed by whichever processor is responsible for executing the module during a given minor frame.

When a compound millimodule is executed, it first checks to see if its MCV has reached its maximum count (the modulus of that module). If it has reached maximum, the module resets it to zero (by broadcasting on the bus) and then executes its assigned low-rate task. If it has not, then the module increments it one count (by broadcasting on the bus) and marks time without doing anything until the millisecond is over. The net result is that the assigned task is executed once every M calls where M is the modulus of the millimodule.

Another variation to the compound millimodule design attempts to make use of the M-1 milliseconds in each low-rate iteration during which the module only marks time. This approach interlaces M diffferent low-rate tasks so that one of them (depending upon the MCV) gets executed every time the module is called. While this method may be used in special circumstances, it is not recommended because it makes the task assignment chart less readable and introduces complexities that are contrary to the goal of very simple software.

REFERENCES

1. S. J. Larimer and S. L. Maher, "A Solution to Bus Contention in a System of Autonomous Microprocessors," Proceedings of the IEEE 1981 National Aerospace and Electronics Conference, May 1981, pp. 309-317.

2. S. L. Maher and S. J. Larimer, "Continuous Reconfiguration in a Multi-Microprocessor Flight Control System," AGARD Avionics Panel on Tactical Airborne Distributed Computing and Networks, June 1981.

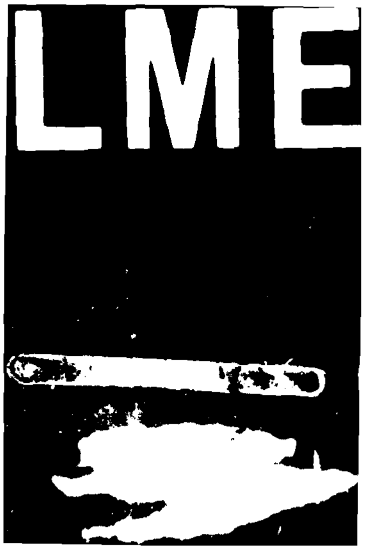
3. W. A. Crossgrove and L. A. Smith, "Distributed Systems: The Next Integration Method," AIAA 2nd Digital Avionics Systems Conference, November 1977.

4. Integrated Computer Systems, Inc., "Distributed Processing and Computer Networks," 1978 Course Notes.

5. J. A. White, et al, A Multi-Microprocessor Flight Control System, Honeywell Interim Report, August 1980.

6. S. J. Larimer, "Managing Software in a Continuously Reconfiguring Multi-Microprocessor System," Proceedings of the 1981 Joint Automatic Control Conference, June 1981.

U.S.Government Printing Office: 1981 — 757-002/574


LME