IEEE Transactions on Nuclear Science, Vol. 36, No. 1, February 1989

MULTIPROCESSOR SHARED–MEMORY INFORMATION EXCHANGE

L. L. Santoline, M. D. Bowers, and A. W. Crew

Engineering Technology Division
Westinghouse Electric Corporation
Research & Development Center
1310 Beulah Road
Pittsburgh, PA 15235
(412) 256–2537/2456/2539

C. J. Roslund and W. D. Ghrist III

Nuclear and Advanced Technology Division
Instrumentation Technology and Training Center
Westinghouse Electric Corporation
P.O. Box 598
Pittsburgh, PA 15230
(412) 733–6780/6343

Abstract

In distributed microprocessor–based instrumentation and control systems, the inter– and intra–subsystem communication requirements ultimately form the basis for the overall system architecture. This paper describes a software protocol which addresses the intra–subsystem communications problem. Specifically, the protocol allows for multiple processors to exchange information via a shared–memory interface. Our primary goal is to provide a reliable means for information to be exchanged between central application processor boards (masters) and dedicated function processor boards (slaves) in a single computer chassis. The resultant Multiprocessor Shared–Memory Information Exchange (MSMIE) protocol, a standard master–slave shared–memory interface suitable for use in nuclear safety systems, is designed to pass unidirectional buffers of information between the processors while providing a minimum, deterministic cycle time for this data exchange. This is achieved by providing multiple buffers for each unique block of information passed between the two processors. Another important feature of the design is that the interface between masters and slaves is identical for different types of slave processors. Thus, the amount of custom software in the final system is minimized. The use of standard system software not only eases initial software verification and validation requirements, it also simplifies long term system software maintenance.

Introduction

This paper describes the design of a standard shared–memory interface for intra–subsystem communications. The interface provides a method for reliable information exchange between processors in a single computer chassis which have access to a subsystem bus and shared–memory resources. This interface protocol, known as Multiprocessor Shared–Memory Information Exchange (MSMIE), is optimized for real–time critical process control and instrumentation systems such as nuclear safety systems.

A distributed processing architecture is a natural choice when designing critical real–time systems such as nuclear safety systems. By distributing the processing requirements of a time–critical function across multiple processors, the tasks to be performed by each component processor are reduced. Furthermore, the processing tasks of most subsystems within a system can be functionally viewed as a unique application function and a set of common "operating system" type functions such as I/O handling and pre–processing, external communication processing, and diagnostics, to name a few. By off–loading the dedicated "operating system" type functions onto individual "slave" processors, two distinct advantages are gained. First, the hardware and software for the processors performing the common system tasks may be of a standard, configurable design. Secondly, the processing burden of the subsystem application processor, or "master" processor, is substantially reduced, both in volume and execution time. However, the use of multiple processors to implement a single subsystem creates an additional communications burden: that of communications among the processors within the subsystem. The most efficient mechanism for intra–subsystem communication is a tightly–coupled architecture in which all processors share a bus, and communicate via a bus–accessible shared memory.

A typical architecture for a functionally–distributed computer system is shown in Figure 1. The system shown contains two subsystems, each containing a "master" processor, a "slave" processor, and a shared memory. Intra–subsystem communications are accomplished via the shared memory, and inter–subsystem communications are accomplished between the two slave processors via an unspecified physical communicating channel.

In order to avoid designing unique interfaces between master processors and each different type of slave processor, a standardized shared–memory interface protocol is required. The MSMIE protocol defines such an interface. It is optimized for real–time, process control type systems, such as nuclear safety systems, where operation in a non–interrupt driven environment is highly desirable.

The MSMIE Interface

The Participants

One of the primary goals of the MSMIE design is to identify a set of standard, configurable slave processor boards, and to design the slave processor software so that each slave microprocessor board of a given type could be used interchangeably throughout the overall system. This type of design not only minimizes the number of different microprocessor board types and the amount of custom software in the overall system, it also restricts custom software to the master processor boards. In a nuclear safety system, this type of design significantly improves overall system quality and integrity by focusing the total design effort (including verification and validation), on a small number of hardware and software components which are used as "building blocks" throughout the system. An additional benefit of such a standardized design is that long term hardware and software maintenance is simplified. For all of these reasons, the MSMIE interface has been designed so that master processors have the ability to configure individual slaves and thereby tailor them to the specific requirements of the subsystem. Thus, slave processors are dependent upon the master processors for their configuration information, which is passed to the slave processors during initialization. Slave processors communicate with the master processors via the subsystem shared memory, usually resident on each slave processor board. To further isolate the functionality of the slaves, they are only permitted to communicate with a master processor via the shared-memory interface. Slave-to-slave communications within a subsystem are not permitted except through some external communications device or via the subsystem master.

For some critical subsystems, an added degree of fault tolerance implemented via redundant subsystem master processors may be necessary. For true fault tolerance, the master processors must be fully redundant and isolated so that a fault of one master processor will not cause the others to fail. To allow the MSMIE interface to function properly, only one master processor may have the power to control the shared-memory interface at any given time. This processor is denoted the "primary" master, while all other masters are called "auxiliary" masters. The primary master is responsible for initial communications establishment with the slave processors, configuration of the slave processors, and if warranted, resetting the slave processors. The auxiliary masters may only monitor the shared-memory interface until after the slaves have been configured and MSMIE communications are fully established. At that point, the auxiliary masters may participate in shared-memory message passing to and from the slave processor boards.

Shared-Memory Organization

Master and slave processor communications is implemented via a predefined set of shared-memory data structures, which form the basis of the MSMIE interface. Between each slave processor and the subsystem host processors, a shared-memory region exists which is organized as shared-memory configuration data structures followed by message buffers. To maintain configurability of the slave processors, the shared-memory data structures are passed from the master to the slave processors during MSMIE initialization. The data structures contain information used by the slave to define the number and operation of any physical communications channels on the slave, the number, directionality, and definition of the messages communicated over each physical channel and between the masters and the slave, and other general configuration information. In addition, the data structures contain locations for passing diagnostic and run-time status between the masters and the slave, and locations for controlling the establishment of communications and resetting the slave from the primary master.

Method

The MSMIE protocol is designed so that each message buffer exchanged between the slave and masters is unidirectional, with the contents of the message being a continuously-updated image. The method for one processor to communicate with another is:

- The processor sending information will continually copy the newest image of a message into a shared-memory buffer.
- The processor receiving information will read from the shared-memory buffer containing the newest image.

A shared-memory buffer is either updated by a master processor, and the flow of information is from master-to-slave; or alternatively, a shared-memory buffer is updated by a slave processor, and the flow of information is from slave-to-master.

The use of shared memory as a means of exchanging information between multiple, asynchronous processors is only successful if a mechanism exists to prevent simultaneous access to a given memory resource. Without this mechanism, the possibility of "data tearing" arises. Data tearing occurs when one processor writes to a memory area while it is being read by another processor. If this situation exists, it is possible that the processor reading the memory area actually reads portions of both old and new data. This can happen whenever the memory location being accessed is of a size that requires multiple machine instructions to read or write the location. Consider the following simple example, where processor number one reads a location which is concurrently being written to by processor number two:

WORD VALUE			
High Byte	Low Byte	Processor One	Processor Two
55	55	READ Low = 55	
55	33		WRITE Low = 33
33	33		WRITE High = 33
33	33	READ High = 33	

As indicated, the word value read by processor number one is invalid as it contains the low byte of the "old" data value, but the high byte of the "new" data value. This simple example can be extended to more sophisticated situations, where the integrity of whole blocks of data must be maintained.

In order to prevent data tearing, mutually-exclusive access to the shared-memory area must be guaranteed. In MSMIE, this is partially accomplished with software semaphores. The semaphores allow a processor, while it has access to a particular shared-memory area, to prevent, or "lock out", other processors from accessing that same shared-memory area.

While semaphores prevent data tearing, they introduce the possibility of one processor being denied access to a shared-memory area because that area is locked by another processor. The processor which desires buffer access must wait for the other processor to release the locked shared memory area. This is readily apparent if only a single shared–memory buffer is allocated to hold each message image. In this case, the buffer acquisition/release scheme of master and slave processors is as shown in Figure 2. As illustrated, buffer contention problems are normal with a single buffer per message allocation scheme. When messages are passed from "slave–to–master", the master processor must wait for the slave to update the data in the buffer, and then release the buffer so the master can access the newest data. While the master is reading the new data from the buffer, the slave has no free area to build a new message. For message images passed from "master–to–slave," the situation is reversed. In either case, each processor must wait for the buffer to be in the correct state (either "idle" or "newest") before it may access the buffer. There is no guarantee that the buffer will be in the correct state at any given time. If this primitive message passing interface is used for information exchange between the master and slave, only a fraction of the full power of a multiple processor architecture can be realized, since each processor wastes a portion of its execution cycle waiting for the other processor to release the shared–memory resource.

The addition of a second buffer for each message image communicated between the master and slave processors solves some buffer contention problems. While one or more processors are reading a message image from the first shared–memory buffer, another processor may be building a message update in the second shared–memory buffer. Using this method, the participating processors can simultaneously operate on (read from or write into) shared–memory buffers without interfering with one another.

However, because master and slave processors may run asynchronously, buffer contention is still possible even with dual memory buffers allocated for each message image. This is true because there are no real restrictions on the amount of time a processor can hold a buffer assigned to itself. The buffer contention problem which occurs using dual shared–memory buffers for each message image is explained in the following scenario (see Figure 3): Assume the slave processor is operating with a slower cycle time than the master processor. This implies that the slave will take longer to access and release a data buffer than the master. When the slave processor is reading an image of a message from one shared–memory buffer, the faster master processor builds a new image of the message and places it in the second shared–memory buffer. If the master processor builds another new message image before the slower slave processor has finished using its data buffer, then the master processor must either wait for the slave to release its buffer to the idle state, or overwrite the previous "newest" image with the fresher data. The first alternative is undesirable because the operation of the two processors is coupled. The second alternative is also undesirable because the newest image is destroyed. In the second case, when the slave processor finally releases its buffer, it can no longer immediately access a new image. It must instead wait for the master processor to finish updating the newest image and release the buffer to the newest state.

To eliminate the possibility of buffer contention, three shared–memory buffers can be allocated for each message image. With this scheme, at any given time, one of the buffers can hold the newest complete image, a second buffer can be assigned to a processor for reading an image, while a third buffer can be assigned to a processor for building another new image. A buffer is guaranteed to be available to each processor at the time it requests access to a buffer, and a third buffer is always available to prevent the newest complete image from being destroyed.

Thus, for each data image which must be communicated between a slave and the masters' processors, the MSMIE protocol maintains a set of triplicated buffers. Information about the buffers and access control to the buffers is provided by "buffer descriptor tables" in shared memory. Each set of triplicated buffers has an associated buffer descriptor containing the following entries:

- A three–element Buffer_Status array which holds the status of each of the three shared–memory buffers. Each buffer can have one of five buffer statuses: "idle", "assigned to master", "assigned to slave", "newest", and "not used".
- A semaphore location which provides exclusive access to the Buffer_Status array for both master and slave processors. The semaphore and the Buffer_Status array are used to control access to the triplicated shared–memory buffers.
- A Number_of_Readers location which maintains a count of the number of master processors currently reading the buffer whose status is "assigned to master".
- An Access_Mode location which determines the number of read accesses permitted to "newest" data buffers.

The method in which the buffer descriptor parameters are used varies according to the direction of information flow and whether the processor desiring buffer access is a master or slave processor. The use of the buffer descriptor parameters will be described in the following sections.

<u>Slave–to–Master Message Passing, Slave Processor Buffer Acquisition/Release:</u> In slave–to–master message passing, the slave processor marks messages for use by the master processor. Typically, the slave processor has a buffer assigned to it at initialization to hold the first message. Then, when the acquire/release buffer procedure is invoked, the slave releases the buffer which was assigned to it (by updating its status to "newest"), then acquires an "idle" buffer to hold the next update of the message. The procedure which the slave processor follows to release its current buffer and acquire an "idle" buffer is described below:

1. The slave processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore. Once the slave processor has the semaphore in the locked state, the master processor is denied access to the Buffer_Status array.
2. The Buffer_Status array is searched for a status of "newest". If a "newest" status is found, then the data which the slave processor is updating for the master replaces this buffer, so the Buffer_Status of "newest" is changed to "idle".
3. The Buffer_Status array is searched for a status of "assigned to slave". If the status of "assigned to slave" is not found, then an error has occurred.
4. The buffer whose status is "assigned to slave" is changed to "newest". This completes the release of the "assigned to slave" buffer. An "idle" buffer must now be acquired.
5. The Buffer_Status array is searched for a status of "idle". This buffer will be used to hold the next message update. If an "idle" buffer is not found, then an error has occurred.
6. The slave processor acquires the "idle" buffer by

changing its status from "idle" to "assigned to slave".

7. The buffer descriptor semaphore is released.

Slave–to–Master Message Passing, Master Processor Buffer Acquisition/Release: The master processor accesses the newest messages provided by the slave processor. Two separate procedures are provided: one to acquire the "newest" buffer, and a second procedure to release the "assigned to master" buffer once the data has been used. The actions which must be taken by the master processor in order to acquire "newest" data buffers are described below:

1. The master processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore. Once the master processor has the semaphore in the locked state, the slave processor is denied access to the Buffer_Status array.
2. The Buffer_Status array is searched for a status of "assigned to master".
3. If an "assigned to master" buffer is found, then another master processor has acquired this buffer. (In this case, multiple master processors have access to the slave's shared memory.) The master processor presently desiring buffer access is constrained to read the data in the current "assigned to master" buffer.
4. If an "assigned to master" buffer is not found, then this master processor is free to search the Buffer_Status array for a "newest" buffer. If a "newest" buffer is found, then it is acquired by changing the Buffer_Status to "assigned to master". If a "newest" buffer is not found, then no message has been provided by the slave processor.
5. To mark the number of master processors reading this buffer, the Number_of_Readers location is incremented.
6. The buffer descriptor semaphore is released.

At this point, the master processor uses the data from the "assigned to master" buffer. When the master no longer desires access to this buffer, it may release the "assigned to master" buffer such that the slave processor can reuse this buffer. The procedure which the master follows to release the buffer is described below:

1. The master processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore. Once the master processor has the semaphore in the locked state, the slave processor is denied access to the Buffer_Status array.
2. The Number_Of_Readers location in the buffer descriptor is decremented, as this master processor no longer requires access to the "assigned to master" buffer.
3. If the Number_Of_Readers location is not equal to zero, then another processor has access to the "assigned to master" buffer. If this is the case, the "assigned to master" buffer cannot be released. The Buffer_Status is left in the "assigned to master" state, and the buffer descriptor semaphore is released.
4. If the Number_Of_Readers location is now equal to zero, then this master processor was the only master processor using the "assigned to master" buffer, and the buffer may be released. In order to release the buffer, the Buffer_Status array is searched for a status of "newest". If such a buffer is found, or the Access_Mode location in the buffer descriptor indicates that each buffer is to be accessed only once, then the status of the buffer to be released is changed to "idle". Otherwise, the buffer which is to be released still contains the newest data, and it should be released by changing its status back to "newest".*
5. The buffer descriptor semaphore is then released.

Master–to–Slave Message Passing, Master Processor Buffer Acquisition/Release: In master–to–slave message passing, the master processor marks messages for use by the slave processor. The master processor uses two separate procedures to access the shared–memory data buffers; one to acquire an "idle" buffer, and a second procedure to release the "assigned to master" buffer once the new message has been moved into the shared–memory buffer. The actions which must be taken by the master processor to acquire an "idle" buffer are described below:

1. The master processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore.
2. The Buffer_Status array is searched for a status of "idle". If an "idle" buffer is not found, then an error has occurred.
3. The buffer whose status is "idle" is acquired by the master processor by changing the status to "assigned to master".
4. The buffer descriptor semaphore is released.

At this point, the master processor moves the "newest" message image into the acquired shared–memory data buffer. Once this data transfer is complete, the master processor may release the "assigned to master" buffer such that the slave processor can use the "newest" data. The procedure which the master follows to release the buffer is described below:

1. The master processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore.
2. The Buffer_Status array is searched for a status of "newest". If a "newest" status is found, then the data which is being provided by the master processor replaces this buffer, so the Buffer_Status of "newest" is changed to "idle".
3. The Buffer_Status array is searched for a status of "assigned to master". This is the buffer which has been filled with the "newest" data. If such a buffer is not found, an error has occurred.
4. The buffer whose status is "assigned to master" is changed to "newest".
5. The buffer descriptor semaphore is released.

Master–to–Slave Message Passing, Slave Processor Buffer Acquisition/Release: The slave processor must access the newest message provided by the master processor. Two separate procedures are provided: one to acquire the "newest" buffer, and a second procedure to release the "assigned to

*If multiple master processors are used, the Access_Mode must be configured to allow an unlimited number of accesses to a single "newest" buffer. This is so all master processors are guaranteed access to a "newest" message once one has been provided by the slave.

slave" buffer once the newest data has been used by the slave processor. The actions which must be taken by the slave processor in order to access "newest" data buffers are described below:

1. The slave processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore.
2. The Buffer_Status array is searched for a status of "newest".
3. If a "newest" buffer is found, it should be acquired for use by the slave processor by changing its status to "assigned to slave". If a "newest" buffer is not found, then the master processor has not yet provided any messages in shared memory.
4. The buffer descriptor semaphore is released.

At this point, the slave processor is free to use the data in the buffer assigned to it. When the slave processor no longer requires access to the "assigned to slave" buffer, it must be released so that the master processor can reuse the buffer. The procedure which the slave follows to release the buffer is described below:

1. The slave processor locks the Buffer_Status array by acquiring the buffer descriptor semaphore.
2. The Buffer_Status array is searched for a status of "newest". If such a buffer is found, or if the Access_Mode location in the buffer descriptor indicates that each buffer is to be accessed only once, then the status of the buffer to be released is changed to "idle". Otherwise, the buffer which is to be released still contains the "newest" data, and it should be released by changing its status from "assigned to slave" back to "newest".
3. The buffer descriptor semaphore is then released.

<u>Multiple Channel Slaves:</u> The basic method of applying triple buffering to shared–memory communications of data images has been described. This method can be extended to suit the particular needs of different types of slave processor boards. As previously mentioned, the slave processors are typically designed to offload the master processors from performing standard system tasks. One common slave processor function is simplex point–to–point (datalink) communications. This type of slave processor benefits from a variation of simple shared–memory triple buffering due to multiple communication channel considerations.

A datalink controller type of slave processor generally has greater than one physical communications device. Each of the physical communications channels (datalinks) can operate as a transmitter, receiver, or bidirectional channel. It is also possible that multiple message images are to be communicated over a single physical channel.

Because datalink activity is serial, only one message at a time can be transmitted or received on any given channel. Thus, triple buffering is implemented as follows: The number of shared–memory buffers required for each datalink channel is equal to two times the number of unique messages communicated over that channel plus one additional buffer. The "extra" buffer is for the physical channel itself, i.e., the buffer into which messages are received or from which messages are transmitted. In this arrangement, the shared–memory buffers are in a free pool of buffer space, and are not associated with a particular buffer descriptor except at initialization. At initialization, two shared–memory buffers are assigned to each buffer descriptor, and are initialized to the "idle" state. The third Buffer_Status in each buffer descriptor is initialized to the "assigned to slave" state, as the third buffer for all buffer descriptors associated with a single channel corresponds to the single "extra" buffer which is assigned to the physical channel. In this case, the "assigned to slave" Buffer_Status can be thought of as "assigned to physical channel". Because the buffers are not rigidly allocated to a particular buffer descriptor, the size of each of the allocated buffers must be at least as large as the largest message received or transmitted over the given channel. At any given time, two buffers are associated with each particular message image, and the third buffer is always assigned to the datalink controller physical communications device. When triple buffering is implemented in this manner, the method of acquiring and releasing buffers from the master side is identical to that previously described. From the datalink controller side, buffer acquisition and release is a "swapping" process.

On datalink controller receive channels, messages are received over the datalink and must be marked for use by the master processor. The messages are received into the shared–memory buffer assigned to the physical channel. Once a new message has been received, the datalink controller must determine which buffer descriptor the message is associated with, and find the correct buffer descriptor. This buffer descriptor is where the "assigned to slave" buffer must be returned, and from where an "idle" buffer must be acquired to rearm the physical channel. Once the correct buffer descriptor is found, the procedure which the datalink controller follows to release its current buffer and acquire an "idle" buffer is identical to standard triple buffering for the slave–to–master message passing case described previously.

For transmitter channels, the master processor provides "newest" message buffers which contain data to be transmitted over the physical datalinks by the datalink controller slave. If multiple messages are transmitted on a single datalink, each message is transmitted separately and in order. In this case, buffer acquisition and release is a two step buffer swapping process, as described below:

1. Acquiring buffers for transmission involves swapping a current "assigned to slave" buffer with the "newest" buffer from the buffer descriptor containing the least recently–transmitted message. "Newest" buffer acquisition is described below:
   a. The semaphore of the buffer descriptor corresponding to the least recently–transmitted message is acquired.
   b. The Buffer_Status array is searched for a status of "newest". If a "newest" status is found, then the datalink controller must swap the current "assigned to slave" buffer with the "newest" buffer so that the datalink controller can transmit the newest data.
   c. The Buffer_Status of the "assigned to slave" buffer is changed to "idle".
   d. The Buffer_Status of the "newest" buffer is changed to "assigned to slave".
   e. The buffer descriptor semaphore is released.
2. The transmission of the "assigned to slave" buffer is initiated. Upon the completion of transmission, the buffer must be returned so that the master processor can reuse the buffer. The buffer <u>must</u> be returned to the same buffer descriptor from which

it was acquired. The procedure for returning "assigned to slave" buffers is described below:

a. The buffer descriptor semaphore of the most recently-transmitted buffer is acquired.

b. The Buffer_Status array is searched for a status of "newest". If such a buffer is found, or if the Access_Mode of this buffer descriptor indicates that each "newest" buffer is to be used only once, then the buffer which is presently "assigned to slave" remains in that state. Otherwise, the buffer which is "assigned to slave" still contains the "newest" data, and it should be released by changing its status from "assigned to slave" back to "newest".

c. If the "assigned to slave" buffer was released (its status changed to "newest"), then an "idle" buffer must be acquired for use by the datalink controller. The Buffer_Status array is searched for a status of "idle". This buffer is acquired by changing its status to "assigned to slave".

d. The buffer descriptor semaphore is released.

The triple buffering procedure followed by slave processors which are similar to datalink controllers is just an extended version of simple shared-memory triple buffering. However, this method significantly reduces the memory requirements when many messages of a similar size must be transmitted or received over a single physical channel. This method reduces to simple triple buffering when only one message is transmitted or received per physical channel.

Multiple Master Processor Considerations: The basic triple buffer acquire/release algorithms which have been described are applicable regardless of whether one or more than one master processor is communicating with the slave processor. When multiple masters are present, the Number_of_Readers location in each buffer descriptor allows each master processor to simultaneously read the same buffer of a message passed from slave-to-master. However, only a single master processor is permitted to supply each image of a message passed from master-to-slave in order to prevent master-to-master buffer contention.

Although no changes are required to the buffer acquire/release algorithms, there are various operating constraints when more than one master processor is present in a subsystem:

1. Because each multiple master processor may run asynchronously with respect to other master processors and the slave processor, one master could essentially prevent the others from ever reading data provided by the slave if the Access_Mode selection on slave-to-master message passing buffers were configured for one time buffer access. This leads to the requirement that all buffer descriptors with direction "slave-to-master" must be configured to allow an unlimited number of accesses to a single "newest" buffer to ensure that all master processors are able to access the "newest" data at least once per master processor cycle.

2. A timing constraint must be placed on the amount of time any master is allowed to access a buffer. This constraint is necessary so that a buffer which is "assigned to master" is guaranteed to be released by all masters at least once per master processing cycle. For a system with M masters, the amount of time any master processor may assign a buffer to itself must be less than $1/M$ of the smallest loop cycle time of any master.

Reliability

The method of using triplicated buffers for message passing between master and slave processors provides for information exchange between the processors within a minimal and deterministic time frame. For use in nuclear safety systems, the information exchange must also be extremely reliable. The MSMIE protocol provides reliability in the data exchange by several mechanisms, which are summarized below:

- A field specifying the length of the message is embedded into a header which is part of every message image.
- A message serial number is embedded into the message header. The serial number is used by the receiving processor to determine if a message has been read before.
- New message buffers are timestamped when they are placed into the shared memory. The receiving processor can calculate the age of a buffer by comparing the timestamp of the buffer with a representation of "current time", maintained in shared memory by subsystem slaves.
- Source-to-destination error detection is provided by a word-wide checksum which is embedded into the message. The checksum is computed by the processor which originates a message; it is recomputed by the end processors which eventually receives the message.

Summary

The MSMIE protocol has several features which make it ideally suited to inter-processor communications in distributed, microprocessor-based nuclear safety systems. At this time, implementation of the MSMIE protocol is a central part of the embedded software of several large Westinghouse nuclear system designs. The MSMIE protocol maximizes overall system performance in a multiprocessor environment while guaranteeing reliable communications between processors, deterministic performance, and maximum software reusability. This protocol represents a significant development in the design of nuclear safety system software.

Acknowledgments

The concepts and procedures described in this paper are taken from internal Westinghouse documents authored by the following persons: Mark D. Bowers, Albert W. Crew, William D. Ghrist III, Gilbert W. Remley, Charles J. Roslund, and Linda L. Santoline.

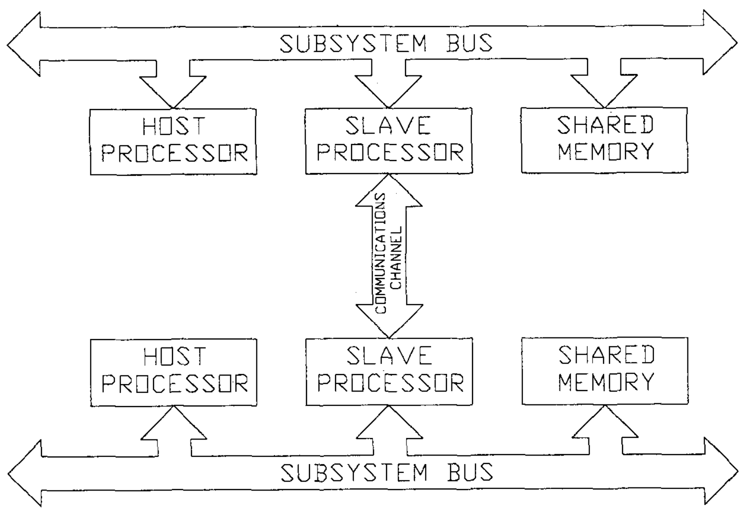


Figure 1: Intra–Subsystem and Inter–Subsystem Communications

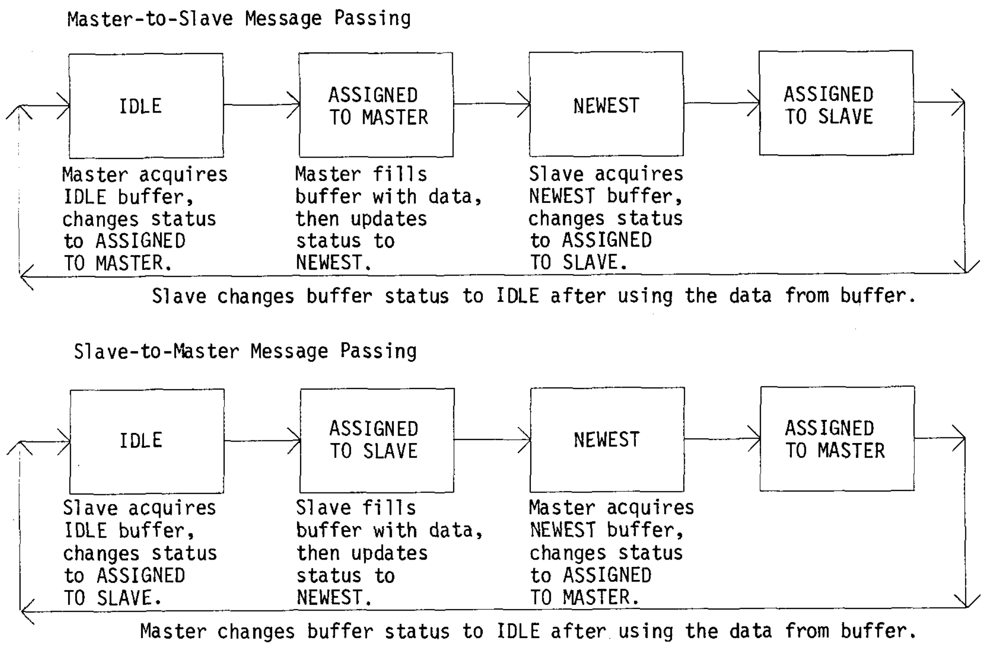
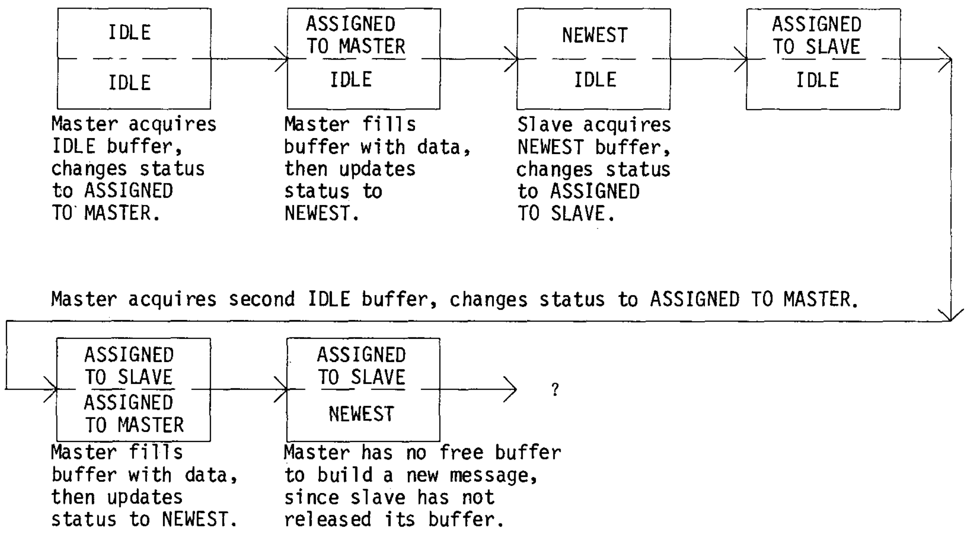

Master-to-Slave Message Passing
IDLE
ASSIGNED TO MASTER
NEWEST
ASSIGNED TO SLAVE
Master acquires IDLE buffer, changes status to ASSIGNED TO MASTER.
Master fills buffer with data, then updates status to NEWEST.
Slave acquires NEWEST buffer, changes status to ASSIGNED TO SLAVE.
Slave changes buffer status to IDLE after using the data from buffer.
Slave-to-Master Message Passing
IDLE
ASSIGNED TO SLAVE
NEWEST
ASSIGNED TO MASTER
Slave acquires IDLE buffer, changes status to ASSIGNED TO SLAVE.
Slave fills buffer with data, then updates status to NEWEST.
Master acquires NEWEST buffer, changes status to ASSIGNED TO MASTER.
Master changes buffer status to IDLE after using the data from buffer.

Figure 2: Master/Slave Buffer Acquisition and Release — Single Buffer Per Message Case

Master-to-Slave Message Passing



Master acquires IDLE buffer, changes status to ASSIGNED TO MASTER.

Master fills buffer with data, then updates status to NEWEST.

Slave acquires NEWEST buffer, changes status to ASSIGNED TO SLAVE.

Master acquires second IDLE buffer, changes status to ASSIGNED TO MASTER.

Master fills buffer with data, then updates status to NEWEST.

Master has no free buffer to build a new message, since slave has not released its buffer.

Figure 3: Master/Slave Buffer Acquisition and Release -- Dual Buffers Per Message Case