NISTIR 4777

A Formal Analysis of the BACnet MS/TP Medium Access Control Protocol

Steven T. Bushby

Building and Fire Research Laboratory
Gaithersburg, Maryland 20899

United States Department of Commerce
Technology Administration
National Institute of Standards and Technology

NISTIR 4777

A Formal Analysis of the BACnet MS/TP Medium Access Control Protocol

Steven T. Bushby
Building Environment Division
Building & Fire Research Laboratory

April 1992

U.S. Department of Commerce
Barbara Hackman Franklin, *Secretary*
Technology Administration
Robert M. White, *Under Secretary for Technology*
National Institute of Standards and Technology
John W. Lyons, *Director*

Prepared for:
U.S. Department of Energy
Dean Devine
Federal Energy Management Program
Washington, DC 20585

ABSTRACT

BACnet, a draft standard communication protocol for building automation and control systems, contains options for physical and data link layer protocols. One option is to use an EIA-485 physical layer combined with a Master-Slave/Token Passing (MS/TP) media access control protocol which was specifically designed for BACnet. This paper presents a formal model of the MS/TP protocol using the technique of communicating machines with shared variables. Using this model, the protocol is analyzed and shown to be deadlock free. It is also shown that if a controller has a message to send it will eventually be transmitted.

1. INTRODUCTION

BACnet is a standard communication protocol for building automation and control networks currently being developed by the American Society of Heating, Refrigerating, and Air-Conditioning Engineers [1-4]. The BACnet draft standard was released for public review and comment in August 1991. BACnet is designed to meet the communication requirements of the entire range of control devices found in a typical, hierarchical, building management and control system.

The protocol offers a choice of three possible networking technologies, providing a range in both cost and performance. One of these choices is based on a twisted-pair bus using EIA-485 signaling [5]. This is the most commonly used physical layer technology in today's building automation systems. It was included in the standard in order to provide a means for backward compatibility with existing systems (perhaps through gateways) and because, for the foreseeable future, there will be building automation devices whose cost and communication requirements cannot justify the expense of the other local area networking options provided for in the draft standard.

It was necessary to develop a medium access control (MAC) protocol to reside above the EIA-485 physical layer in the protocol stack. Many devices used in current building automation systems use a Master-Slave approach to regulating access to the network medium. Others use some sort of peer-to-peer arbitration approach. A hybrid of these two approaches is used in BACnet. The resulting MAC protocol is called Master-Slave/Token-Passing, or MS/TP for short [1].

The basic concept of the MS/TP protocol is that the network will have one or more master nodes and zero or more slave nodes. The master nodes participate in a logical token ring. When a master has the token it may send a request to a slave node. The slave node must respond within a fixed time window. Slave nodes have no way to initiate communication and can only respond to valid requests from a master.

EIA-485 is an asynchronous serial communication standard. In the MS/TP protocol data are transmitted using "bytes" which consist of 8 data bits preceeded by 1 start bit and terminated by 1 stop bit, for a total of 10 bits. The length of time needed to transmit these 10 bits is refered to as 1 **Unit Time**. The data portion of each byte is refered to in BACnet as an octet.

This paper describes the MS/TP protocol using a formal model based on communicating machines with shared variables (CMSV). This formal model is then used to analyze the protocol for safety and liveness properties. Section 2 briefly describes the features of the CMSV model. Section 3 is the formal description of the MS/TP protocol. Sections 4, and 5 present an analysis of deadlocks (safety) and liveness of the protocol for Master-Slave and Token-Passing operation respectively. Section 6 discusses timing considerations for the protocol.

2. THE CMSV MODEL

In the CMSV model developed by Miller and Lundy [6], network entities are represented by a finite state machine and a set of local variables. Each transition in the machine has associated with it an **enabling predicate**, and an **action** which may alter the variable values. If the enabling predicate is TRUE the transition may (but need not) fire. When a transition fires, the associated action is carried out and the machine changes state in one atomic step. If more than one transition in a single machine is enabled, a single transition is selected for firing in a non-deterministic manner. If transitions are enabled in more than one machine they may fire simultaneously. Machines communicate with each other through **shared variables** (shared variables will be denoted in this paper by using italics). If two or more machines attempt to write to a shared variable at the same time the value of the variable becomes **undefined**. In this paper the model will be restricted to prevent simultaneous firing of transitions whose actions include writing to the same variables.

More formally, a system of communicating machines is an ordered pair C = (M, V), where:

$M = \{m_1, m_2, \ldots, m_n\}$ is a finite set of machines, and
$V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of shared variables.

For each machine m_i, the set of shared variables is divided into two subsets R_i, and W_i which indicate the **read access** variables and **write access variables** respectively. The subsets R_i and W_i need not be mutually exclusive.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s, L_i, N_i, \tau_i)$ where

(1) S_i is a finite set of states;

(2) $s \in S_i$ is the initial state of m_i;

(3) L_i is a finite set of local variables;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p,a) where "p" is a predicate on the variables of $L_i \cup R_i$, and "a" is an action on the variables of $L_i \cup W_i \cup R_i$. An action is a partial function

$$(a: L_i \times R_i \dashrightarrow L_i \times W_i)$$

mapping values from the local variables and read access variables to the values of the local variables and write access variables .

(5) $\tau_i: S_i \times N_i \dashrightarrow S_i$, is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

The machines model the entities which make up a protocol system, namely the communicating processes and the channels that connect them. A transition, $\tau(s_1, n) = s_2$, is considered to be enabled whenever the machine m_i is in state s_1 and predicate p, associated with the name n is TRUE. Transition, τ, may be executed at any time if it is enabled. If more than one transition is enabled at the same time, one of the enabled transitions is selected non-deterministically for execution. Every transition is executed atomically, that is, both the state change and the action associated with the transition occur simultaneously. Enabled transitions in different machines may occur simultaneously provided that they do not write to the same shared variable.

There is an assumption of **fairness**, which is, if a transition is enabled indefinitely, then it will eventually fire. Also, if a transition is enabled infinitely often it will fire.

3. APPLYING THE CMSV MODEL TO THE MS/TP PROTOCOL

The MS/TP protocol is specified in the draft standard by a state machine representation of the local information available to a network node and the internal logic needed to implement the protocol. State machines are defined for both master nodes and slave nodes. State transitions are enabled based on locally defined predicates. There is only one transition enabled at a given time in a single process, thus the state machines are deterministic. The state machines do not model any of the characteristics of the physical layer of the protocol. The control logic is based on local timers and the reception and processing of incoming messages on a byte-by-byte basis. This approach can be translated in a fairly straight-forward way to the formalism of the CMSV model.

Figure 1 shows a high level view of the CMSV model of the MS/TP protocol for the case of two network nodes. Processes are represented as solid boxes and shared variables are represented by dashed boxes. The processes which interact with a shared variable are indicated by dashed lines. This model is intended to be used for analyzing the MAC layer of the protocol. Most of the details of the physical layer are abstracted away. The physical layer characteristics which are needed for the model are represented by the processes and variables in the shaded region of the figure.

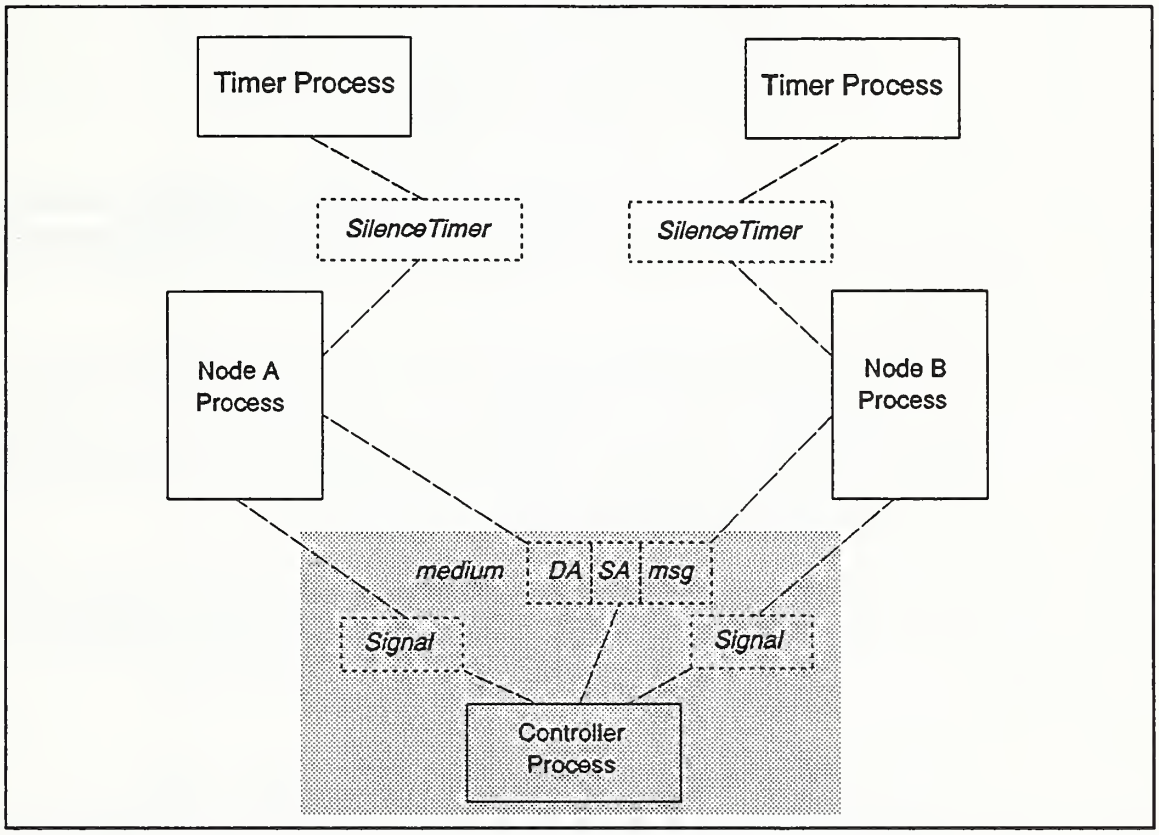

Figure 1. A high level view of the CMSV model of the BACnet MS/TP protocol.

Timing is very important in token passing protocols. In the draft standard an internal timer with a resolution of one Unit Time is assumed. In the CMSV model timers are represented by separate processes. A Timer process shares an integer variable, *SilenceTimer*, with a single network node process. The Timer process has write access to the *SilenceTimer* variable and the node process has both read and write access. The Timer process increments the *SilenceTimer* variable representing the passing of one Unit Time. The associated node process may read the variable's value when evaluating predicates and may also reset the value to zero when appropriate.

The MS/TP protocol has a bus topology. In a real network every node can see all of the traffic and terminators at each end of the bus absorb the electrical signals, thus removing the message from the bus. In the CMSV model, the variable *medium* is shared by all node processes and represents the ability to see all network traffic. All node processes have both read and write access to *medium*. A Controller process is defined to remove messages from *medium* as appropriate, representing the bus terminators. The Controller process is also used to introduce transmission errors.

Messages are considered to be atomic units represented by the shared variable *medium*. The variable *medium* is a data structure consisting of three parts; *medium.DA*, *medium.SA*, and *medium.msg*, representing the destination address, source address and data respectively. The variable *medium* may also take on the value ∅, which indicates that the bus is clear and no message is being transmitted, and Error, which indicates a framing error, or a checksum error. The field *medium.msg* can take any of the following values:

Token	- indicates a token frame
PFM	- indicates a poll-for-master frame
PFM Reply	- indicates a reply to PFM
Reply	- indicates a response to a confirmed service request
Confirmed_req	- indicates a confirmed service request
Unconfirmed_req	- indicates an unconfirmed service request

The BACnet application layer defines two kinds of service requests. A confirmed service request indicates that a reply is expected from the peer application layer. An unconfirmed service request indicates that no reply is expected. This distinction is a key factor in the protocol design because slave nodes must be allowed to respond to confirmed service requests.

Master nodes are distinguished from slave nodes by their address. Master nodes are given integer address in the range 0 - 31. Slave nodes are assigned addresses in the range 32 - 254. The address 255 is reserved for broadcast messages. For clarity, broadcast messages will be distinguished in the CMSV model by a destination of BROADCAST. In the CMSV model the local variable TS is used to hold the address for this station and the local variable NS is used to hold the value of the next station in the logical token ring. An NS value of 255 indicates that this station is the only master (true master-slave operation).

Interactions with upper layers of the protocol are not modeled except to provide local variables Inbuff and Outbuff, which hold messages which have been received and need to be passed to upper layers and messages from upper layers which need to be transmitted respectively. The transitions which clear Inbuff and fill Outbuff are not modeled explicitly.

3.1 Modeling a Master Node

The state machine for a master node, the most complex case, is described in the draft standard using 10 states, 35 transitions, and three procedures. The procedures are used to describe on a byte-by-byte basis the processes of receiving a frame, validating a frame, and sending a frame. Some of this complexity can be reduced by applying a slightly different level of abstraction. If messages are considered to be atomic units instead of a stream of bytes, all of the procedures and some of the transitions described in the draft standard can be abstracted away. Using this approach a master node process m_i is defined formally as the tuple $(S_i, s, L_i, N_i, \tau_i)$ where:

S_i = {Idle, Answer PFM, Receive, No Token, Use Token, Wait For Slave Reply, Done With Token, Poll For Master, Pass Token}

s = Idle

L = { MsgCount, TokenCount, Inbuff, Outbuff, NS, T_{idle}, T_{gen_idle}, T_{slot}, T_{no_token}, $T_{token\_timeout}$, $T_{reply\_timeout}$, N_{jabber}, N_{poll}, N_{retry_PFM}, N_{retry_token}, TS}

where:

MsgCount - integer representing the number of messages seen, initially 0

TokenCount - integer representing the number of tokens received since last reset, initially 0

Inbuff - structure of type *message* which buffers an incoming message

Outbuff - structure of type *message* which buffers an outgoing message

NS - integer representing the next node process in the ring, initially TS

All other variables are actually constants which represent fixed time intervals, and maximum values for counter variables. The draft standard defines these constants and prescribes their values. These constants are:

T_{idle} - integer representing the minimum Unit Times of silence between valid receive frames

T_{gen_idle} -integer representing the minimum Unit Times of silence required before transmitting the next frame (T_{idle} + 1)

T_{slot} - integer representing the Unit Times per address dependent reply slot

T_{no_token} - integer representing the Unit Times of silence before declaration of loss of token

$T_{token\ time\text{-}out}$ - integer representing the Unit Times of silence waiting for a station to begin using the token

$T_{reply\_timeout}$ - integer representing the maximum Unit Times of silence a slave can wait to begin transmitting.

N_{jabber} - integer representing the number of messages (bytes in the draft standard) received without a gap before "jabbering" is declared

N_{poll} - integer representing the number of tokens received or used before a Poll For Master cycle is executed

N_{retry_PFM} - integer representing the number of retries on sending Poll For Master

N_{retry_token} - integer representing the number of retries on sending token

TS - the address of this station (node process)

Figure 2 illustrates the transition state machine for the CMSV model of a master node process and Table 1 is the corresponding predicate-action table.

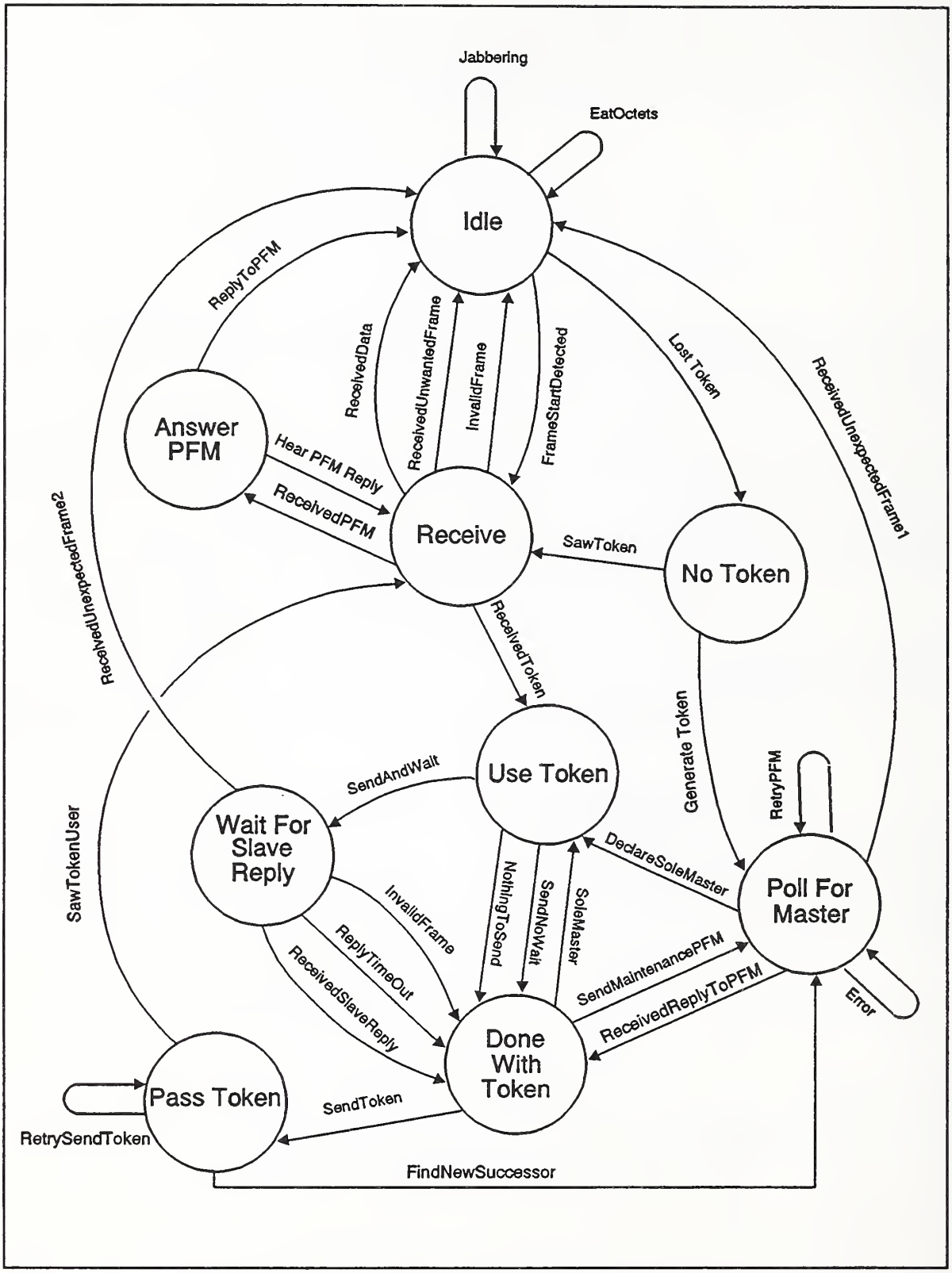

Figure 2. A state machine diagram for the CMSV model of a master node.

Table 1-A. Predicate - Action Table for Master Node Processes

Transition	Predicate	Action
LostToken	*SilenceTimer* > $T_{no\ token}$	----------
FrameStartDetected	*Signal*(TS) = clear ∧ *SilenceTimer* > T_{idle} ∧ *medium* ≠ ∅	*SilenceTimer* := 0; MsgCount := 0;
EatOctets	*Signal*(TS) = clear ∧ *SilenceTimer* < T_{idle} ∧ *medium* ≠ ∅ ∧ MsgCount < N_{jabber}	MsgCount := MsgCount + 1; *SilenceTimer* := 0; *Signal*(TS) := transceive;
Jabbering	*Signal*(TS) = clear ∧ *SilenceTimer* < T_{idle} ∧ *medium* ≠ ∅ ∧ MsgCount ≥ N_{jabber}	*SilenceTimer*, MsgCount := 0,0; *Signal*(TS) := transceive;
InvalidFrame	*Signal*(TS) = clear ∧ *medium* ≠ ∅ ∧ *medium.msg* = Error	*SilenceTimer* := 0; *Signal*(TS) := transceive;
ReceivedUnwantedFrame	*medium* ≠ ∅ ∧ *medium.DA* ∉ {TS, BROADCAST}	*SilenceTimer* := 0; *Signal*(TS) := transceive;
ReceivedToken	*medium* ≠ ∅ ∧ *medium.DA* = TS ∧ *medium.msg* = Token	TokenCount := TokenCount + 1; *Signal*(TS) := transceive;
ReceivedPFM	*medium* ≠ ∅ ∧ *medium.DA* = TS ∧ *medium.msg* = PFM	*SilenceTimer* := 0; *Signal*(TS) := transceive;
ReceivedData	*medium* ≠ ∅ ∧ *medium.DA* = TS ∧ *medium.msg* ∈ {Reply Unconfirmed_req, Confirmed_req}	*SilenceTimer* := 0; Inbuff := *medium.msg*; *Signal*(TS) := transceive;

GenerateToken	*SilenceTimer* ≥ T_{no_token} + TS * T_{slot}
	∧ *medium* = ∅
		TokenCount, RetryCount := 0,0;
		SilenceTimer := 0;
		medium.DA := BROADCAST;
		medium.SA := TS;
		medium.msg := PFM;
		Signal(TS) := transceive;

SawToken	*medium* ≠ ∅	*SilenceTimer* := 0; *Signal*(TS) := transceive;

Table 1-B. Predicate - Action Table for Master Node Processes - cont.

Transition	Predicate	Action
DeclareSoleMaster	*medium* = ∅ ∧ *SilenceTimer* ≥ T_{slot} * 32 ∧ RetryCount ≥ $N_{retry\ PFM}$	*SilenceTimer*, NS := 0, 255; RetryCount := 0;

SendMaintenancePFM	*medium* = ∅
	∧ TokenCount ≥ N_{poll}
	∧ NS ≠ TS + 1
		TokenCount, RetryCount := 0, 0;
		SilenceTimer := 0;
		medium.DA := BROADCAST;
		medium.SA := TS;
		medium.msg := PFM;
		Signal(TS) := transceive;

SoleMaster	*medium* = ∅ ∧ TokenCount < N_{poll} ∧ NS = 255	----------

SendToken	TokenCount < N_{poll} ^ NS ≠255
	∧ *medium* = ∅
		SilenceTimer := 0;
		medium.DA := NS;
		medium.SA := TS;
		medium.msg := TOKEN;
		Signal(TS) := transceive;

RetrySendToken	*SilenceTimer* > $T_{token\ time\text{-}out}$
	∧ *medium* = ∅
	∧ RetryCount < N_{retry_token}
		SilenceTimer := 0;
		medium.DA := NS;
		RetryCount := RetryCount + 1;
		medium.SA := TS;
		medium.msg := TOKEN;
		Signal(TS) := transceive;

SawTokenUser	*SilenceTimer* ≤ $T_{token\ time\text{-}out}$ ∧ *medium* ≠ ∅	*SilenceTimer* := 0; *Signal*(TS) := transceive;

FindNewSuccessor	*SilenceTimer* > $T_{token\ time\text{-}out}$
	∧ *medium* = ∅
	∧ RetryCount ≥ N_{retry_token}
		SilenceTimer, NS := 0,255;
		RetryCount, TokenCount := 0,0;
		medium.DA := BROADCAST;
		medium.SA := TS;
		medium.msg := PFM;
		Signal(TS) := transceive;

ReplyToPFM	*medium* = ∅
	∧ *SilenceTimer* ≥ T_{slot} * Addr_Dist
	∧ *SilenceTimer* ≤ Tslot *
	(Addr_Dist + 1)
		SilenceTimer := 0;
		medium.DA := BROADCAST;
		medium.SA := TS;
		medium.msg := PFM Reply;
		Signal(TS) := transceive;

Error	*Signal*(TS) = clear ∧ *medium* = Error	*Signal*(TS) := transceive;

Table 1-C. Predicate - Action Table for Master Node Processes - cont.

Transition	Predicate	Action

RetryPFM	*SilenceTimer* ≥ T_{slot} * 32
	∧ RetryCount < N_{retry_PFM}
	∧ *medium* = ∅
		SilenceTimer := 0;
		RetryCount := RetryCount + 1;
		medium.DA := BROADCAST;
		medium.SA := TS;
		medium.msg := PFM;
		Signal(TS) := transceive;

ReceivedUnexpectedFrame1	*Signal*(TS) = clear ∧ *SilenceTimer* ≥ T_{idle} ∧ (*medium.DA* ≠ TS ∨ *medium.msg* ≠ Reply)	*SilenceTimer* := 0; *Signal*(TS) := transceive;
ReceivedUnexpectedFrame2	*Signal*(TS) = clear ∧ *SilenceTimer* ≥ T_{idle} ∧ (*medium.DA* ≠ TS ∨ *medium.msg* ≠ PFMReply)	*SilenceTimer* := 0; Signal(TS) := transceive;
ReceivedReplyToPFM	*Signal*(TS) = clear ∧ *SilenceTimer* ≥ T_{idle} ∧ *medium.DA* = TS ∧ *medium.msg* = PFM Reply	*SilenceTimer*, RetryCount := 0, 0; TokenCount := 0; NS := *medium.SA*; *Signal*(TS) := transceive;
ReceivedSlaveReply	*Signal*(TS) = clear ∧ *SilenceTimer* ≥ T_{idle} ∧ *medium.DA* = TS ∧ *medium.msg* = Reply	inbuff := *medium.msg*; *Signal*(TS) := transceive;

SendAndWait	*Signal*(TS) = clear
	∧ outbuff.msg = confirmed_msg
	∧ outbuff.DA > 32
		SilenceTimer := 0;
		medium.DA := Outbuff.dest;
		medium.SA := TS;
		medium.msg := Outbuff.msg;
		Signal(TS) := transceive;

ReplyTimeOut	*SilenceTimer* > $T_{reply\ timeout}$	----------
HearPFMReply	*Signal*(TS) = clear ∧ *medium* ≠ ∅ ∧ (*SilenceTimer* < T_{slot} * Addr_Dist ∨ *SilenceTimer* > T_{slot} * (Addr_Dist + 1)	*SilenceTimer* := 0; *Signal*(TS) := transceive;
NothingToSend	Outbuff = ∅ ∧ *Signal*(TS) = clear	----------

SendNoWait	*Signal*(TS) = clear
	∧ (Outbuff.msg =
	Unconfirmed_msg
	∨ outbuff.DA ≤ 32)
		SilenceTimer :=0;
		medium.DA := outbuff.dest;
		medium.SA := TS;
		medium.msg := outbuff.msg;
		Signal(TS) := transceive;

3.2 Modeling a Slave Node

The state machine for a slave node is modeled in the draft standard using 4 states, 12 transitions, and the three procedures described above for master nodes. Once again the assumption of atomic messages can be used to simplify the state machine. A slave process is defined formally as the tuple (S_i, s, L_i, N_i, τ_i) where:

S_i = {Idle, Receive, Answer Data Req}

s = Idle

L = {Inbuff, Outbuff, T_{idle}, $T_{reply\_timeout}$, TS} where:

Inbuff - structure of type *message* which buffers an incoming message

outbuff - structure of type *message* which buffers an outgoing message

All other variables are actually constants which represent fixed time intervals, and maximum values for counter variables. The draft standard defines these constants and prescribes their values. These constants are:

T_{idle} - integer representing the minimum Unit Times of silence between valid receive frames

$T_{reply\_timeout}$ - integer representing the maximum Unit Times of silence a slave can wait to begin transmitting.

TS - the address of this station (node process)

Figure 3 illustrates the transition state machine for the CMSV model of a slave node process and Table 3 is the corresponding predicate-action table for the process.

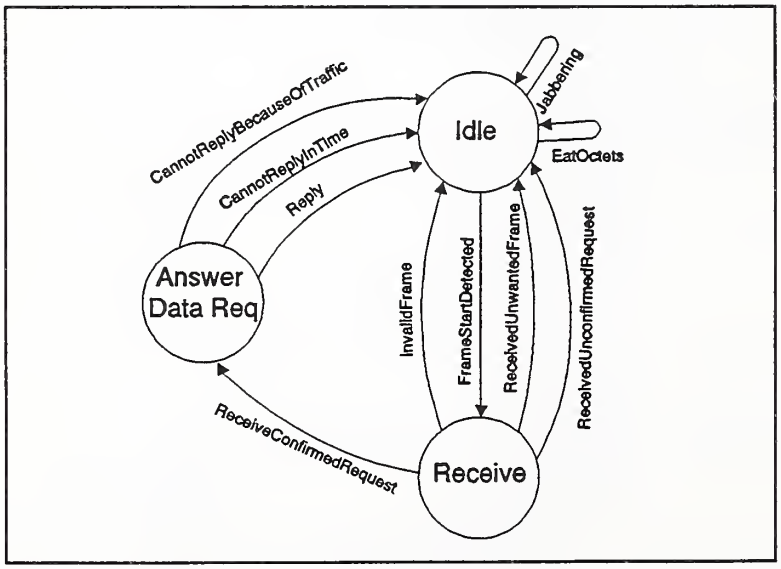

Figure 3. State machine diagram for the CMSV model of a slave node.

Table 2. Predicate - Action Table for Slave Node Processes

Transition	Predicate	Action
EatOctets	*SilenceTimer* < T_{idle} ∧ *medium* ≠ ∅ ∧ MsgCount < N_{jabber}	MsgCount := MsgCount + 1; *SilenceTimer* := 0; *Signal*(TS) := transceive;
Jabbering	*SilenceTimer* < T_{idle} ∧ *medium* ≠ ∅ ∧ MsgCount ≥ N_{jabber}	*SilenceTimer*, MsgCount := 0,0; *Signal*(TS) := transceive;
FrameStartDetected	*SilenceTimer* > T_{idle} ∧ *medium* ≠ ∅	*SilenceTimer* := 0; MsgCount := 0;
ReceivedUnwantedFrame	*medium* ≠ ∅ ∧ *medium.DA* ∉ {TS, BROADCAST}	*SilenceTimer* := 0; *Signal*(TS) := transceive;
InvalidFrame	*medium* ≠ ∅ ∧ *medium* = Error	*SilenceTimer* := 0; *Signal*(TS) := transceive;
ReceivedConfirmedRequest	*medium* ≠ ∅ ∧ *medium.DA* = TS ∧ *medium.msg* = confirmed_req	*SilenceTimer* := 0; Inbuff.msg := *medium.msg*; master = *medium.SA;* *Signal*(TS) := transceive;
ReceivedUnconfirmedRequest	*medium* ≠ ∅ ∧ *medium.DA* = TS ∧ *medium.msg* = confirmed_req	*SilenceTimer* := 0; inbuff.msg := *medium.msg*; *Signal*(TS) := transceive;

Reply	*SilenceTimer* < $T_{reply\_timeout}$
	∧ inbuff.msg = confirmed_req
	∧ *medium* = ∅
		medium.DA = master;
		medium.SA = TS;
		medium.msg = reply;

CannotReplyInTime	*SilenceTimer* ≥ $T_{reply\ timeout}$	----------
CannotReplyBecauseOfTraffic	*Signal*(TS) = clear ∧ *SilenceTimer* < $T_{reply\ timeout}$ ∧ *medium* ≠ ∅	----------

3.3 Modeling the Communication Bus

The physical communication medium of an MS/TP network is a bus made of shielded, twisted-pair wires. The bus is equipped with pull-up and pull-down resisters which serve as active terminators and ensure that an undriven communications line is in a known state. The communication hardware is a UART, capable of transmitting and receiving eight data bits with one stop bit and no parity, an EIA-485 transceiver, and a timer with resolution of one Unit Time.

The model of the physical layer is an abstraction of those characteristics which are necessary to model and analyze the MAC layer of the protocol. Thus the physical and electrical characteristics of the medium and the data encoding will not be included. In the actual bus, signals propagate to the end and are then terminated. In order to model this behavior, an additional process called the "Controller" is used to "clear" the medium after all of the node processes have seen the message. A shared variable array, *Signal(1..n)*, with one element for each network node,

is used to coordinate between the node processes and the Controller process, so that the Controller "knows" when to clear the medium.

The Controller process has two states, 0 and 1. The initial state is 0. The Controller process has both read and write access to the shared variable *Signal* and it has no local variables. Figure 4 illustrates the Controller process state machine and Table 4 is the corresponding predicate-action table. Each element of *Signal* may take on one of two values: clear and transceive. When a message is placed in *medium*, the Controller will make a transition to state 1. When the other processes see the message on the bus they set their corresponding *Signal* value to "transceive". When all of the node processes have seen the message the Controller can execute the Reset transition which clears the bus and the array *Signal*. This is approach to modeling the bus is similar to the approach used by Lundy and Miller to model a CMSA/CD protocol [7].

The Controller can also introduce errors by writing the value "Error" to *medium*. With an EIA-485 physical layer protocol it is difficult to tell the difference between transmission errors and collisions due to the presence of multiple tokens. This is because EIA-485 uses a logic level method to determine the sense of a bit. When two transmissions collide, the sense of one or more bits may be changed due to destructive interference, but there is no way to distinguish this from any other interference which corrupts the signal.

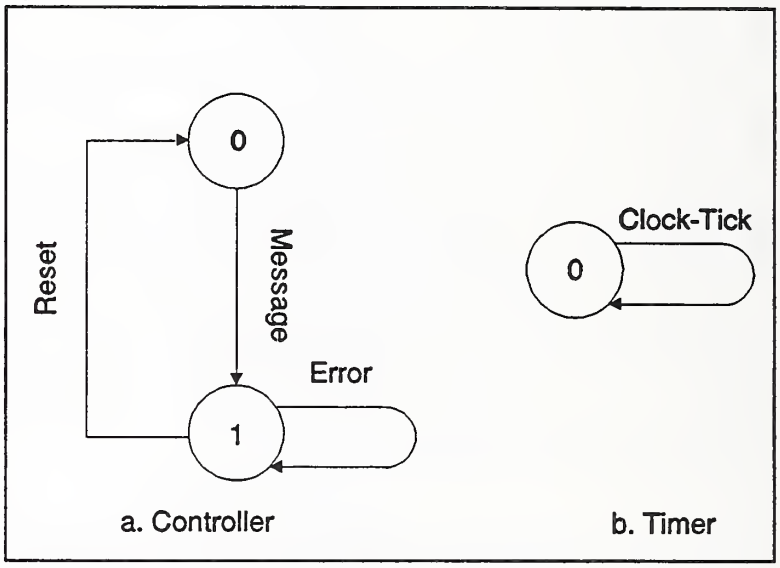

Figure 4. State machine diagram for the controller process and the timer process.

Because collisions cannot be detected another approach is needed to determine when multiple tokens are present. If a device has a token and receives an unexpected frame, the protocol assumes that the cause is the presence of multiple tokens. The receiving process then drops the token and returns to the idle state (see the ReceivedUnexpectedFrame1 and ReceivedUnexpectedFrame2 transitions in Table 1). This unexpected frame could actually be the result of a late reply to a poll-for-master or a confirmed service request to a slave. If this is the real cause the result of the protocol action is loss of the token.

TABLE 3. Predicate-Action Table for the Controller Process

Transition	Predicate	Action
Message	*medium* $\neq \varnothing$	------
Reset	*Signal*(1..n) = transceive	*medium* := Ø; *Signal(1..n)* := clear;
Error	TRUE	*medium* := Error;

Theorem 1. If a process i transmits a message (writes to the shared variable *medium*) the Controller process will clear *medium* in a finite time.

proof. Proof of this Theorem requires showing that each node process eventually sets its corresponding *Signal*(i) value to transceive. Consider a master process in any local system state except the Use Token state or the Done With Token state (they will be considered separately). It is clear from examining Figure 1 and Table 1 that for each of these possible system states there exists one or more transitions which becomes enabled if *medium* $\neq \varnothing$. Examination of Figure 2 and Table 2 shows that for slave processes the same property holds for every possible system state. Furthermore, for each such system state in both master and slave processes there is exactly one such transition which becomes enabled for each possible message and no other local transitions can be simultaneously enabled. This transition remains enabled until it is fired. In each case the corresponding action involves setting the value of Signal(i) to transceive. By the fairness assumption these transitions will be executed, thereby setting *Signal*(i) to transceive.

Consider the case of a master process, i, in the Use Token state. If another process, j, writes to the variable *medium* this represents an error condition caused by multiple tokens or timers that are out of synchronization. If process i has just entered the Use Token state and the received token which caused it to enter this state has not yet been cleared by the Controller process, then Signal(i) = transceive is already true. If *Signal*(i) = clear and i has a message to send, it will execute either the SendNoWait or the SendAndWait transition and set *Signal*(i) to transceive. If *Signal*(i) = clear and i has no message to send it will execute the NothingToSend transition and enter the Done With Token state.

Consider the case of a master process, i, in the Done with token state and *Signal*(i) = clear. Either the SendToken transition or the SendMaintenancePFM transition must be enabled. When this transition fires it will set the *Signal*(i) value to transceive.

This covers all possible cases, therefore eventually it will be the case that $\forall$ i *Signal*(i) = transceive.

At this point the Controller's Reset transition becomes enabled. Once the Receive transition is enabled it remains enabled and by the fairness assumption will eventually fire, thereby clearing *medium*.

3.4 Modeling Time

In a token passing protocol there should never be more than one node attempting to transmit a message at a time. Enforcing this requires careful timing considerations. As indicated above, timers for the MS/TP protocol are only required to have a one Unit Time resolution. With a resolution this coarse it is expected that from time to time a token will be lost or multiple tokens will be generated. The protocol is intended to be biased in favor of lost tokens. Nevertheless, provisions are included to handle the generation of multiple tokens.

In order to model the timing aspects of the protocol a special Timer process is introduced for each node process. The Timer process has only one state and one transition. Whenever the *medium* is clear the Timer process can execute this transition and increment the *SilenceTimer* for its corresponding process. The predicate-action for this transition is:

Transition	Predicate	Action
Clock-Tick	*medium*	*SilenceTimer* := *SilenceTimer* +1;

The restriction that the Timer can only be incremented when *medium* = $\varnothing$ reflects the fact that the real clocks are synchronized to activity on the bus. Since all nodes see the same traffic, the clocks drift out of synchronization during periods of silence, not while messages are being transmitted. In a real system several Unit Times may pass while a message is being transmitted, but in the CMSV model messages are atomic units. Later in the paper further restrictions will be placed on Timer process in order to determine how closely the process clocks must be

synchronized in order to avoid generating multiple tokens.

4. ANALYSIS OF MASTER-SLAVE OPERATION

This section presents an analysis of safety properties, liveness properties, and timing considerations for master-slave operation of the MS/TP protocol. Safety is defined as the absence of deadlock. Liveness is defined as follows: ∀ master node processes i with a message m to send, m will be transmitted within a finite amount of time. For this analysis it is assumed that there is a single master node process in the token ring and N slave processes.

Lemma 1 After a finite amount of time, the master node process will enter and never leave the set of states defined by $S_i \in$ {Use Token, Poll For Master, Done With Token, Wait for Slave Reply} $\wedge$ NS = 255.

proof. The proof is presented in two steps. Step 1 is to prove that the master node process must enter the specified set of states. Step 2 is to prove that once the process has entered one of these states it cannot execute a transition which will leave it in a state outside of this set.

Step 1. When the master node powers-up or is initialized the system will be in one of the global states constrained by the following:

∀i *Signal*(i) = clear		
medium = ∅		
master node process	-	S_i = Idle
		SilenceTimer = 0
		NS = TS
∀ slave nodes	-	S_i = Idle
		SilenceTimer ≥ 0
Controller	-	S_i = 0

No other global states are possible because slave processes cannot initiate messages and there is only one master process.

From one of these global states the only transitions that are enabled are Clock-Tick transitions for the various Timer processes. These transitions occur in an arbitrary order until the *SilenceTimer* variable associated with the master node process becomes greater than T_{no_token}. At this point the LostToken transition becomes enabled and must eventually fire, placing the master node process in the No Token state. Once again only Clock-Tick transitions are enabled and eventually conditions will be met for the master node process to execute the GenerateToken transition and enter the Poll For Master state.

At this point the system is in a global state such that the master process has no enabled transitions, the Controller has the Message transition enabled, and each slave process has exactly one transition that can "receive" the message enabled. The Controller and slave process can execute these transitions in an arbitrary order. Eventually each slave process will receive this message or an error, depending on the order of Controller transitions, and set the corresponding *Signal* variable to transceive. This enables the Controller Reset transition which eventually executes, clearing the medium. Since slave processes do not respond to poll-for-master frames, the master node will retry the poll until the DeclareSolemaster transition is enabled and executed. Executing this transitions sets NS = 255 and leaves the master node in one of the states in the desired set.

Step 2. Only four transitions can cause the master node process to leave this set of states, ReceivedReplyToPFM, ReceivedUnexpectedFrame1, ReceivedUnexpectedFrame2, and SendToken. Consider the ReceivedReplyToPFM transition first. This transition can become enabled only if another process replies to a previously issued PFM frame. Since slave processes cannot do this and a master process cannot reply to its own poll, this transition never becomes enabled.

The ReceivedUnexpectedFrame1 and ReceivedUnexpectedFrame2 transitions can be considered together. In both cases the transition becomes enabled because the master node process is expecting a particular reply but receives instead a valid message of an unexpected type. Two possible cases must be considered:

case 1 - the unexpected message was sent by a master process
case 2 - the unexpected message was sent by a slave process.

Since there is only one master process case one is impossible. Examination of the slave process transitions shows that the only message a slave can send is a Reply and that reply comes only in response to a previously received confirmed service request. Since a Reply message is expected when the master process is in the Wait For Slave Reply state this case does not enable the ReceivedUnexpectedFrame2 transition. The only potential problem is if a slave transmits a Reply after the master process has left the Wait For Slave Reply state. The slave must reply while *SilenceTimer* $< T_{\text{reply timeout}}$ and the master process must remain in the Wait For Slave Reply state until a message is received or *SilenceTimer* $> T_{\text{reply\_timeout}}$. Therefore the ReceivedUnexpectedFrame2 transition can never be enabled as long as the two process timers are synchronized. The timers become synchronized when the confirmed request is transmitted and must remain in synchronization only for $T_{\text{reply\_timeout}}$ Unit Times. This is a reasonable assumption as long as $T_{\text{reply\_timeout}}$ is short.

The SendToken transition can become enabled only when NS $\neq$ 255. For all of the global states in the specified set NS = 255. The only transition which can assign a value other than 255 to NS is the ReceivedReplyToPFM transition. It was shown above that this transition cannot be enabled. Therefore the SendToken transition cannot become enabled.

These cases exhaust the possibilities, therefore Lemma 1 holds. Since the global states defined in Lemma 1 all indicate that the master process has a token, the following corollary holds.

Corollary 1 After a finite time the system will reach a global state with exactly one token and will remain forever in a state with exactly one token.

Lemma 2 If there is a single master process and it is in the Use Token state, it will leave this state and return in a finite time.

proof. A master node process has three transitions leaving the Use Token state; NothingToSend, SendNoWait, and SendAndWait. The enabling predicates for these transitions all depend solely on the status of the local variable Outbuff. If Outbuff is empty (∅) then NothingToSend is enabled. If Outbuff is not empty and Outbuff.msg contains unconfirmed_msg, then SendNoWait is enabled. If Outbuff is not empty and Outbuff.msg contains confirmed_msg, then SendAndWait is enabled. Since this exhaust all possible cases, one of these transitions must be enabled. By the fairness assumption this enabled transition must eventually fire causing the master process to leave the Use Token state.

After leaving the Use Token state, the master process will be in either the Wait For Slave Reply state or the Done With Token state. Consider the Wait For Slave Reply case first. From Lemma 1 it is clear that the UnexpectedFrame2 transition can never be enabled. If a message is received either the ReceivedSlaveReply transition or the InvalidFrame transition must become enabled. If a message is not received, then the timer process will eventually increment *SilenceTimer* until the ReplyTimeOut transition becomes enabled. Thus it must be the case that one of the transitions to the Done With Token state must become enabled and eventually fire causing the master process to enter the Done With Token state.

From Lemma 1 it is clear that the SendToken transition cannot become enabled and that NS = 255. Therefore, either the SoleMaster transition or the SendMaintenancePFM transition must be enabled. If the SoleMaster transition executes the process returns to the Use Token state and the Lemma holds. If the SendMaintenancePFM

transition executes the process enters the Poll For Master state. The proof for Lemma 1 showed that the master process must eventually leave the Poll For Master state by executing the DeclareSoleMaster transition, returning the process to the Use Token state. Therefore, the master process must eventually return to the Use Token state.

Theorem 2. The master process in an MS/TP protocol system with a single master is deadlock free.

proof. The proof to Lemma 1 showed that, once initiated, the master process must enter the Use Token state in finite time and NS = 255. From Lemma 2 the process must leave this state and eventually return. Therefore, the master process is deadlock free.

Theorem 3. The liveness property holds for MS/TP systems with a single master process.

proof. The proof to Lemma 1 showed that, once initiated, the master process must enter the Use Token state in finite time. From Lemma 2, the master process must eventually leave the Use Token state. If the process has a message, m, to send, it can only leave the Use Token state by executing either the SendNoWait transition or the SendAndWait transition. For either case the theorem holds. If the master process has no message to send and leaves the Use Token state, from Lemma 2 the master process will eventually reenter the Use Token state.

Theorem 4. A slave process which leaves the Idle state will eventually return.

proof. A slave process which leaves the Idle state must enter the Receive state. It must be the case that *medium* $\neq \varnothing$ and exactly one of the four transitions leaving the Receive state is enabled. This transition must remain enabled until it is executed. Three of the possible transitions cause the process to return to the Idle state satisfying the Theorem. The fourth transition causes the process to enter the Answer Data Req state.

When the process enters the Answer Data Req state, none of the transitions defined for this state can be enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. At any time while *SilenceTimer* $< T_{reply\_timeout}$ the upper layers can generate a response, enabling the Reply transition. If this transition executes the process enters the Idle state and the Theorem is satisfied. If a reply is not ready and the medium remains clear, the Timer process will increment *SilenceTimer* until the CannotReplyInTime transition becomes enabled. Once enabled, this transition will remain enabled until executed. Execution of this transition returns the process to the Idle state satisfying the Theorem. If a new message is detected this means either that the slave process *SilenceTimer* is out of synchronization with the master process timer and the master has executed a retry or, in the case of multiple masters, there may be more than one token. For either case the CannotReplyBecauseOfTraffic transition becomes enabled an remains enabled until executed. Execution of this transition caused the precess to return to the Idle state satisfying the Theorem. This exhausts all possible cases.

5. ANALYSIS OF TOKEN-PASSING OPERATION

This section presents an analysis of safety properties, liveness properties, and timing considerations for token-passing operation of the MS/TP protocol. Safety and liveness are defined in the same way that they were for the master-slave analysis.

Theorem 5. An MS/TP protocol system with multiple master processes is deadlock free.

proof. The proof involves examining all possible system states for an arbitrary master process and showing that in a finite time a transition must occur which causes the process to enter a new system state.

Case 1: The Idle State. For a process with a local system state of Idle, two cases must be considered: 1) the medium is free; and 2) there is a message to be received. If the medium is free, the Clock-Tick transition for the timer process associated with this node must be enabled continuously until there is a message on the medium. This

will cause the *SilenceTimer* to be repeatedly incremented until *SilenceTimer* > T_{no_token}. At this point the LostToken transition becomes enabled and execution of this transition causes the process to enter a new state.

If a message appears at any time while the process is in the Idle state and *SilenceTimer* > T_{idle}, then the FrameStartDetected transition becomes enabled and remains enabled until it is executed. Execution of this transition causes the process to enter a new system state.

If a message appears at any time while the process is in the Idle state and *SilenceTimer* ≤ T_{idle} either the EatOctets or Jabbering transition must become enabled. Jabbering is an error condition. If it is assumed that errors are an infrequent occurrence, a process cannot become trapped in the idle state indefinitely while another process is jabbering. Either the FrameStartDetected or the LostToken transition will eventually become enabled and executed. This exhausts all possible cases for a process in the Idle state.

Case 2: The Answer PFM State. When a process i enters the Answer PFM state *medium* ≠ ∅ ∧ *Signal*(i) = transceive. None of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. At this point there are again two possible cases. The first case is that a message is detected while *SilenceTimer* < T_{slot} * Addr_Dist. This causes the HearPFMReply transition to become enabled and remain enabled until executed. Execution of this transition causes the process to enter a new system state. The second case occurs when the medium remains empty long enough for the Timer process associated with this node to increment the *SilenceTimer* enough times that *SilenceTimer* ≥ T_{slot} * Addr_Dist, enabling the ReplyToPFM transition. If this transition fires, the process will enter a new system state.

If the ReplyToPFM transition does not fire, *SilenceTimer* will be incremented until *SilenceTimer* ≥ T_{slot} * (Addr_Dist + 1), disabling the transition. The set of global states consistent with this local system state requires that there be another process which has a token. That process must eventually use the token, pass the token or conduct a poll-for-master cycle. Any of these circumstances will enable the local HearPFMReply transition and it will remain enabled until executed, causing the local process to enter a new system state.

Case 3: The Receive State. in order for a process to enter the receive state there must be a message on the medium. For each possible message there is a corresponding transition out of the Receive state which is enabled. This receive transition will remain enabled until it is executed or until the Controller process executes an Error transition. If an error occurs the InvalidFrame transition becomes enabled and remains enabled until executed. Thus the local process must eventually execute a transition which leaves it in a new system state.

Case 4: The No Token State. When a process enters the No Token state the medium must be clear. If a message is detected at any time while the process is in this state the SawToken transition becomes enabled and remains enabled until executed. Execution of this transition will cause the process to enter a new system state. If the medium remains clear the Timer process associated with this process will increment *SilenceTimer* until *SilenceTimer* ≥ T_{no_token} + TS * T_{slot}. At this point the GenerateToken transition becomes enabled and remains enabled until executed or until a message is detected. Either way, the process will eventually enter a new system state.

Case 5: The Use Token State. When a process i enters the Use Token state *medium* ≠ ∅ ∧ *Signal*(i) = transceive. None of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. When the medium becomes clear, either process i has a message to send or it does not. If it does not have a message to send, the NothingToSend transition is enabled. If the process has a message to send either the SendAndWait or the SendNoWait transition must be enabled. Execution of any of these transitions will cause the process to enter a new system state.

Case 6: The Wait For Slave Reply State. When a process i enters the Wait For Salve Reply state *medium* ≠ ∅ ∧ *Signal*(i) = transceive. None of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. If a message is detected anytime after the

medium is cleared and the process is still in the Wait For Slave Reply state, then one of the transitions {ReceivedUnexpectedFrame2, InvalidFrame, ReceivedSlaveReply} must be enabled and will stay enabled until executed. Execution of any of these transitions will cause the process to enter a new state. If the medium remains clear the Timer process associated with this node will increment *SilenceTimer* until *SilenceTimer* > $T_{reply\_timeout}$. At this point the ReplyTimeOut transition becomes enabled until it is executed or until a message is detected. Either way the process must eventually execute a transition which causes it to enter a new system state.

Case 7: The Poll For Master State. When a process i enters the Poll For Master state *medium* $\neq \varnothing \wedge$ *Signal*(i) = transceive. None of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. If a message is detected anytime after the medium is cleared and the process is still in the Poll For Master state, then either the ReceivedReplyToPFM transition or the ReceivedUexpectedFrame1 transition must be enabled and remain enabled until executed. Execution of either of these transitions will cause the process to enter a new system state. If the medium is clear, the Timer process associated with this node will increment the *SilenceTimer*. If the medium remains clear sufficiently long the DeclareSoleMaster transition will become enabled (possibly after retries) and remain enabled until executed or until a message is received. In either case the process will execute a transition which leaves it in a new system state.

Case 8: The Done With Token State. When a process i enters the Done With Token state, if *medium* $\neq \varnothing \wedge$ *Signal*(i) = transceive then none of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. When the medium is cleared, exactly one of the three transitions leaving the Done With Token state must be enabled. This transition will remain enabled until it is executed and execution of the transition will leave the process in a new system state.

Case 9: The Pass Token State: When a process i enters the Done With Token state, if *medium* $\neq \varnothing \wedge$ *Signal*(i) = transceive then none of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. If a message is detected after the medium is cleared and the process is still in the Pass Token state, the SawTokenUser transition becomes enabled and remains enabled until executed. If the medium remains clear, the timer process begins to increment *SilenceTimer*. Eventually this will cause the RetrySendToken transition to become enabled and remain enabled until executed or until a message is detected. If the RetrySendToken transition is executed the *SilenceTimer* is cleared and the process begins again. This can repeat until the FindNewSuccessor transition becomes enabled. Once enabled this transition will remain enabled until executed or until a message is detected. Either way, executing the transition will cause the process to enter a new system state.

Lemma 3. In an MS/TP protocol system with multiple masters, all master processes which do not currently have a token will receive a token in finite time.

proof. The proof contains three parts. The first part shows that if the system is in a global state such that no token exists, eventually a token will be generated. The second part shows that if a process has the token it must pass the token to its successor in finite time. The third part shows that all master processes become the successor of the current token holder in finite time. The proof requires an important assumption, namely that errors are a rare occurrence.

Consider a global state where no token exists. If a master process is in any of the states {Use Token, Poll For Master, Done With Token, Pass Token, Wait For Slave Reply} then this process has a token. If a master process is in any of the states {Answer PFM, Receive} then the process has just received a message and some other process has a token. Therefore, if no token exists, all master processes must be in either the Idle state or the No Token state. The only transitions that can be enabled in such a global state are the Clock-tick transitions of the various Timer processes. If a master process is in the Idle state, execution of its Timer process will eventually enable the Lost Token transition and cause it to enter the No Token state. From the proof to Theorem 5, a master process in the No Token state must eventually leave. Only two possibilities exist, the SawToken transition and the Generate

Token transition. The SawToken transition can only become enabled if a message is generated by another process which indicates that some other process has generated a token. Execution of the GenerateToken transition generates a token. Therefore a token must eventually be generated.

Consider a master process in one of the system states {Use Token, Poll For Master, Done With Token, Pass Token, Wait For Slave Reply}, meaning that it has a token. Assume that there are no errors. Since the ReceivedUnexpectedFrame1, ReceivedUnexpectedFrame2, and FindNewSuccessor transitions all represent error conditions none of these transitions can become enabled. If the process is in the Use Token state it must eventually leave and enter either the Wait For Slave Reply state or the Done With Token state. Consider the Wait For Slave Reply state first. Since there are no errors, the process will eventually leave the Wait For Slave Reply state and enter the Done With Token state. From the Done With Token state the process can enter either the Poll for Master state or the Pass Token state. Consider the Poll For Master case first. Since there are no errors and multiple master processes, the local process must leave this state by executing the ReceivedReplyToPFM transition. Execution of this transition leaves the process in the Done With Token state with TokenCount = 0. This makes immediate return to the Poll For Master State impossible and the process must eventually enter the Pass Token state. Since there are no errors the process must leave the Pass Token state by executing the SawTokenUser transition which indicates that the token has been successfully passed to the successor process. Therefore, if there are no errors a process with the token must pass the token to its successor process.

Consider the consequences of errors. Errors which cause the ReceivedUnexpectedFrame1 or ReceivedUnexpectedFrame2 transitions to be executed cause a token to be lost and can result in one of two possible outcomes. If there were multiple tokens then the MS/TP system continues to operate with one fewer token. If there was only one token and a timing error caused the transitions to become enabled the system will reach a global state with no tokens and eventually a new token will be generated.

The FindNewSuccessor transition is executed because repeated attempts to pass the token have failed. The result of executing this transition is that a new successor is found. If the original successor process has not failed this could cause this process to be removed from the token ring. It will be shown below that it will have an opportunity to reenter the ring at a later time. These errors only prevent a process from passing the token if they happen infinitely often, hence the assumption the errors are a rare occurrence.

Consider the case of a process i with NS = k and a third process j which is not in the token ring. Because of process j's address it should be the successor to i. When process i enters the Done With Token State after having received the token N_{poll} times, the only transition which can become enabled is the SendMaintenancePFM transition. Execution of this transition caused the process to enter the Poll For Master state and issue a poll for master. This gives process j a chance to enter the token ring. With the assumption that errors are a rare occurrence, process j will eventually enter the token ring.

Theorem 6. The liveness property holds for MS/TP systems with multiple masters.

proof. From Lemma 3 it follows that any master process i with a message to transmit will eventually receive the token. A process receives the token by executing the ReceivedToken transition which leaves the process in the local Use Token state. When the process enters the Use Token state $medium \neq \varnothing \wedge Signal(i) = \text{transceive}$. None of the transitions defined for this state can become enabled until the medium is cleared. It follows from Theorem 1 that this will eventually happen. Since the process has a message to send, either the SendNoWait transition or the SendAndWait transition must be enabled. This transition will remain enabled until it is executed and the message is transmitted.

6. TIMING CONSIDERATIONS

In proving the safety and liveness properties for an MS/TP protocol system with a single master process it was assumed that the timers in master and slave processes remain synchronized for at least $T_{reply\ timeout}$ Unit Times. This is necessary to achieve the desired operation of the system but is not strictly necessary to achieve safety and liveness. The Token-Passing case is more general than the Master-Slave case and both safety and liveness were proved without the assumption. A timing error of this sort will degrade performance but not prevent useful communication.

There are other important factors involved in selecting the appropriate value for $T_{reply\ timeout}$. It must be sufficiently long to permit a slave device to do the processing needed to generate the reply. The time-out constants are specified in terms of Unit Times to accommodate networks with different transmission speeds. This causes a problem for this particular constant because the processing time a slave needs to generate a reply is independent of the transmission speed of the bus. If $T_{reply\ timeout}$ is large enough to ensure that the slave has sufficient time in a high speed network this will lower the efficiency of a low speed network because the time-out value will be too long. On the other hand, if $T_{reply\ timeout}$ is selected to optimize the low speed network, a slave will not have time to reply in the high speed case. This problem suggest that this timer should be specified in absolute time or else one transmission speed should be specified in the standard.

The protocol specification states that messages must be separated by at least T_{gen_idle} Unit Times ($T_{gen\_idle} = T_{idle} + 1$). There is no transition defined for a process in the Wait For Slave Reply state which accounts for the possibility of a message being received when *SilenceTimer* $< T_{idle}$. In order to avoid an ambiguity under these conditions it is necessary to establish that the timers remain synchronized within 1 Unit Time over a period of T_{gen_idle} Unit Times. Another alternative could be to remove the *SilenceTimer* $\geq T_{idle}$ condition on the predicates for transitions leaving the Wait For Slave Reply state. Under these conditions, if the slave timer begins to run ahead of the master timer, the reply can still be received correctly. If the slave timer begins to run behind the master timer and the master times-out before the slave transmits a reply, a collision might occur, but the master process will still continue to function.

There are two places in the protocol where a slotted time window is used, when polling for masters and when the token is lost. It is important that T_{slot} be appropriately sized so that timer drift does not cause one process to encroach on another process's slot. A late reply to a poll for master may be rejected causing the replying node to be dropped temporarily from the token ring.

7. CONCLUSIONS

A formal modeling technique called communicating machines with shared variables was described and used to model the BACnet MS/TP protocol. The level of abstraction used in the model differs from the abstractions used in the draft BACnet standard but it captures the features important for analyzing the medium access control portion of the protocol. Using the formal model it was possible to prove safety and liveness properties for both the Master-Slave and Token-Passing modes of operation for the MS/TP protocol. In some cases the proofs rely on an assumption that errors are a rare occurrence, a reasonable assumption for a practical protocol system. In another case the proof required making an assumption about how long timers in the protocol devices can remain synchronized. This and other timing considerations were discussed and some rough guidelines for determining time-out values were given.

REFERENCES

[1] ASHRAE Standard 135P, "BACnet: A Data Communication Protocol for Building Automation and Control Networks, First Public Review Draft", August 1991, American Society of Heating, Refrigerating, and Air-Conditioning Engineers, Inc., 1791 Tullie Circle NE, Atlanta, GA 30329.

[2] Bushby, S. T., and Newman, H. M. "Standardizing EMCS Communication Protocols". ASHRAE Journal Vol. 31 No. 1. p. 33-36. January 1989.

[3] Bushby, S. T., and Newman, H. M. "BACnet - The Communication Protocol for Building Automation Systems". ASHRAE Journal Vol. 33 No. 4. p. 14-21. April 1991.

[4] Fisher, D. M. "ASHRAE Protocol: A Route to Building System Compatibility". HVAC Technology. June 1990.

[5] EIA Standard 485, "Standard for Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems". April 1983. Electronic Industries Association, 2001 Eye St. N. W., Washington, D. C. 20006.

[6] Miller, R. E. and Lundy G. M. "An Approach to Modeling Communication Protocols Using Finite State Machines and Shared Variables". IEEE Global Telecommunications Conference. December 1-4, 1986.

[7] Lundy G. M. and Miller, R.E. "Analyzing a CMSA/CD Protocol through a System of Communicating Machines Specification". CESDIS Technical Report TR-90-01, CESDIS, NASA Goddard Space Filght Center, Greenbelt, MD. January 1990.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4777

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
APRIL 1992

4. TITLE AND SUBTITLE

A Formal Analysis of the BACnet MS/TP Medium Access Control Protocol

5. AUTHOR(S)

Steven T. Bushby

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

U.S. Department of Energy
1000 Independence Ave., SW
Washington, DC 20585

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

BACnet, a draft standard communication protocol for building automation and control systems, contains options for physical and data link layer protocols. One option is to use an EIA-485 physical layer combined with a Master-Slave/Token Passing (MS/TP) media access control protocol which was specifically designed for BACnet. This paper presents a formal model of the MS/TP protocol using the technique of communicating machines with shared variables. Using this model, the protocol is analyzed and shown to be deadlock free. It is also shown that if a controller has a message to send it will eventually be transmitted.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

BACnet; building automation; CMSV; control; formal model; protocol

13. AVAILABILITY

[x] UNLIMITED

FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
25

15. PRICE
A02

ELECTRONIC FORM