NASA Lewis Research Center

Description of a MIL-STD-1553B Data Bus Ada Driver for the LeRC EPS Testbed

Michael A. Mackin

Overview

Abstract

This document describes the software designed to provide communication between control computers in the NASA Lewis Research Center Electrical Power System Testbed using MIL-STD-1553B. The software drivers are coded in the Ada programming language and were developed on a MSDOS-based computer workstation.

The Electrical Power System (EPS) Testbed is a reduced-scale prototype space station electrical power system. The power system manages and distributes electrical power from the sources (batteries or photovoltaic arrays) to the end-user loads. The electrical system primary operates at 120 volts DC, and the secondary system operates at 28 volts DC.

The devices which direct the flow of electrical power are controlled by a network of six control computers. Data and control messages are passed between the computers using the MIL-STD-1553B network. One of the computers, the Power Management Controller (PMC), controls the primary power distribution and another, the Load Management Controller (LMC), controls the secondary power distribution. Each of these computers communicates with two other computers which act as subsidiary controllers. These subsidiary controllers are, in turn, connected to the devices which directly control the flow of electrical power.

Description of Operating Environment

The Ada driver for network communication was integrated into a pre-existing hierarchical control system. The control system is responsible for data acquisition, data filtering, status monitoring, and for periodically updating an operator display screen. The system is also responsible for delivery of operator commands (*e.g.* setpoint changes) to the power devices. All communication on the control networks (computer-to-computer and computer-to-device) is done using the MIL-STD-1553B protocol.

A more detailed description of the EPS testbed and its operation is given in [1]. The control scheme and its software implementation are described in [2] and [3].

General Design Constraints

The existing Ada control program that is currently is use in the EPS testbed operates by routing *messages* among a collection of software *objects*. Each computer is configured with a different mix of executing objects. Objects operate by interpreting and processing the instructions contained within a received

message. Messages which are destined for remote objects are delivered, via an inter-processor communication subsystem, to the control program executing on another computer.

A high-level module diagram of the existing EPS control software is shown in figure 1. Objects which control the power devices are contained within the *Device Control Subsytem*. Objects which display status or buffer user commands are in the *Command Acquisition & Status Subsystem*. Code for communication between control computers resides in the *Inter-Processor Communication Subsystem*. The *Synchronous Data Acquisition Subsystem* acquires and stores data that is periodically obtained from the testbed. The *Messaging Subsystem* ties the other subsystems together and ensures the reliable delivery of control messages.

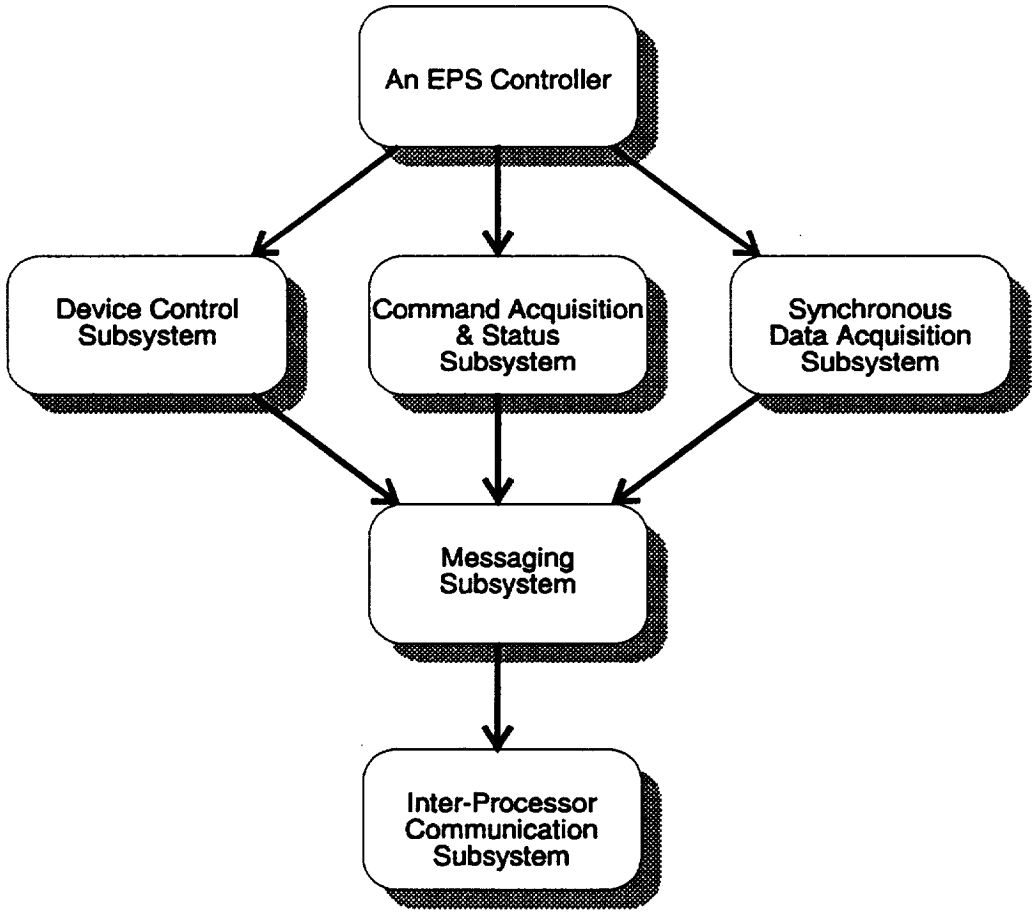

Figure 1 - EPS Controller Module Diagram

Modifying the existing EPS software so that the MIL-STD-1553B protocol is used entails changing the program's Inter-Processor Communication subsystem and incorporating a new 1553 driver routine. Since the existing program was designed for use with a peer-to-peer communication network, and the 1553B uses a master-slave protocol instead, changes were made to the program's Messaging subsystem as well.

Software Compatibility Requirements

The following requirements are necessary to minimize effects of the new network driver on the existing Ada software.

- *Packet Size* - The 1553 driver must be capable of sending a continuous block of data up to 1800 bytes long. This was the maximum block size used with the previous network driver and is maintained in the new driver to ensure compatibility.
- *Message Routing* - The new control program must maintain the capability of inter-processor message routing. Previously, the control computers were able to broadcast messages to remote objects without

specifying the node on which they resided. The 1553 protocol can transmit broadcast commands, but there is no explicit means for the bus controller to determine if each remote terminal is ready to receive a broadcast message at the time it is issued. If a remote terminal is not ready, then an error will occur and the message will have to be re-transmitted.

An alternative to broadcasting messages is address lookup and routing. In this scheme the bus controller knows the network address of each software object and directs messages to the node on which the object resides. Messages originating at the remote terminals may be passed only to the bus controller.

Messages must flow from the bus controller to the remote terminals, and from the remote terminals to the bus controller. There is no need for the remote terminals to communicate between each other.

Other Software Requirements

- *Timing* - The 1553 driver design must allow each remote terminal to transmit a message to the bus controller at least once per second. The bus controller must be capable of obtaining at least one message from each of its remote terminals in one second.
- *Network Addresses* - It must be possible to assign each computer on a 1553 network a unique network address. The PMC and LMC must be bus controllers.
- *Development Environment* - The Ada language must be used to generate the new 1553 driver.

Hardware Requirements

- *1553 Hardware* - The control program operates on a Compaq 386 computer and uses a DDC BUS-65515 1553 card for the inter-processor communication. The DDC card is a memory-mapped card that uses an 8 Kbyte area of memory space.

1553B Data Bus Operation

This section gives a brief overview of the 1553 protocol, highlighting those features that pertain to the design of the testbed control program.

Bus Controller

The 1553B network communication protocol is primarily master-slave. A master-slave network protocol operates by having a single node, the *master,* initiate every communication transaction. The other nodes on the network, the *slaves,* respond to the master's commands. In the 1553 standard, node masters are known as *bus controllers* and node slaves are *remote terminals* [4].

The computer designated as 1553 bus controller is master of the 1553 bus. The 1553 standard allows the bus controller to communicate with up to 30 remote terminals using network addresses 1 - 30. Network address 31 is reserved for broadcast messages to all remote terminals. In the testbed, the bus controller issues just three types of commands: controller-to-RT transfers, RT-to-controller transfers, and mode commands. The remote terminals respond with a status word to each of these commands. The status word indicates whether the commands were received correctly. There may be only one active bus controller.

In a controller-to-RT command or a RT-to-controller command the bus controller specifies a network subaddress and a data word count. The subaddress is in the range 1 - 30. The word count range is 1 - 32. No more than 32, 16-bit words may be transmitted with each 1553 command.

Mode commands are used to issue special commands to the remote terminals. Mode commands may be broadcast to all the remote terminals, or directed to a particular one.

Remote Terminal

A remote terminal responds to commands issued by the bus controller. The response normally consist of a stream of 0 - 32 data words and a status word, which indicates the success/failure of the transfer.

The maximum memory available for data transfer from the remote terminal is

(30 subaddresses) x (32 words/subaddress) x (2 bytes/word) = 1920 bytes.

This memory area will be known in this document as the remote terminal's *message space.* The message space is hardware mapped to a free segment of local memory.

The controller-to-RT command and the RT-to-controller command provide the bus controller with the capability of writing or reading any word of memory within the remote terminal's message space. The bus controller accesses a particular remote memory location with a combination of random and sequential access techniques. The bus controller uses the 1553 subaddress to select one of 30 records of data. Each record contains 32 data words. The bus controller sequentially reads/writes each word within the record until it obtains/updates its target word.

Operating Concept

This section describes the general operation of the 1553 driver. Refer to figure 2 for a diagram of the various software operating modes. The modes are described in more detail below.

Program Startup

When execution of the testbed control program is started, the 1553 network is inactive and waiting to be placed into either bus controller (BC) or remote terminal (RT) operating mode.

The network driver software may be configured to operate in either bus controller or remote terminal mode. Once the mode has been chosen it may not be changed unless the control program is restarted. Only one controller on a 1553 network may be chosen to be bus controller. Bubble 1 of figure 1 represents the uninitialized state of the 1553 driver. No network transactions may occur until the driver is initialized. Initialization causes the driver to enter either BC Initialization state (bubble 2) or RT Initialization state (bubble 4). A user command is used to specify the operating mode as well as a network address and node identifier (if needed).

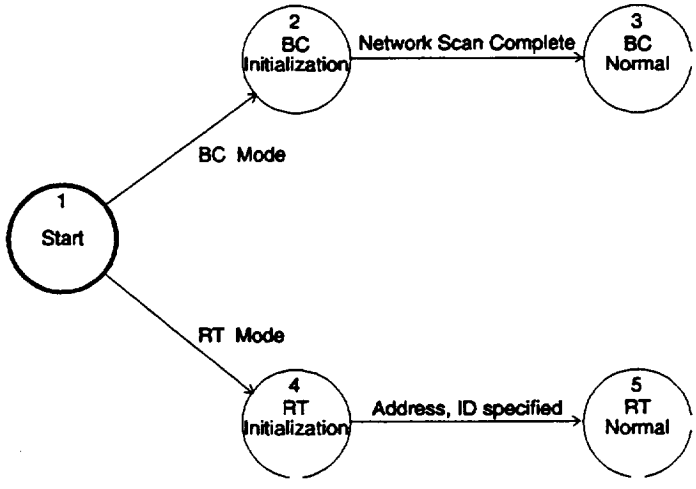

Figure 2 - 1553 driver operating modes

RT Initialization

A network address and node identifier are provide by the user when the driver is initialized to remote terminal mode. The network address specifies the RT address to which the node will respond (1-30). The node identifier is saved for use by the network's bus controller and used for message routing. The software enters *RT Normal* mode after initialization has completed.

BC Initialization

When the controller is started as a network bus controller it scans through all 1553 addresses looking for active RTs. For this reason the *bus controller is not initialized until after all active remote terminals have been initialized.* Active RTs will respond to the scan with a node identifier. The node identifiers are saved for use in routing future messages to the correct node. The software enters *BC Normal* mode after initialization has completed.

Normal BC Operation

The operation of the bus control software under normal conditions consists of

- Periodically scanning RTs for new messages.
- Reading new messages from RTs.
- Delivering messages to RTs.

The message scans are performed at a rate which is fast enough to receive at least one message from each active RT per second (see *General Design Constraints*).

Before the start of each scan, the BC issues a global system synchronization signal which initiates the cyclic data acquisition process on the RTs (as described in [3]).

Normal RT Operation

The operation of the remote terminals is similar to that of the bus controller. Periodically, the RTs will

- Store any message destined for the bus controller in 1553 message space (if space is available).
- Check for a message from the BC.

Message Delivery and Routing

Local message routing is performed automatically by the control program's Messaging subsystem. Delivery of remote messages (those destined for objects located on a different computer than the originating object) is performed via a *routing table* resident on the bus controller.

The routing table is a list of entries containing the names of remote objects together with the RT address of the node on which they reside. The driver software on the bus controller automatically scans the 1553 network and installs entries containing the node ID and address of each RT on the network into the routing table. This activity is done within bubble 2 of figure 2. Additional routing information may be entered into the table via operator commands.

On a bus controller, messages received by the testbed's control program are examined to determine the location of their destination object. Messages destined for local objects are delivered via task rendezvous. Addresses of messages destined for non-resident objects are determined by using the routing table. If no entry for a non-resident object is found, then the message is discarded.

On a remote terminal, any message destined for a non-resident object is delivered to the bus controller. If the object does not reside on the bus controller, then the message is discarded. Local messages are delivered in the usual way.

Preliminary Design

This section describes the incorporation of a new 1553 driver into the high-level design of the existing Ada control code.

Subsystem Decomposition

A refinement of a portion of the software design, first introduced in figure 1, is given in the module diagram of figure 3. This diagram shows the key components of the software's Messaging and Inter-Processors Communication subsystems. These subsystems are the ones affected by changes to the system's communication network. The design shown in figure 3 was originally designed for a system using a peer-to-peer communication protocol.

Arrows are used in the diagram to indicate compilation order. An arrow is directed from higher-level modules towards the lower-level modules upon which they depend. An Ada package is represented by a large rectangle with smaller oval bubbles and thin rectangles protruding from it. Ada tasks are shown by shaded diamonds with thin rectangles. The thin rectangles represent public procedures or entry points, and the ovals represent public type specifications.

The Messaging subsystem contains the software objects, as well as the *Message_Package* module, which performs all inter-object communication. The objects serve as an interface between the higher-level software subsystems and the Messaging subsystem. The Message_Package uses procedures provided by the *Network_Package* to deliver messages to objects residing on remote controllers.

One object shown within the Messaging subsystem of figure 3 is responsible for directing and controlling message routing. This object, the *Router*, must operate differently for a master-slave network than for a peer-to-peer one. For a master-slave network the router must know the address of each remote object. It may not broadcast messages to unspecified addresses as it may for a peer-to-peer network. For this reason, an additional module, the *Network_Utilities_Package*, has been added to the system to maintain a remote messaging capability when a master-slave protocol is used (see figure 4).

In figure 4, new procedure interfaces are indicated by thicker lines. Note that the most commonly used interfaces (between the objects and the Message_Package) did not change. New type specifications, as indicated by a dark *type oval*, and additional procedures, as indicated by a dark *procedure rectangle*, do require the recompilation of both the Messaging and Inter-Processor Communication subsystems.

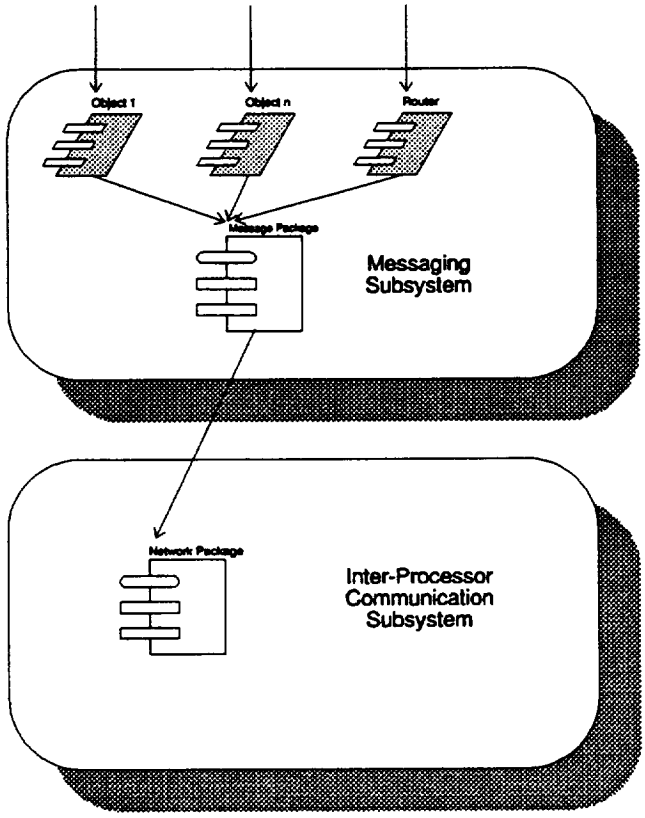

Figure 3 - Subsystem Decomposition for Peer-to-Peer Network

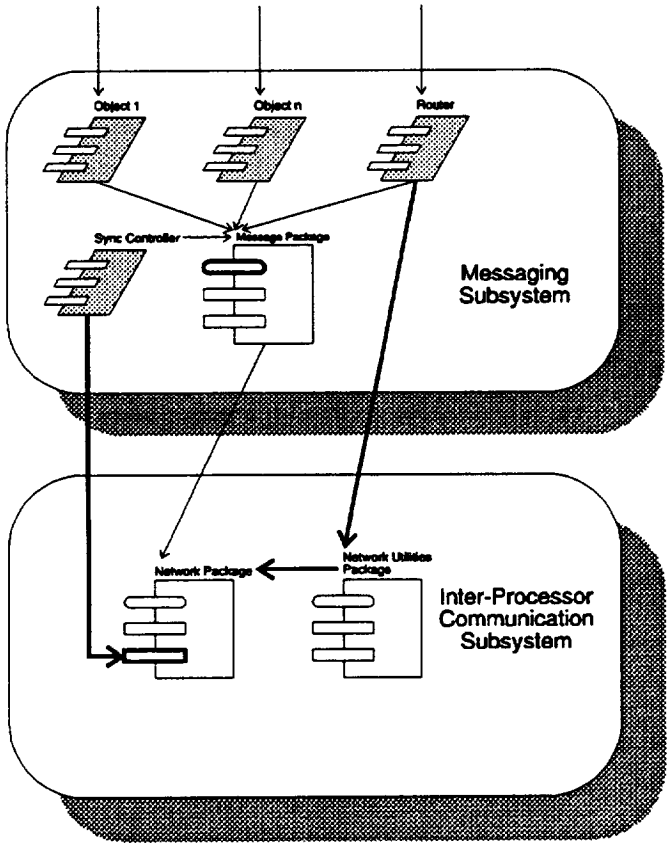

Figure 4 - Subsystem Decomposition for Master-Slave Network

Figure 4 also shows the addition of a new object, the *Sync_Controller*. The Sync_Controller manages the execution of those objects responsible for periodic data acquisition. New procedures have been added to Network_Package to provide information from the master bus controller about whether or not periodic data acquisition should occur.

The remaining part of this section describes the new software components, as well as the changes to old components, that are shown in figure 4. The components described are: the Network Driver (the Network_Package), the Network Utilities (the Network_Utilities_Package), the Message Package, the Router, and the Sync_Controller.

Network Driver

The 1553 network driver provides the routines that the control program uses to manage message transactions over the network.

General Routines

The general purpose routines are

- *Initialize* — Initialize the network. Set the local network card to either BC or RT mode. For RT mode, the user provides a network address and a node identifier. For BC mode, the software will scan the network and find any initialized RTs. The identifiers from these RTs are stored and may be retrieved via the *Network_IDs* routine.
- *Network_IDs* — Retrieve a table containing network identifiers of all active RTs attached to this bus controller.
- *Get* — Obtain data packets from the network. Wait until a packet destined for this local node arrives and then return with it. Only one task may access this procedure at a time (i.e. concurrent access is NOT allowed).
- *Read* — Extract data from a packet. Obtain data from a previously received packet.
- *Write* — Store data in a packet. Save a data item into a packet for use in a future transmission.
- *Transmit* — Transmit a packet. Transmit a data packet over the network to a particular RT, or from a RT to the BC. Several tasks may access this procedure concurrently.

Synchronization Routines

These are routines to initiate and control the low level data scans on the RTs

- *Synchronize* — The bus controller notifies all RTs to operate synchronously.
- *Unsynchronize* — The bus controller notifies all RTs to operate asynchronously
- *Get_Sync_Mode* — Obtain the operating mode of RTs.

Status Routines

These routines provide board status information

- *Local_Address* — Obtain the local 1553 address.
- *Network_Started* — Obtain 1553 board status (initialized/not initialized).

1553 Monitor

The 1553 Network Driver contains a monitoring task which prevents concurrent access of the 1553 board's hardware registers and memory by more than one Ada task.

Messages to be transmitted to remote nodes are routed through the 1553 monitor and are handled in a first-in-first-out manner.

The 1553 Monitor stores messages received from remote nodes in a temporary buffer. The contents of the buffer are acquired from the monitor by invoking the *get* procedure. Only one task is permitted to receive messages from the 1553 monitor. This task is responsible for distributing the message to the correct local object.

The 1553 monitor is responsible for performing the periodic operations described in section *Normal BC Operation* and in section *Normal RT Operation.*

Module Diagram

A pictorial representation of the Network Driver package, showing its public procedures and internal task, is shown in figure 5.

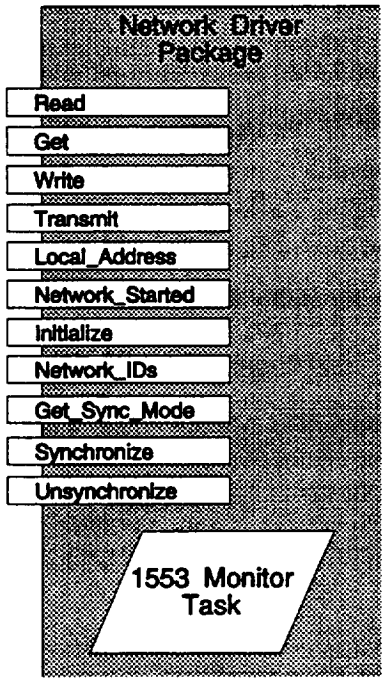

Figure 5 - Network Driver Package

Exceptions

Problems that may occur when the package routines are called include

- *Initialize* — Raise *Already_Initialized* exception to indicate that the board has already been initialized.
- *Network_IDs* — Raise *Usage_Error* if board has not been initialized to BC mode.
- *Get* — An indefinite delay will occur if the 1553 board malfunctions.
- *Read* — Raise *Underflow* if an attempt is made to read past the end of the available data.
- *Write* — Raise *Overflow* if data is stored past the allowable maximum size.

- *Transmit* — Raise *Transmit_Error* if an inactive address has been specified as the destination address, or if an RT is attempting to transmit to another RT.
- *Synchronize* — Raise *Usage_Error* if the local board is not operating in bus controller mode.
- *Unsynchronize* — Raise *Usage_Error* if the local board is not operating in bus controller mode.
- *Get_Sync_Mode* — Nothing should go wrong.
- *Local_Address* — Nothing should go wrong.
- *Network_Started* — Nothing should go wrong.

Network Utilities

The network utilities package provides routines that are used primarily for message routing. The package includes procedure that operate on an internal routing table. The routing table consists of a list of entries indexed by object identifiers. Each entry contains the text identifier and the network address where the object resides.

Routines

- *Initialize* — Used by the bus controller to obtain and initialize the table of active RTs.
- *Show* — Display the routing table on the operator's screen.
- *Add_Alias* — Add an alias to an existing entry. Allows a node to be identified by more than one name.
- *Add_Item* — Add an item to the routing table.
- *Address_Of* — Obtain the node address of a particular item.

Semaphore Task

The Network Utilities Package contains an Ada task to guarantee sequential access to the remote routing table. Sequential access is necessary since attempts to access the table may occur concurrently by more than one task. Most tasks access the table while sending a remote message, but the table may also be accessed, to be updated or changed through operator commands, by the Router Task. The task used to provide sequential access and mutual exclusion is called the Semaphore Task.

Module Diagram

A pictorial representation of the Network Utilities package is shown in figure 6.

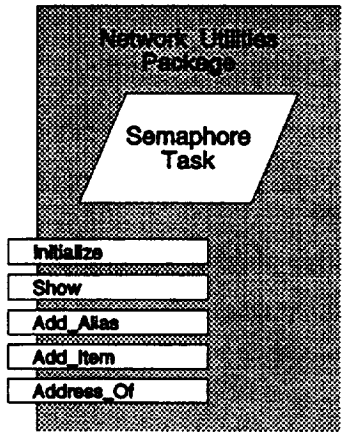

Figure 6 - Network Utilities Package

Exceptions

Some Network Utility procedures may generate Ada code exceptions. The exceptions and the conditions that cause them are identified below.

- *Initialize* — Assuming that only a node setup as an BC calls this routine, nothing should go wrong.
- *Show* — Nothing should go wrong.
- *Add_Alias* — Raises *Lookup_Error* if the item that requires an alias is not already present in the routing table.
- *Add_Item* — Nothing should go wrong.
- *Address_Of* — Raises *Lookup_Error* if the item is not present in the routing table.

Modifications to Message Package

The control program's Message Package is responsible for directing messages between active software objects. It must access the 1553 network to deliver messages to those objects which are executing on remote nodes.

Few changes were made to the public interface defined between the Message_Package and the other software components in the testbed control program. The biggest change was made to a field within the public *Message* structure.

The Message structure contains a field which was defined to hold the node address of the controller where the message originated. This field was, in previous versions of the control program, initialized with the value *null*. The control program interpreted this null value to indicate that a broadcast was to occur if no local object could be found. However, since broadcasting is not allowed in the 1553 version of the control program, the null initialization was eliminated. Instead, the field is assigned the 1553 address of the local node.

Router Task Modifications

The Router Task is an existing element of the testbed Control Program. Modifications enable the task to process commands which initialize, control, and display the status of the 1553 driver.

1553 Commands

The following commands were added to the existing Router object.

- START NETWORK, *mode*

  mode = BC | RT, *rt_address*, *rt_id*
  rt_address = 1|2|3| ... |30
  rt_id = Any six byte or less node identifier.

  The "START NETWORK" command is used by the operator to initialize the 1553 network. The command "START NETWORK, BC" notifies the software that the board will be operating in BC mode. For operation of the software in RT mode at address 1 and with node identifier "MBC" use: "START NETWORK, RT, 1, MBC". The "START NETWORK" command may be issued only once. **All RTs on the system must be initialized before the BC is started.** The RTs may be initialized automatically from the DOS command line if the control program is invoked in the following manner:

  CONTROL_ RTR, START NETWORK,RT,1,MBC

  (Note: "CONTROL_" is the DOS file name of the executable Ada control program)

- SHOW REMOTES

  This command causes the router to invoke the Network_Utilities package and display the current routing table.

- REMOTE ALIAS,*existing_name,alias*

  existing_name = text string of object preexisting in the routing table.
  alias = new text string to be associated with the existing object.

  Use the "SHOW REMOTES" command to see a list of existing remote objects.

- SYNCHRONIZE

  Causes all RTs to begin synchronous data collection.

- UNSYNCHRONIZE

  Causes RTs to cease synchronous data collection.

Exceptions

These are the exceptions that may be raised by the provided router commands. The exception handler within the Router task body handles the exceptions and displays an error message.

- START NETWORK Network board is inactive or not operating.
- SHOW REMOTES Nothing should go wrong.
- REMOTE ALIAS Name to be aliased does not exist.
- SYNCHRONIZE Local node is operating as an RT instead of a BC.
- UNSYNCHRONIZE Local node is operating as an RT instead of a BC.

Sync Controller

The Sync Controller is a task which is responsible for the execution and control of those tasks which perform the periodic data acquisition. It is active on remote terminal nodes only. The Sync Controller issues commands to the periodic acquisition tasks based upon the operating state obtained from the Network Driver routine *Get_Sync_State*. *Get_Sync_State* returns a value which indicates whether the remote terminal is to obtain data in a synchronous or an asynchronous manner.

Preliminary Design Module Diagram

If the procedures defined in the sections above are used, a more detailed module diagram than that shown in figure 4 is possible. The more detailed high-level module diagram is presented in figure 7. Arrows show how the package procedures are invoked. The procedure *transmit*, for example, is called from some procedure within Message_Package. Procedure *get* is called by the Network_Monitor_Task, which is also within Message_Package.

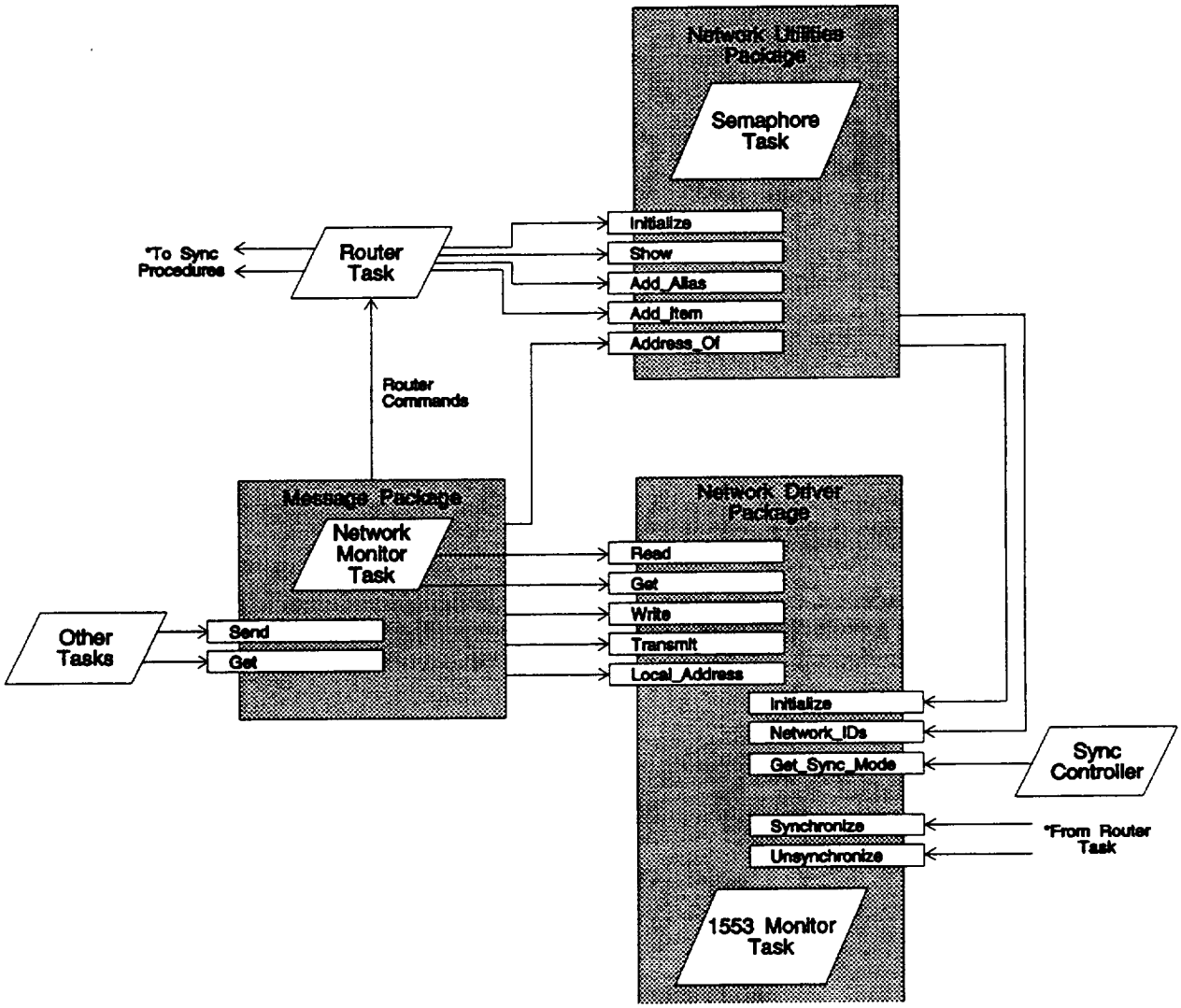

Figure 7 - Preliminary Design Module Diagram

Detailed Design

This section presents the internal design of the software modules previously identified in section *Preliminary Design* as well as the design of any new packages necessary for the implementation of the detailed design.

Subsystem Decomposition

In the EPS testbed software, detailed design features are implemented in Ada package bodies. Ada bodies are represented in a module diagram by a shaded icon shown partially obscured beneath the package specification icon. In figure 8, new packages have been introduced into the module diagram shown in figure 4.

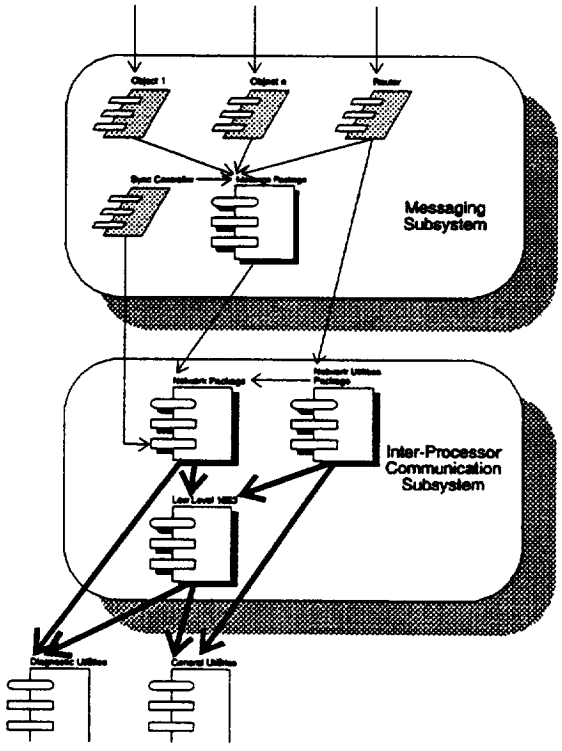

Figure 8 - Detailed Design Module Diagram

New modules in the diagram are: the Low_Level_1553 package, the Diagnostic_Utilities package, and the General_Utilities package. The Low_Level_1553 package provides routines for composing, transmitting, and receiving 1553 messages, the Diagnostic_Utilities package provides routines that are useful for obtaining timing and diagnostic information from a logic analyzer, and the General_Utilities icon represents a number of Ada utility packages which provide routines for sorting and indexing tables of data. The Diagnostic_Utilities and General_Utilities packages are not part of the Inter-Processor Communication subsystem, per se, but may be used by any module within the software system.

Figure 8 also shows, in addition to the new packages, the package bodies for the Message_Package, the Network_Package, the Network_Utilities package, and for the Low_Level_1553 package.

The compilation dependencies among the new packages and the package bodies are indicated in the diagram by the darker arrows.

Network Driver

This section describes the internal design of the 1553 network driver. The Network Driver makes use of a separate package of routines, the *1553 Low Level Driver*, to manipulate the buffers and memory locations associated with the 1553 board (as shown in figure 8).

Allocation of 1553 Resources

The network driver must allocate and manage the 1553 traffic so that the control program messages are reliably delivered among the control computers. Since the 1553 protocol provides a limited number of subaddress locations, they must be carefully allocated and assigned.

The 1553 driver allocates the RT subaddresses to those functions specified in table 1. Note that subaddress 2 is used for two separate functions.

Description	Subaddress	Data Words
Semaphore for BC-To-RT Messages	1	1
Semaphore for RT-To-BC Messages	2	1
Node Identifier	2	2 - 5
Data Packet (Block 1)	3	1 - 32
Data Packet (Block 2)	4	1 - 32
...	...	1 - 32
Data Packet (Block 28)	30	1 - 32

Table 1 - Subaddress Assignment.

Client/Server Semaphores

Client/Server semaphores are used to insure the reliable delivery of long packets of data between the BC and RTs, to regulate the data transfer rate, and to prevent buffer overflow. A server is a computer that is attempting to deliver a message to a remote node. The client is the receiving computer. Basically, the semaphores ensure that clients do not process messages before they have been completely delivered, and that servers don't deliver new messages until old ones have been processed.

Two subaddresses on each RT are reserved for use by these semaphores. Each controls the data flow in a single direction. See table 2 for the subaddress assignments.

Message Direction	Client Controller	Server Controller	RT Subaddress
BC-To-RT	RT	BC	1
RT-To-BC	BC	RT	2

Table 2 - Client/Server Semaphores

These subaddresses are known as the *buffer status semaphores*. The semaphores are used to indicate the current status of the RT's transmit and receive buffers and to prevent the bus controller from overwriting or reading a buffer that is not yet ready. One semaphore regulates the buffer used for messages transmitted from the BC to the RT, and the second semaphore regulates messages destined for the BC from the RT. Notice that the controller at the message source is the server and that the controller at the message destination is the client.

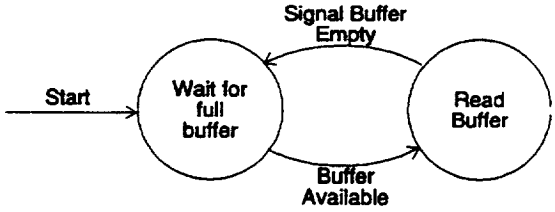

Figure 10 - Client Operation

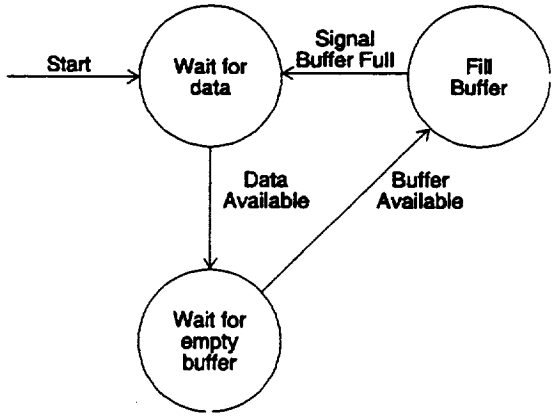

Figure 9 - Server Operation

Client operation is shown in figure 10, and server operation in figure 9.

Figure 10 shows how a client controller operates while it is waiting to receive a message for processing. Initially the controller is idle; it is waiting for a change in the buffer status semaphore. Once the semaphore changes to indicate that a buffer of data is available, it is read and processed. After processing the data, the status semaphore is reset by the client, indicating an empty buffer.

Figure 9 shows the server state transitions that occur when data becomes available for transmission. If the data is available and an empty buffer is available, then the server fills the buffer with data and sets the buffer status semaphore. New data may again be copied to the buffer after the client has read the data and reset the semaphore.

Network Package Detailed Design

1553 Monitor - BC Mode

The 1553 monitor is an Ada task which controls access to the 1553 board and performs the periodic operations necessary to maintain network message flow. Figure 11 shows the typical operation of the monitor task over time.

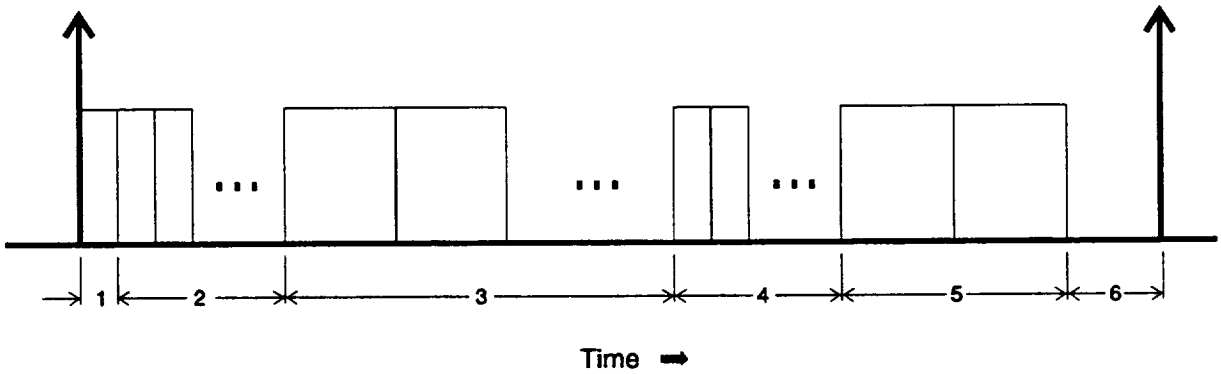

Figure 11 - BC cycle

The large upright arrows mark the start and end points of a typical cycle. The cycle repeats continuously, and is divided into six regions, as indicated on the figure. The operations performed in each region are

1. Broadcast synchronization command - At the start of each cycle, the bus controller broadcasts a 1553 synchronization command. This signal is transmitted to each RT simultaneously, indicating the start of a new BC cycle. The BC uses a standard 1553 mode command (with data) as the synchronization signal. The data word indicates whether the system is operating in a synchronous or an asynchronous manner. No buffer semaphores or other handshaking are required with the mode command since only a single data word is transmitted. The time to perform the cycle synchronization is approximately 60 Osec. This time estimate is based upon the 1553 bus speed only and contains no allowance for software overhead. The time required to broadcast the synchronization command is relatively insignificant compared to the other operations which the 1553 Monitor performs, and may be effectively neglected.

2. Scan for buffer status - Obtain the status of the client/server semaphores on each RT. Use the subaddresses specified in table 2. Query each active RT for the current value of the semaphores. There are actually two command/response transactions for each RT query. One for each semaphore value. The time to complete a query of all RTs is

   (2)(88)*k* Osec = 176 Osec

   where *k* is the number of active RTs, and the 88 Osec constant is obtained from the 1553 standard (88 = 68 + 20N, where N=1 [the number of data words requested]). This is the absolute minimum value that can be achieved and includes no software overhead.

3. Obtain data from RTs - Obtain a data packet from each RT that has new data (as indicated by the value of the semaphore obtained in step 2). Read 1 - 28 data blocks of 32 words using the 1553 RT-to-BC command. Each RT uses

   (28)(68 + 20 N) Osec

   per message, so if N, the number of data words, is 32, then the total time to complete region 3 would be 19824*k* Osec, or 19.8*k* msec., where *k* is the number of RT-to-BC transfers. This analysis assumes that there is a 0 Osec inter-message gap.

4. Update semaphore values - The BC updates the semaphores for every RT from which a message was received. This is accomplished in

   $88k$ Osec

5. Deliver message to RTs - Perform requested BC-to-RT transfers if RT buffers are available. Also, set the RTs semaphore that indicates that new data has arrived. The maximum time to completely deliver all messages would be

   $(19.8 + .88)k$ msec. $= 20.58k$ msec.

6. Idle time - The 1553 Monitor waits for the start of a new cycle.

Assuming that a network has the maximum number of devices attached ($k = 29$), and that every RT has data to be read, then the minimum possible cycle time for region 1 through 4 is

$t_{max} = (29)(176 + 20580 + 88$ Osec)

$t_{max} \cong (29)(20.0$ msec)

$t_{max} \cong 580$ msec

If each data packet contains the latest data scan from the RT, then the 580 msec cycle is sufficient to meet the control program's one second data acquisition requirement (see *General Design Constraints*). However, if just a single packet in the data stream contains an aperiodic message (e.g. a caution/warning message), then the delay between complete data updates increases to 1.6 second, which fails to meet the one second specification. A cycle time of less that 500 msec is necessary to adequately handle the periodic data scans from the RTs, the aperiodic message traffic from the RTs, and commands issued from the BC.

The best solution for reducing the cycle time of the 1553 BC to less than 500 msec is to limit the number of RTs allowed on the bus. If the number of RTs is reduced to 10, then the minimum cycle time becomes

$t_{max} = (10)(20.0$ msec) $= 200$ msec

Allowing an increase in the cycle time from 200 msec to 250 msec introduces enough idle time into the cycle for the BC to perform two BC-to-RT transfers.

1553 Monitor - RT Mode

A 1553 Monitor task operates on the remote terminal node also. Its operating cycle is shown in figure 12.

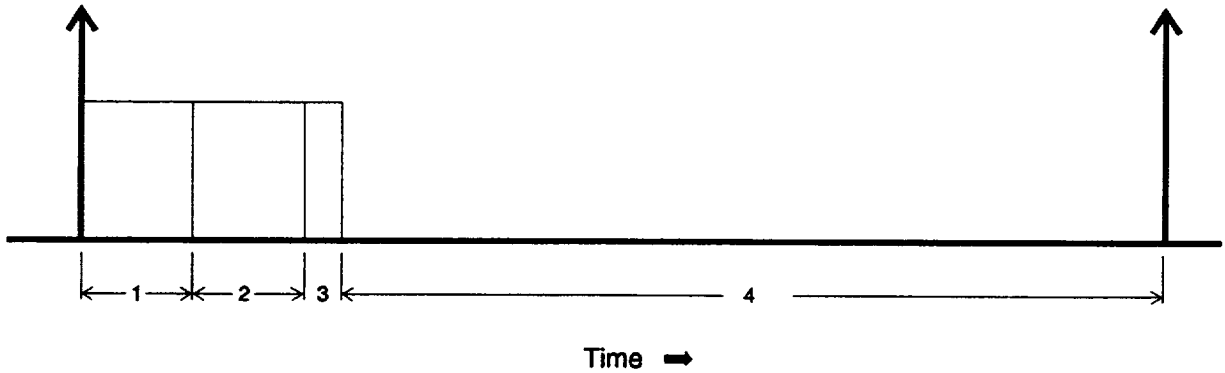

Figure 12 - RT cycle

Cyclic operation of the 1553 Monitor in RT mode is simpler than it is in BC mode. The figure shows that there are only 4 unique regions in the cycle. The beginning of the RT cycle is currently NOT synchronized with the beginning of any other controller's cycle (i.e. the RT cycle does not necessarily start at the same time as the BC cycle, or at the same time as any of the other RTs). The 1553 Monitor, in RT mode, performs the following tasks.

1. Copy a message destined for the BC into the RT buffers - If the buffer used for transferring data from the RT to the BC is empty and new data is available for transfer, then copy the data to the correct RT buffers and then set the semaphore notifying the BC that new data is available.
2. Copy a message received from the BC into a temporary buffer - If the BC has set the semaphore that indicates that a new message has been transferred to this RT, then save the message into a local buffer and reset the semaphore.
3. Check for a change in synchronization state - Check the value of the data word sent from the BC in the most recent synchronization command. If its status has changed, then save the new value.
4. Idle time - Wait for the start of the next cycle.

Private Types

The Network Driver subaddress assignments (as specified in table 1) are part of the specification of private types. The specification also contains the data type *packet*. A packet is used as a temporary store for items to be transmitted over the network. The size of the store is determined by the number of 1553 subaddresses allocated. It currently consists of 28 data blocks (see table 1).

Internal Structures, Variables, and Constants

The 1553 network driver maintains several state variables which contain information about the current 1553 network configuration. These variables are

- *Active_Node_List* - Used when the controller is operating as a BC. The *Active_Node_List* indicates which RT addresses responded to a query for node identifiers. This variable is setup during execution of the *Initialize_Network* procedure.
- *Net_Started* - A flag indicating whether the *Initialize_Network* procedure has successfully executed. Many procedures raise exceptions if this value has not been set to *true*.
- *Packet_Request* - A block of messages which contain the sequence of 1553 commands necessary to transfer a data packet from a RT to the BC.
- *Query_RT_For_Data* and *Query_RT_For_Available_Buffer* - A block of messages that are sent out in region 2 of figure 11, requesting status of each RT buffers. The number of messages depends upon the number of nodes within the *Active_Node_List*.
- *Packet_Response* - Contains the lookup keys necessary for retrieving the data requested via the *Packet_Request* messages.
- *Packet_Size_Lookup* - A global variable that is used to contain the size of the packet currently being retrieved.
- *Output_Pipe* - A buffer where messages received from RTs are stored temporarily until they can be retrieved by a task invoking the *get* procedure.

Internal Procedures

- *Poll_Network* — Scan each 1553 address for a valid node identifier. Save the identifier and address if a valid response is obtained. Also, construct the blocks of messages which are periodically used to obtain RT buffer values.
- *RT_Query* — Restore the block of messages that query the semaphores at all active nodes (both RT-to-BC buffers and BC-to-RT buffers). Construct a list of nodes which require servicing.
- *Build_Packet_Request* — Construct the *Packet_Request* variable.

- *Process_Packet* Copy the data obtained from a RT to the internal buffer *Output_Pipe.*
- *Service_Board* This generic procedure is used to place a particular set of messages into the 1553 board, transmit the block, and then process any responses. It makes use of the controller's double buffering capabilities. The procedure returns a list of RT nodes which require further processing.
- *Poll_For_Packets* An instantiation of the *Service_Board* procedure. Uses *Build_Packet_Request* to construct a sequence of messages which obtain data packets, and uses *Process_Packet* to save the responses.
- *Confirm_Packet_Read* An instantiation of the *Service_Board* procedure. Sets up messages clearing RT semaphores and checks for any resulting errors.
- *Initialize_RT_Mode* Set the node identifier and allocate the RT's buffers between those that are shared and those that are not shared. Also sets the initial state of the semaphore flags. Finally, *Initialize_RT_Mode* clears the 1553 busy flag.
- *Setup_RT_Packet* Copy a data packet into the proper RT buffers.
- *Send_BC_Packet* Build a data packet into a sequence of 1553 messages and then start transmission.

Data Flow - BC Mode

A data flow diagram for the Network Driver operating in BC mode is show in figure 13.

Bubbles 1 - 6 represent the public processes available through the package specification. Bubbles 7 - 12 represent processes which occur within the 1553 Monitor task. The arrows indicate how the processes communicate with each other and with the public data stores (which are represented as two horizontal lines). Communication between bubbles 7 - 12 and the low level 1553 driver are not explicitly indicated.

The cyclic operation of the 1553 Monitor is initiated by the *cycle start signal*, which is provided by the Ada runtime system (when the appropriate time delay has elapsed). The monitor operates as previously discussed (and shown pictorially in figure 11). A cycle consists of: transmission of a 1553 synchronization signal (bubble 7), a query of active RT nodes (bubble 8), and the processing of messages from nodes with data (bubbles 9, 10, and 11). The transfer of messages from the BC to an RT is made near the end of each cycle. The cycle is completed when the monitor enters a quiescent state and waits for the next *cycle start* signal.

Users obtain data from the monitor via the data stores *Output_Pipe* and *Is_Synchronized.* The use of temporary stores permits the 1553 Monitor to continue operation without requiring inter-task rendezvous. Data and control are provided to the monitor through the *Synchronize, Unsychronize,* and *Transmit* processes. Bubble 11 (Transmit Packets) and bubble 12 (Change_Sync_Mode) are able to process input flows from the user procedures if they exist, but will not block if they are empty.

To maintain the requirement for cycle completion time, the flow from bubble 3 (Transmit Packets) is restricted so at most two transmission can occur per cycle.

Data Flow - RT Mode

A data flow diagram for the Network Driver operating in RT mode is shown in figure 14.

On this diagram bubbles 1 - 4 represent public processes that may be invoked when the driver is operating in RT mode. Several of the data stores reside in the memory regions contained within the 1553 board (as shown by the shaded rectangle). These regions are actually accessed through processes provided by the low level 1553 driver package, but these processes have been removed from the diagram for the sake of clarity. It is easier to understand the overall data flow without them.

Operation of the 1553 Monitor in RT mode is similar to its operation in BC mode. The monitor idles until it activated by the Ada runtime system. It then examines the semaphore, checking for the existence of a free memory buffer. If an empty buffer is indicated, a message is copied onto the board. Next, the monitor

checks for messages from the BC. If messages are present, they are copied into a temporary buffer (the *output_pipe*) and the buffer semaphore is reset. Finally, the monitor examines the most recent value of the synchronization data word. If the value has changed since the last cycle, then the new value is stored.

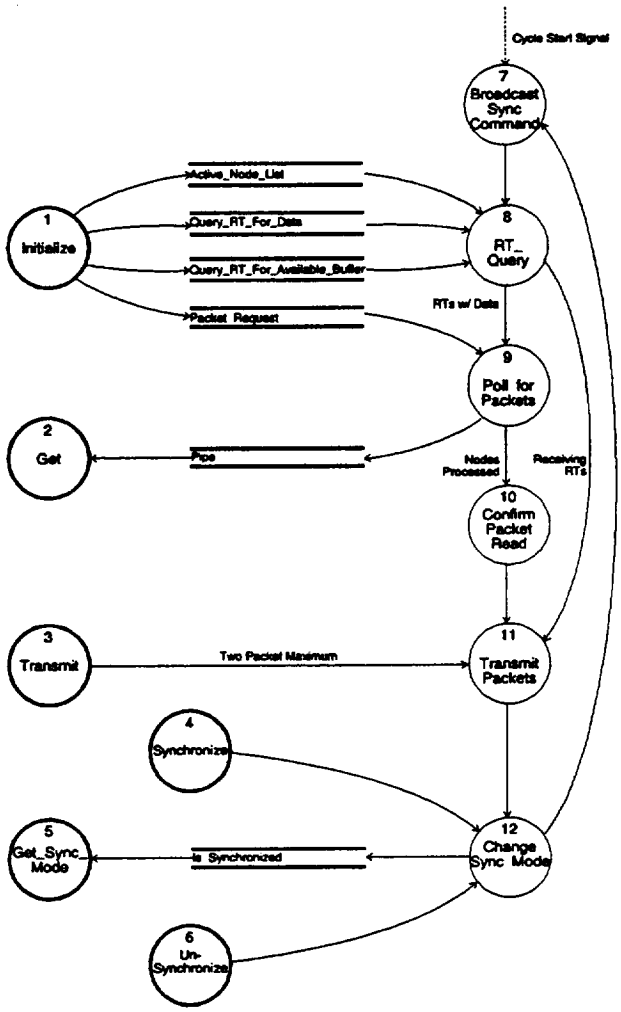

Figure 13 - Network Driver data flow diagram (BC mode)

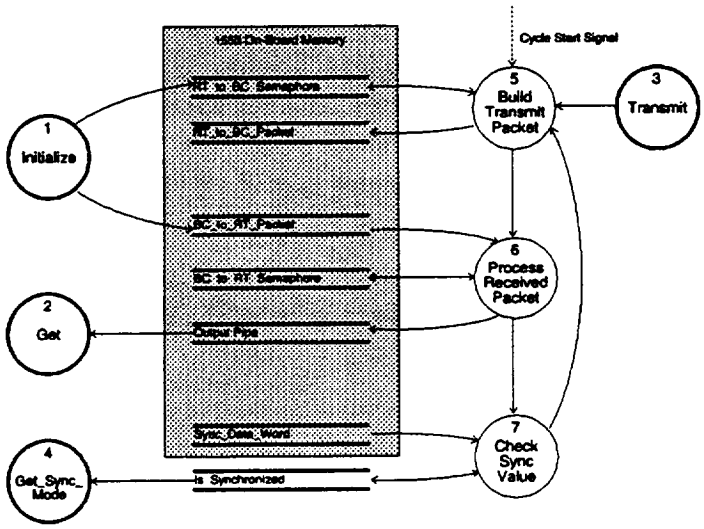

Figure 14 - Network Driver data flow diagram (RT mode)

Network Utilities

Network Utilities Package Detailed Design

This section describes the internal design features of the Network Utilities package.

Private Types

There are no private data types defined within the Nework_Utilities package.

Internal Structures, Variables, and Constants

- *The_Lookup_Table* - A table containing the node identifiers and object names of all testbed remote objects. This table is an object obtained from the instantiation of the generic Unbounded_Cached_Map package which is described in [5].
- *The_Semaphore* - An Ada object which is used to regulate access to *The_Lookup_Table*. It is an object obtained from the Semaphore package described in [5].

Internal Procedures

- *Send_Line* — Used to send a text message to the operator.
- *Hash_Task_Name* — A function used by the map package which hashes a text string to a positive number. The hash function obtains the numerical value by XORing every character in the object name string together. This is a quick way to generate a single value that is dependent on every character in the string.

Low Level 1553 Driver

The low level 1553 driver is a package of Ada routines for accessing the lowest level registers and memory locations of the 1553 controller board. The package also provides routines for composing, transmitting, and receiving 1553 messages. No mutual exclusion among users of this package is provided.

Three groups of routines are contained within the low level package: those for use with controllers initialized into BC mode, those for use with controllers initialized into RT mode, and general routines, whose use is independent of the controller mode.

General Routines

- *Set_Mode* — Set the board to operate in either BC or RT mode. If RT mode is selected then, initially, the 1553 *busy flag* is set. Use the RT procedure *Clear_Busy_Flag* to enable error-free message transactions.
- *The_Mode* — Return current operating mode of 1553 board.

BC Routines

To transmit a message from the 1553 board, when operating as a BC, use the sequence of steps identified in figure 15.

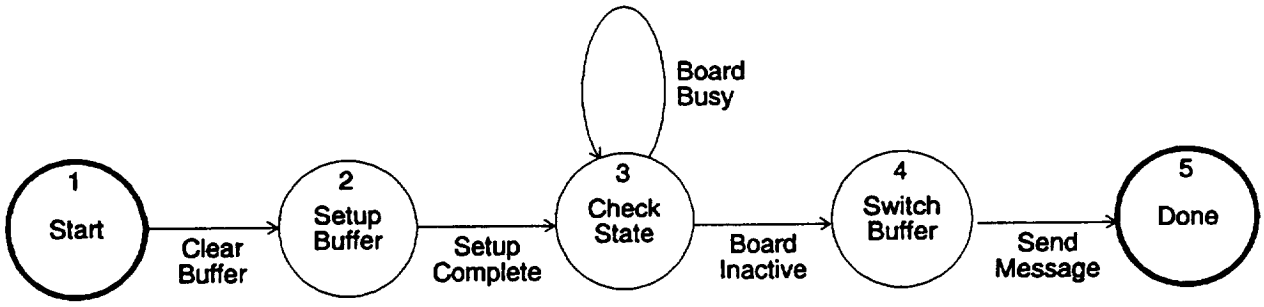

Figure 15 - Operation of board in BC mode

Operating the 1553 driver in this manner takes advantage of a double buffering scheme employed on the 1553 board which allows one buffer to be setup and configured while another one is being transmitted. To use this scheme, first clear the board's inactive buffer (as shown by the transition from bubble 1 to bubble 2). Then, store a sequence of 1553 messages into the buffer (as shown in bubble 2). These messages will be sent as a continuous "block" of 1553 messages. Now, before initiating the transmission, ensure that the board is idle and has finished transmitting any previous block of messages. It may be necessary to check the board's status several times (as shown by the loop both starting and ending in bubble 3). To send the block of messages, make the new block the active buffer (bubble 4), and start transmission.

If any of the messages were requests for a RT to transmit data back to the BC, then that data must be retrieved before a new series of new messages is configured. In this case, replace bubble 5 of figure 15 with those shown in figure 16.

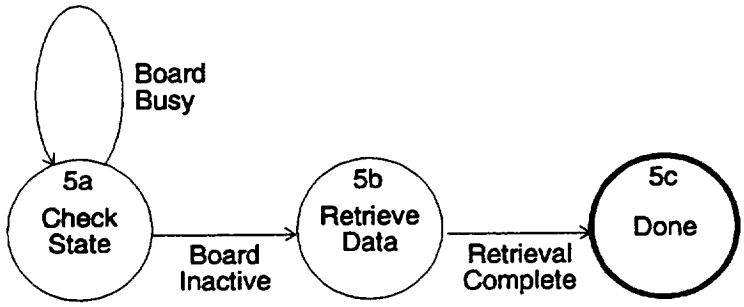

Figure 16 - BC Data Retrieval

A description of the routines necessary for control of the board in BC mode (as shown in figure 15 and figure 16) are given below.

To Clear the Inactive Buffer

- *Clear_Buffer* Clear the inactive buffer.

To Build the 1553 Message Buffer

- *Build_Receive_Data_Message*
  Construct a 1553 message with 1-32 words of data. Set the 1553 bit which indicates that the RT will be receiving data from the BC. Store the message into a 1553 buffer, but do not send the message. This procedure may be called repetitively to build a sequence of messages that will be sent over the 1553 bus.
- *Build_Receive_Data_Message*
  Same as the procedure above except optimized for transmitting a single word of data from the BC to an RT.

- *Build_Transmit_Data_Message*
  Construct a 1553 message requesting that 1-32 words of data be transmitted from an RT. Store the message into a 1553 buffer, but do not send the message. This procedure may be called repetitively to build a sequence of messages that will be sent over the 1553 bus. It may be interspersed with messages constructed using the *Build_Receive_Data_Message* procedure. This procedure returns a lookup key that is used to obtain the data that is transmitted from an RT in response to this message.

- *Build_Transmit_Data_Message*
  Same as the procedure above except that the message is not stored in a buffer on the 1553 board, but instead, to a data structure from which it may be retrieved for future use. See the description of the *Restore_Message_Block* procedure below.

- *Build_Mode_Code_Without_Data*
  Construct a 1553 message which transmits a 1553 mode code. Store the message into a 1553 buffer, but do not send the message. This procedure may be called repetitively to build a sequence of messages that will be sent over the 1553 bus. It may be interspersed with any of the other procedures which store messages into the 1553 board's buffer (*Build_Receive_Data_Message* or *Build_Transmit_Data_Message).*

- *Build_Mode_Code_With_Data*
  Same as the procedure above except that one word of data may be included with the mode code message.

To Save and Restore a Block of Messages

For some sequences of messages that are often used, it may make sense to save and restore the entire block of messages instead of repetitively building up the message block with the routines described above. Routines to save and restore message blocks are described below.

- *Save_Message_Block* Save the status of the current buffer. Must be used before the buffer has been made active.

- *Change_Destination_Address_Of*
  This routine changes the destination RT address of a saved block of messages to a specific value. This procedure is used to repetitively direct a block of messages to several different RTs. It operates on a block of messages that has been previously saved with the *Save_Message_Block* procedure.

- *Change_Block_Destination_Address*
  This procedure is the same as the one above except that it operates on the block of messages that has been built up in the controller's internal buffer.

- *Restore_Message_Block*
  Restore a sequence of messages that had been previously saved by the *Save_Message_Block* routine.

To Check the Board's Status

- *Message_Complete* Returns *true* when the 1553 board has completed processing of any previous message and is now idle.

To Send a Message

- *Switch_BC_Buffers* Notify the 1553 board that a new buffer is to be used for 1553 message transmission.

- *Send_Message_Block* Initiate the transmission of a block of 1553 messages. The 1553 controller card will continue the message transmission after the routine has completed.

To Retrieve Data Transmitted from an RT

- *Data* Used to retrieve data that was sent from an RT in response to a *Build_Transmit_Data_Message*. A lookup tag obtained from the *Build_Transmit_Data_Message* is used to obtain the correct data item.

BC Exceptions

Problems that may occur in calls to BC routines are

- *Clear_Buffer* No problems should occur.
- *Build_Receive_Data_Message* Raises *Invalid_Mode* if the controller is not in BC mode. Raises *Overflow* if the message buffer is filled.
- *Build_Transmit_Data_Message* Raises *Invalid_Mode* if the controller is not in BC mode. Raises *Overflow* if the message buffer is filled.
- *Build_Mode_Code_Without_Data* Raises *Invalid_Mode* if the controller is not in BC mode. Raises *Overflow* if the message buffer is filled.
- *Build_Mode_Code_With_Data* Raises *Invalid_Mode* if the controller is not in BC mode. Raises *Overflow* if the message buffer is filled.
- *Save_Message_Block* Raises *Invalid_Mode* if the controller is not in BC mode.
- *Change_Destination_Address_Of* Raises *Invalid_Mode* if the controller is not in BC mode.
- *Change_Block_Destination_Address* Raises *Invalid_Mode* if the controller is not in BC mode.
- *Restore_Message_Block* Raises *Invalid_Mode* if the controller is not in BC mode.
- *Message_Complete* No problems should occur.
- *Switch_BC_Buffers* Raises *Busy* if the 1553 controller has not finished processing a buffer.
- *Send_Message_Block* Raised *Invalid_Command* if buffer of commands is not formatted correctly.
- *Data* No problems should occur.

RT Routines

The 1553 card, initialized as an RT, contains an area of memory that can be accessed by both the local control computer and by the remote BC (as described in the *1553 Overview*). This memory area consists of thirty blocks of thirty-two data words. These blocks are known, in this package specification, as *RT Buffers*. Each RT buffer corresponds to a particular RT subaddress. A RT buffer may be configured so that the same area of memory is used for both receiving data from the BC and for transmitting data to the BC. A RT buffer may also be configured so that a different data area is used for receiving than for transmitting. If different data areas are used, then the same subaddress may be used to both send and receive data. Data to be sent to the BC is placed in the area designed for RT-to-BC transfers, and data received from the BC is obtained from the area designated for BC-to-RT transfers.

A buffer that is used for both RT-to-BC transfers and BC-to-RT transfers is *shared* between the two transfer types. Initially, all thirty RT buffers are shared.

Once the RT buffer configuration is set, they are read/written by the local control computer through the *Set_Data_Block* and *Get_Data_Block* routines as shown in figure 17 and figure 18.

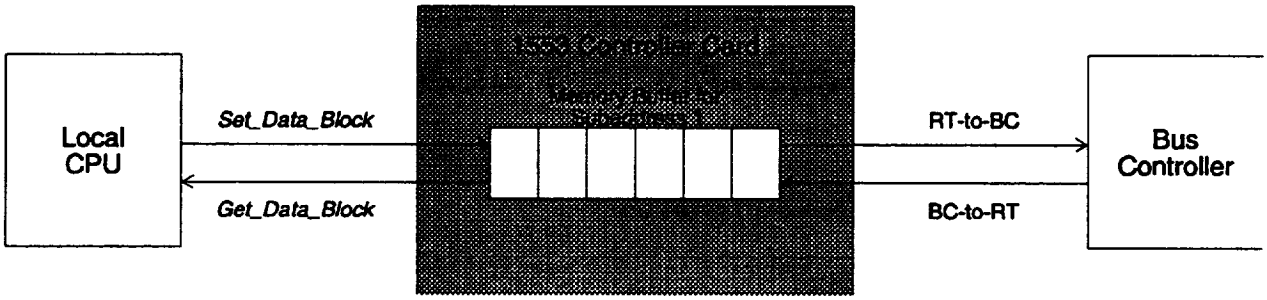

Figure 17 - Reading/writing a shared RT buffer

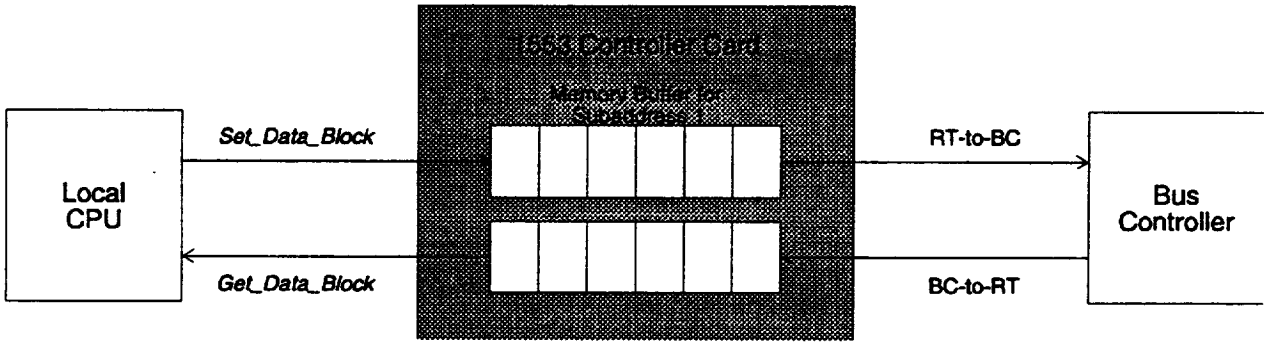

Figure 18 - Reading/writing a unshared RT buffer

A brief description of each RT routine is given below.

- *Share_RT_Buffer* Share the memory area for this particular subaddress with both RT-to-BC transfers and BC-to-RT transfers. Initially, all subaddresses use a shared RT buffer.
- *Unshare_RT_Buffer* Use a different memory area for RT-to-BC transfers than for BC-to-RT transfers.
- *Set_Shared_Block* Set memory within a shared memory block. This procedure may only be used if the RT busy flag has been set. Otherwise, it is possible for the BC to read or write to the block at the same time the RT is writing to it; resulting in a corrupted data structure within the RT buffer.
- *Set_Data_Word* Set the first word within a shared memory block. This procedure is a version of the *Set_Shared_Block* procedure that is optimized for modifying a single data word. It is not necessary to set the busy flag before using this procedure since only a single word is being modified and no possibility of data corruption occurs.
- *Set_Data_Block* Set memory within an unshared memory block.
- *Get_Data_Block* Get data within the memory block at this subaddress. This procedure may be used on both shared and unshared blocks.
- *Get_Data_Word* This routine is the same as the one above except that it is optimized for transfer of a single data word.

- *Set_Busy_Flag* — This routine is used to set the RT's busy flag, preventing the BC from successfully delivering a message to this RT. The node operating as the BC receives an error indicating that a message command failed due to a busy condition in effect at the destination RT. The busy flag is initially set by the *Set_Mode* routine.
- *Clear_Busy_Flag* — Use this routine to clear the RT busy flag. Ensure that the RT's buffers have been initialized to contain valid data before invoking.
- *RT_Address* — Return the local RT address.

RT Exceptions

- *Share_RT_Buffer* — Raises *Invalid_Mode* if the controller is not operating in RT mode.
- *Unshare_RT_Buffer* — Raises *Invalid_Mode* if the controller is not operating in RT mode.
- *Set_Shared_Block* — Raises *Invalid_Command* if the block has not been shared or if the RT *busy flag* has not been set.
- *Set_Data_Word* — Raises *Invalid_Command* if the data block is not being shared.
- *Set_Data_Block* — Raises *Invalid_Command* if the data block is being shared.
- *Get_Data_Block* — Raises *Invalid_Mode* if the controller is not operating in RT mode.
- *Get_Data_Word* — Raises *Invalid_Command* if the data block is not being shared.
- *Set_Busy_Flag* — Raises *Invalid_Mode* if the controller is not operating in RT mode.
- *Clear_Busy_Flag* — Raises *Invalid_Mode* if the controller is not operating in RT mode.
- *RT_Address* — Raises *Invalid_Mode* if the controller is not operating in RT mode.

Low Level 1553 Driver Detailed Design

The low level 1553 driver provides the software interface to the DDC BUS-65515 controller card. The card is configured to operate using an 8K-byte region of memory that is jumper selectable. In the testbed control computers, the 1553 card's memory is configured to reside at memory segments CC00H - CDFFH. The card is described in detail in the reference manual [4]. The terminology and definitions introduced there will be used throughout this description. Please note that any memory locations specified are offsets from the base segment of CC00H.

DDC BUS-65515 Operation

The DDC 65515 card, in this implementation, operates in one of two possible modes: bus controller or remote terminal. Both modes provide the capability of double-buffering messages or data.

BC Mode

Data Blocks

Data for 1553 messages is stored in memory offsets 280H - 1FFFH. This memory region provides 58 data blocks of 64 words. Each data block is big enough to contain one 1553 message, since the largest 1553 message, *Receive_Data_Block*, has a maximum size of 36 words.

Descriptor Stacks

To provide optimum utilization of the 1553 board's double buffering scheme, the descriptor stacks are configured to provide an equal allocation of data blocks between the two available buffers. The descriptor stack for buffer A is set so it contains references to data blocks 1 - 29, and

the descriptor stack for buffer B is set to contain references to blocks 30 - 58. This setup allows each buffer to issue a maximum of 29 messages at one time.

RT Mode

Data Blocks

Data to be store/retrieved from 1553 subaddresses is kept within data blocks at memory locations 400H - 1EEFH. There are a total of 106 data blocks of 32 words. The data blocks are divided into three regions of 32 blocks. Region 1 (blocks 1 - 32) is used for data transfers to those subaddresses configured as *shared buffers*. Region 1 is also used for BC-to-RT transfers to *unshared buffers*. Region 2 (blocks 33 - 64) and region 3 (blocks 65 - 86) are used for RT-to-BC transfers to unshared buffers. Region 2 is used when buffer A is active and region 3 when buffer B is active. The remaining memory blocks are unallocated.

RT Lookup Tables

There are two memory regions on a RT that are used to determine where data (transferred via a 1553 message) is to be stored. These memory regions are known as the *RT lookup tables*. One table is used when buffer A is active and the other when buffer B is active. The lookup tables are used to map RT subaddresses to particular data blocks. Each lookup table contains two entries for each of 1 - 29 subaddresses. One entry is used for RT-to-BC transfers and the other one is used for BC-to-RT transfers.

Initially, the software sets the lookup tables to reference a single set of data blocks (blocks 1 - 32), as shown in figure 19. Each subaddress is mapped to its own unique data block, but that block is used for both RT-to-BC transfers and for BC-to-RT transfers. The arrows designate which RT buffer that will be used when the BC stores/retrieves data at this subaddress. The same blocks are used for RT-to-BC transfers as for BC-to-RT transfers.

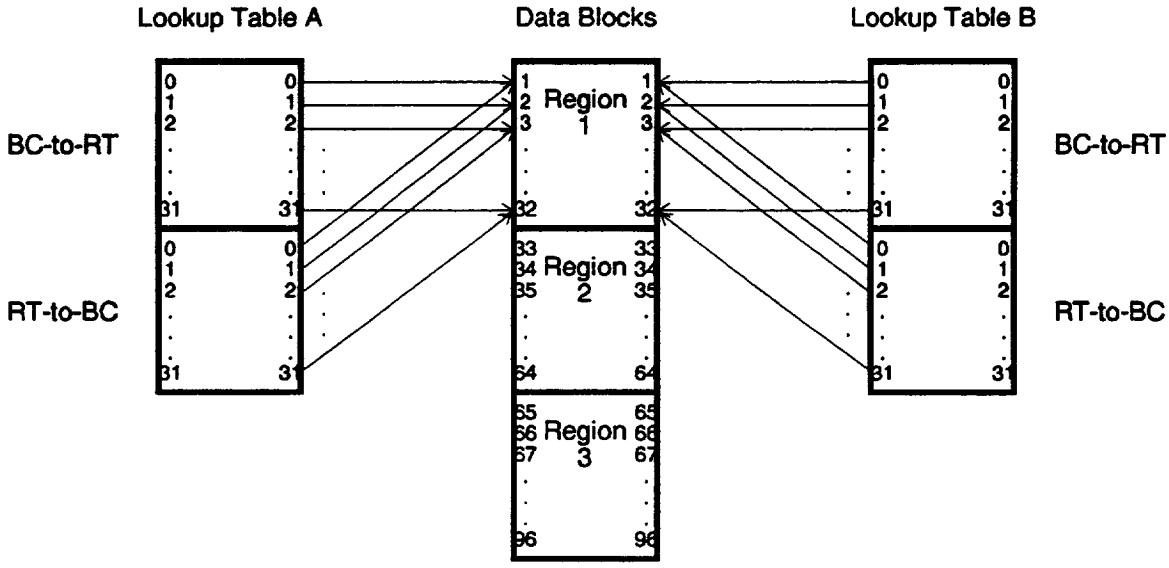

Figure 19 - Lookup pointers for shared blocks

This *shared buffer* mode may be changed so that the transfers use different data blocks. If, for example, RT buffer 1 is place into *unshared* mode, then the lookup tables are changed so that subaddress 1 maps into two different data blocks. RT-to-BC transfers use data block 33 and BC-to-RT transfers use data block 1. Figure 20 shows, for example, the configuration of the lookup pointers when the buffers for subaddresses 0 - 2 are in shared mode and those for subaddress 3 - 31 are in unshared mode.

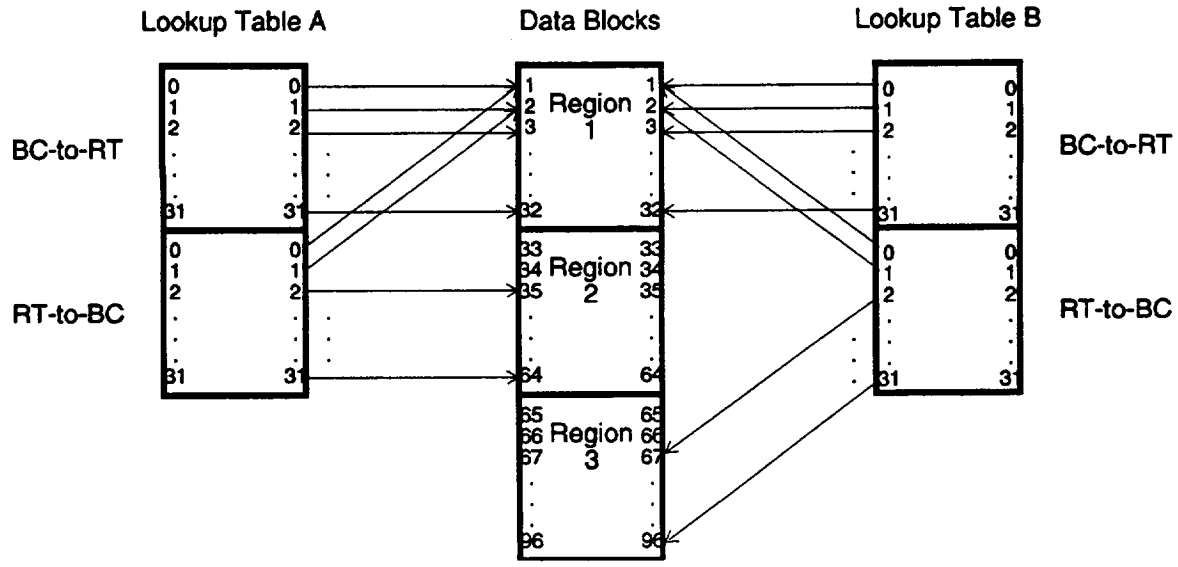

Figure 20 - Lookup up pointers for mix of shared and unshared buffers

Low Level 1553 Driver Internal Design

Private Types

The low level driver defines the following private data types

- *Lookup_Key* - A reference pointer to data that will be obtained once a response to a sequence of 1553 messages has been received. The lookup key simply points to the data block which will hold the response. A lookup key is obtained from the *Build_Transmit_Data_Message* procedure, and the returned data is obtained by using the key in the *Data* function.
- *Message_Block* - A structure which is used to hold a sequence of 1553 messages.

Internal Structures, Variables, and Constants

- *Address_Parity* - An array containing the values *even* or *odd* for each of the 1 - 30 valid 1553 addresses. This array is used to set the correct parity value when the driver is operating in RT mode.
- Command word format - A number of variables (*address_offset, address_bits, TR_offset, TR_bits, subaddress_offset, subaddress_bits, count_offset, count_bits, receive_bit,* and *transmit_bit*) are used to define particular bit fields within the 1553 command word.
- Descriptor Stack - The format of the DDC-65515 board's descriptor stack is define by the structures *BC_Descriptor_Stack_Element, RT_Descriptor_Stack_Element, BC_Descriptor_Stack,* and *RT_Descriptor_Stack.*
- Lookup Table - The RT lookup table structure is defined by the *Lookup_Array* and *Lookup_Table* types.
- Message formats - Structures used to construct 1553 messages are define by the types *Receive_Data_Command, Transmit_Data_Command, Mode_Code_Command,* and *Unspecified_Command.*
- Memory allocation and register definitions - Allocation of memory to match that of the DDC 65515 board is determined by a variety of constants. Some constants define those areas to be used when operating in BC mode and others define areas for RT mode. There is also an area that is used by both

modes. These constants are defined within the code section identified by the heading *Memory Allocation for 1553 Board.*

- *Queue_Info* - A structure which is used to manage 1553 message blocks. Queue_Info is used to insert new messages into the proper area of the board's memory and to prevent overflow. Two state variables *Queue_A_Info* and *Queue_B_Info* are used to control the message construction within the board's two memory buffers.
- *RT_Block_Is_Shared* - This state variable maintains information regarding each of the board's RT buffers. Those that are operating in shared mode (as shown in figure 17) are indicated by a *true* value in this variable.

Internal Procedures

- *Current_Buffer* — Access the 1553 board's register and return the value of the currently active buffer (A or B).
- *Inactive_Buffer* — Return the value of the currently inactive buffer.
- *Queued_Messages* — Return the number of messages currently queued into the specified buffer.
- *Status_Of* — Operates on the 1553 status word and returns a value indicating whether an error occurred.

Diagnostic and Utility Packages

Several diagnostic and utility packages are used by 1553 communication software. The *Semaphore* package and the *Map* package are used by the Network_Utilities package to control operation and access to the message routing table. These packages are general purpose utility packages that are not directly related to the 1553 messaging system. These packages are described in detail in [5].

The diagnostic package, *C_Parallel_Port*, is used by several packages with the testbed system (see figure 8). It sends a voltage signal onto one of the controller's parallel port lines that may be detected and stored by a laboratory logic analyzer. These signals are useful for debugging and testing the Ada code. Its coding is not detailed within this document.

Test Results

Timing and verification statements were added to the 1553 Network Driver code so that its operation could be analyzed. The added statements generate voltage signals on the control computer's parallel port that may be captured and displayed by a digital logic analyzer. The signals that are generated produce a series of output signals that are used to determine that actual execution time of various sections of the Ada code. Table 3 provides a listing of the execution times that were obtained from a bus controller during a typical sequence of network transactions.

The processes listed in columns one and two of table 3 are the same as those shown in figure 13. Column three, *1553 Transmission Time*, provides the time (in micro-seconds) that the 1553 bus is busy during the process indicated in columns one and two. Columns four and five show the amount of time that was spent by the 1553 driver setting up the controller board and, if necessary, retrieving and copying any responses. The final column, *Overhead*, indicates how much time was spent during setup and retrieval as a percentage of the actual bus transmission time.

Process		1553 Transmission	1553 Message Processing		Overhead
#	Name	Time (usec)	Setup (usec)	Response (usec)	%
7	Broadcast Synchronization Command	60	266	0	343%
8a	Query RTs for new data (single node)	88	302	0	243%
8b	Query RTs for free buffer (single node)	88	416	0	373%
8 (a+b)	RT Query (total)	176	718	0	308%
9	Poll for Packets from RTs	19800	4262	23840	42%
11	Transmit Packet to RT	20680	734	0	96%

Table 3 - Network Driver execution times

Conclusion

A description of the 1553 communications software for the LeRC Electrical Power System testbed has been presented. A description of each component of the software, as well as major portions of the Ada code itself, have also been provided. This information is intended primarily for use in future upgrades so that an accurate description of the present design and implementation is available.

References

1 J.F.Soeder and R.J.Frye, "Overview and Evolution of the LeRC PMAD DC Test Bed", IECEC-92, *Proceedings, August, 1992.*

2 A.N.Baez and G.L.Kimnach, "Description of the Control System Design for the Space Station Freedom PMAD DC Testbed", IECEC-91 *Proceedings, August, 1991.*

3 K.Ludwig, T.Wright, and M.Mackin, "Description of Real-Time Ada Software Implementation of a Power System Monitor for the Space Station Freedom PMAD DC Testbed", IECEC-91 *Proceedings, August, 1991.*

4 DDC ILC Data Device Corporation, "MIL-STD-1553 Designers Guide", 1982.

5 Booch, Grady, "Software Components with Ada", © 1987, Benjamin/Cummings Publishing Co.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1995	Technical Memorandum

4. TITLE AND SUBTITLE

Description of a MIL–STD–1553B Data Bus Ada Driver for the LeRC EPS Testbed

5. FUNDING NUMBERS

WU–478–29–10

6. AUTHOR(S)

Michael A. Mackin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135–3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E–9478

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM–106866

11. SUPPLEMENTARY NOTES

Responsible person, Michael A. Mackin, organization code 5450, (216) 433–5326.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62

This publication is available from the NASA Center for Aerospace Information, (301) 621–0390.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This document describes the software designed to provide communication between control computers in the NASA Lewis Research Center Electrical Power System Testbed using MIL–STD–1553B. The software drivers are coded in the Ada programming language and were developed on a MSDOS-based computer workstation. The Electrical Power System (EPS) Testbed is a reduced-scale prototype space station electrical power system. The power system manages and distributes electrical power from the sources (batteries or photovoltaic arrays) to the end-user loads. The electrical system primary operates at 120 volts DC, and the secondary system operates at 28 volts DC. The devices which direct the flow of electrical power are controlled by a network of six control computers. Data and control messages are passed between the computers using the MIL–STD–1553B network. One of the computers, the Power Management Controller (PMC), controls the primary power distribution and another, the Load Management Controller (LMC), controls the secondary power distribution. Each of these computers communicates with two other computers which act as subsidiary controllers. These subsidiary controllers are, in turn, connected to the devices which directly control the flow of electrical power.

14. SUBJECT TERMS	15. NUMBER OF PAGES
MIL–STD–1553B; Network communication; Data and control messages; Ada driver	39
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	