Object-Based Messaging

for the

Dallas Semiconductor

Network Bridge Operating System

Revision 0.85 (Preliminary)
July 31, 1997

By: Chris Fox
Chris.Fox@dalsemi.com

Copyright © 1997 by
Dallas Semiconductor, Inc.
All Rights Reserved

Abstract

The Dallas Semiconductor, Inc. Network Bridge device ("Natural Bridge") is designed to serve as a coupling device between iButton_ (or other One Wire_ serial bus devices or Micro-LAN_ networks) and commercial Ethernet Ten-base-T networks. The unit includes a powerful micro-controller, 32K bytes of working code and data space, over-the-network re-programmability, and an RS232 serial port, all of which make possible a wide variety of network and non-network applications and functions at low cost, some of which include:

- iButton-to-network interface
- Local iButton scanning and asynchronous arrival/departure reporting
- Access control (with local control in case of network failure)
- Machine or subsystem interfaces
- Serial RS232-to-Ethernet interfaces
- iButton-to-serial RS232 interfaces
- Low-cost networked control and sensing
- Local sensing of temperature using temperature sensing iButtons and control of equipment using iButton addressable switches.

To make the Bridge unit easier for users to customize and apply, Dallas Semiconductor provides an operating system that handles all the tedious tasks that add time and risk to the development and deployment of an application project. The Bridge Operating System (BOS) provides:

- Ethernet UDP/IP/Ethernet messaging with assured message delivery facilities and network support (PING, Trace Route, ARP, etc.)
- RS-232 support including interrupt-driven 1024 byte buffered input and output drivers, and optional auto-baud rate detection.
- iButton port support including the lower level TMEX functions (Reset, Bit, Byte, First, Next, Access, and Block-I/O) at standard (14.3 KBPS) and Overdrive (143 KBPS) speeds.
- iButton port capable of +12V programming for EPROM iButton parts, strong pull-up for temperature sensing and Crypto-iButton parts.
- Real-time clock (seconds) and timed events (100 ms)
- Object-based messaging support and MIB
- Administrative page manager (user application code does not have to manage administrative variables)
- Statistics page manager (user application code does not have to manage statistics counters and reporting)
- Application download and verification support
- Four-level application scheduler

One of the most important features of the Bridge Operating System (BOS) is the object based messaging subsystem. This module allows users to perform complex remote tasks with little or no application code. This document will describe the object based messaging concept and the operation of the object based message processing in the Bridge Operating System.

1.0 An Introduction to Object Based Data Transfer

The classic method for the delivery of raw data from one networked device to another is to simply append the various data end-to-end to form a message and send it. The fields in this transmitted data record must also be known to the intended recipient so that the record can be understood and the data elements extracted. Advances in Object Oriented programming methods, however, have taught us that there are advantages to packaging data in ways that make it more self-describing and self-delimiting.

For example, if unit A wishes to send employee badge number 1734, door number 56, and a 4-byte time stamp (2:49AM) to unit B, the typical message might look something like this:

1734 0056 0249

The recipient of the message would have been programmed to expect that the first four characters represent the employee number, the next four are the door number, and the last four are the time of day in order to make sense of the message. And any change in the form of the message, or the data included in it, requires a corresponding change in the programs at both ends of the conversation.

We will now discuss an alternative method for packaging data and commands that is derived from Object Oriented Programming, which we will call *Object-Based Messaging*. Object-based messaging techniques specify that each data item be *tagged* to specify its size, format and intended use. These tags come from a well-known set of codes (called *descriptors*) that are agreed upon universally, or are exchanged when the connection is established or in response to the receipt of an unrecognized code.

Our example message, written in object-based form, looks like this:

Data:Employee ID, Length = 4, ASCII Integer, "1734"
Data:Door Number, Length = 2, ASCII Integer, "56"
Data:Time Of Day, Length = 3, ASCII Time, "249"

Of course, the message does not include the text "Employee ID", but merely a *code* that everyone agrees represents "Employee ID", which is called a *descriptor*. (Indeed, our example indicates ASCII where the actual codes are binary, but you can see the point.)

The *actual* message might look more like this:

12 4 **1734** 24 2 **56** 76 3 **249**

Actual data payload is in **bold** text.

Object Descriptors (underlined):

12	=	Employee ID, Integer
24	=	Door Number, Integer
76	=	Time of Day, ASCII

The 12, 24, and 76 codes are *descriptors* and serve to inform the reader about the format and intention of the data to follow. Notice that the data object type code not only tells what the data item *is* but also includes information about the *format* in which it should be displayed (i.e. Integer, ASCII, etc.). This allows programs that process and display these items to convert them to the appropriate format for our interpretation automatically.

Notice also that including a length with each value not only defines the beginning and ending of each value clearly, but also allows, in this case, three leading zeros to be omitted which helps offset the added overhead imposed by the scheme.

This method also allows messages to be assembled with data objects in any order. New data may be included, or obsolete data omitted, because the recipient can parse the message and use only the values that it recognizes and ignore those it does not.

The normal parser process handles all data the same way. As data objects are parsed, the data they carry is copied out to the appropriate system variable based on the descriptor value. If the area reserved for the data is larger than the data that has been provided in the object, the format field tells the system how to pad it, or interpret it, to fit the space. If the size of the object data sent is too large, the system may ignore it or truncate it as permitted by the format.

Sending a *zero-length* data item represents a *request* for the item, and causes the recipient of the message to retrieve that data item and return it to the requestor.

2.0 Grouping Data Objects

Now that we have defined a method for organizing and delimiting data objects in a message, we also need a method for *grouping* these data objects together and indicating how they are *related* to one another. This will allow us to send more than one event message or report in the same message, or even to send a mixed variety of different reports in the same message, and easily separate them at the receiving end.

To accomplish this data object grouping, a special class of these object type codes is used. These "special" object descriptors are used to signal the start of a group of objects instead of the start of a single data object. These *group descriptors* define the start of a group, and a special marker called a *group end maker* identifies the end of a group. In fact, groups may contain not only data objects, but also other group objects which in turn may contain data and/or group objects, and so on. This hierarchy is very powerful as messages become more complex and serve a wider variety of purposes.

Now, if we apply grouping to our example message, and send two event reports in the same message, we might see a data record that looks something like this:

19 12 4 **1734** 24 2 **56** 76 3 **249** FF 19 12 4 **5676** 24 2 **56** 76 3 **251** FF

Group #1 Group #2

Where:

(Actual data payload is in **bold** text.)

19	=	Group start, Door Access Control
FF	=	Group end marker

Notice that this message contains two door-access messages (Group #1 and Group #2), and yet each is distinct and separable because they are each in a distinct and separate *group*, so there is no doubt which data objects go together and relate to the same access event.

Of course, grouping provides another benefit because additional information about the *meaning* of the data in the group can be conveyed by the group descriptor that is selected. In our example, the group descriptor code 19 could represent "Access Control Event", and so the recipient of this message knows exactly how to process it, or perhaps how to route it to the appropriate processing routine or subsystem.

3.0 Making Things Happen

We have defined an effective way to read and write data, and even to correlate data objects to one another, but our scheme lacks one important thing — the ability to cause things to happen - to invoke *functions*.

One obvious solution to this problem is for the sender to provide a data object with a value that specifies a command of some kind, and then arrange for the recipient to examine this variable when it is received and act upon the value that was written there. But, because most actions taken require additional information about exactly what, when, or how the action is to be performed, and because most data values provided do *not* induce an action to occur, this method is not the best choice.

A more effective approach is to perform actions based on the groups that *contain* data objects and not on the data objects themselves. Group descriptors carry meanings that relate to an event, not just to one data element of the event report, and actions are usually related to the receipt of the report and not to each item of data involved.

Also, because a group contains data objects that deliver information needed to carry out an operation, we should perform these actions at the *group end,* after the data contained inside the group has been evaluated, rather than at the start of the group. This implies that group objects are more like *functions*, and data objects are more like *arguments,* to use classic programming terminology.

4.0 Object Descriptor Codes

The Object Based Messaging scheme requires a predefined set of 16-bit object descriptors that are known to all parties (or "well-known"). The descriptor codes selected need to have some basic qualities to speed-up evaluation and message processing, and to help bring order to the system. These include:

- One specific bit that indicates if the descriptor refers to a data object or a group object.
- A unique and recognizable descriptor code that presents a group-end marker (which should be one of the group-type descriptors).
- In the case of data objects, a fixed set of bits that define the format of the data so that we can always tell how to display it, pad it, encode it, or process it.
- In the case of data objects, a group of bits that provides *general* information about what the data represents.
- In the case of group objects, a group of bits that determines the general *application type* for which the group is intended or by which the group was created.

In addition, this most basic data should be included in the least significant byte of the descriptor because our hardware is Intel_ based and so the least significant byte (LSB) of the descriptor arrives first.

To differentiate between group descriptors and data object descriptors, the most significant bit of the leading (LS) byte is used. When this bit is a zero, the bytes that follow constitute a *data* type object. When this bit is a one, the bytes to follow constitute a *group* type object.

To indicate group-end, the all-ones byte (0xFF) is used. (This value was selected because PROM or EPROM devices that may be used to hold object-based messages usually represent un-programmed bits as all ones. If an un-programmed byte (0xFF) represents a group-end marker, then subsequent over-programming could be performed that would change this byte and extend the group, adding additional data and/or other group objects to it.)

Also, the case where the LSB of any object descriptor is zero (0x00) is reserved as an indicator for some special cases and is never to be assigned to any valid object. This value, for one thing, always halts parsing when encountered as the leading byte of an object descriptor.

4.1 Data Object Descriptors

In all data object descriptors, the least significant four bits (bits 0 through 3) of the LSB indicate the generic data format, as follows:

MSB--LSB
- 0000 — Unknown format (MS nybble cannot be zero)
- 0001 — Byte Array (hex bytes)
- 0010 — Integer (decimal numeric value, leading zero may be suppressed)
- 0011 — Printable String (valid ASCII, printable or display-able as text)
- 0100 — Dallas Serial Number (un-compressed) (FC, S/N, CRC8)
- 0101 — Dallas Serial Number (compressed) (FC, 1-7 bytes of S/N)
- 0110 — Dotted-decimal notation (as in IP addresses) (NBO)
- 0111 — Colon-Delimited Hex (as in Ethernet addresses) (NBO)
- 1000 — Time/Date
- 1001 — Boolean (Flags)
- 1010 — Password (not displayed or masked by "*" characters)
- 1011 — NBO Integer (decimal integer) (NBO)
- (The remaining combinations are undefined at this time)

Where:

NBO means Network Byte Order (MS byte first)
All values, *except* those marked NBO, are LSB first.
FC means Family Code (the first byte of a Dallas iButton serial number)
S/N means Serial Number (LSB first)

In data object descriptors, the remaining three bits (bits 4 through 6) in the object descriptor LSB *may* have further significance. Optionally, these bits may describe further qualities of the data object, such as:

- 000 = Unclassified
- 001 = Noun: Person (name, unique ID)
- 010 = Noun: Place (location, address)
- 011 = Noun: Thing (item, component)
- 100 = Verb: Action (event)
- 101 = Verb: Status (state-of-being)
- 110 = Adjective: Modifier
- 111 = Undefined

The upper-order byte (MSB) of the data object descriptor uniquely identifies the data object. These are assigned as new objects are defined and have no inherent or predefined meaning. As a rule, these objects will be formed as needs arise, and then every user should use the defined ones where they are applicable, instead of creating duplicate data object types which will deplete the number pool. (More about data selecting and using objects will follow in a later section.)

4.2 Group Object Descriptors

In group object descriptors, fewer bits are pre-defined. However, some attempt will be made to assign these descriptors in groups based on the families to which they belong. For example, the following group descriptor LS bytes have already been defined:

0x0080 — 0xFF80	=	System Control, administrative
0x0081 — 0xFF81	=	Dallas Semiconductor, Inc., TMEX
0x0082 — 0x7E82	=	Asset/People/Document Tracking
0x8082 — 0xFF82	=	Access Control, Security
0x0083 — 0xFF83	=	General Data Collection
0x0085 — 0x0F85	=	Serial-to-One-Wire Bridge Applications
0x00FC - 0xFFFD	=	Dynamic Dictionary Objects
0x00FE — 0xFFFE	=	Experimental (prototype)
0xXXAA	=	Reserved, Do not use any group descriptor with 0xAA as an LS byte!
0xCFC2	=	Parse-able Data Group

Note that the MS bit of the LS byte is always a one (0x0080) because that bit defines a group type object descriptor.

Note: For iButton TMEX file structure compatibility, NEVER assign the 0xAA value as the least significant byte in any group descriptor.

(Dallas Semiconductor, Inc., will be the authority for the assignment of "official" group numbers, and E-Mail should be directed to the webmaster@dalsemi.com to request an application for the assignment of a block of numbers for a specific customer application. Groups will be allocated for third party implementors if their application may potentially co-exist with others and the impact of the implementation appears significant enough to warrant this assignment. Blocks will be assigned as small as possible to preserve the number pool.)

4.3 Dictionary Exchange

In programming circles, the term for the official definition of such things as object descriptors is the *dictionary*. As more and more data objects are created and added to the master list of data object descriptors, it is hoped that just about any user application will be able to find applicable object descriptors to use. By using these well-known (i.e. "published") descriptor codes, interactivity with diagnostic tools

and other systems may be supported. However, when special cases require unique group descriptors, a block of values has been set aside called Dynamic Dictionary Objects. This code space allows 256 group types to be defined dynamically.

5.0 Generating Object Based Messages

Because object-based messages are generated and parsed from left-to-right, the mechanism to create these messages works the same way. That is to say, a new message starts with an empty message buffer and then messages are added to it in left-to-right sequence.

Generating object-based messages in the outgoing message buffer is usually done using three subroutines:

- Group_Object
- Close_Group
- Data_Object

The Group_Object subroutine is passed the desired group object descriptor code and adds the two-byte group descriptor to the buffer.

The Close_Group subroutine adds a group end marker to the buffer.

The Data_Object subroutine is passed the desired data object descriptor code, a pointer to the first byte of the data itself, and a length value (in bytes) of the data field. The subroutine adds the data object descriptor to the message, then adds the data length, and then the specified number of data bytes from the pointed memory location.

These subroutines are very simple, and building object-based messages is very easy and efficient.

6.0 Evaluating Objects

Any participant in this scheme wishing to receive messages and process them must have the ability to evaluate each object descriptor and decide if it is valid, if it should be processed, and how. To accomplish this, each participant has a table called a MIB ("Message Information Block", taken from a similar concept in SNMP network management protocols). The MIB table contains information that the system uses to evaluate the objects that it receives and determine their use.

A MIB table entry in the BOS implementation is six bytes long. The MIB contains entries for group objects as well as data objects. The objects may be placed in the MIB in any order, although placing frequently-used entries early in the table will improve performance by speeding up descriptor searches.

Each MIB table entry contains a two-byte object descriptor (LS byte first), a two-byte local memory address, a single byte length and a single byte of attribute flags. We use the assembler macro facility to create legible MIB entries. A macro called "MIB" expects these four arguments and forms the six byte MIB table record. A data object MIB entry in assembler (using the MIB macro) might look like this:

```
MIB    134H, RomData, 8, RE+WE          ; RomData data object
```

In this example, the 0134H is the data object descriptor, "RomData" is the symbolic name (label) of a data storage spot in the user program where the data begins, "8" is the length of the stored variable in bytes, and "RE+WE" constitute the attribute byte, which we will discuss later in this document. If we assume that the RomData variable assembles at address 0x3456 in memory, then the actual (hex) bytes generated for this MIB entry would be as follows:

```
DB     34H, 1H, 56H, 34H, 8H, 3H
```

Notice that the 134H and RomData values have been stored in LSB-first format as-per Intel_ byte ordering (IBO).

The MIB table is always ended by a single zero-value byte (which is illegal as a descriptor LS byte). This end marker may be generated using the MIB macro by forming a line like this:

```
MIB End
```

The "MIB End" macro call performs two functions. It generates the 0x00 MIB end byte, but it also sets the address pointer back one increment ("org $-1") so that any subsequent data added to the MIB segment will over-store the ending marker and thereby extend the MIB table. Using this method, the MIB may be defined in many different code segments and each may have a "MIB End"

statement, with only the last one in the assembly being left when the MIB is fully linked.

6.1 Parsing an Object-Based Message

When an object-based message is received, it is processed by a recursive parser. The parser proceeds through the message from left-to-right and processes objects as they are encountered, following roughly this basic flow:

1. Get the first byte of the object descriptor (LSB)
2. If it is a group descriptor (MS bit = 1), then:
3. If the byte is 0xFF (group end), then:
4. Decrement a depth counter
5. If the depth did not go to zero, then "push-under" a return address on the stack to #1.
6. Execute vector that is found on the top of the stack (return).
7. Get the next byte (MSB) of the descriptor
8. Look up the descriptor in the MIB table
9. If found, push the local address from the MIB on the stack, increment a depth counter, and then go #1.
10. If not found, skip this entire group and any or all groups or data objects found within it, then go to #1
11. If it is a data descriptor, then:
12. Get the next byte (MSB) of the descriptor
13. Get the data object length byte
14. Look up the descriptor in the MIB table
15. If not found, skip the contents of the data object, go to #1.
16. If the length value is zero, then:
17. Build a data object in the outbound buffer with the same
18. descriptor and the data object length from the MIB, and then copy the data object bytes to the new object in the outbound buffer.
19. Else, if the length value is non-zero:
20. Verify that the length is not greater than the length specified in the MIB table entry. If it is, skip this object, and then go to #1.
21. Use the memory address from the MIB to copy the data object payload to the memory variable, then go to #1.

Note: "Push-Under" is a method wherein a vector address is placed on the stack under the current top-of-stack vector so that, when a return is executed, the top of stack vector will be executed and then, on the next return, control will go to the vector address that was "pushed-under". It makes for a "do that and then come back to me" kind of action when a return is executed.

As you can see, this parser is not terribly complex. The stack is used to hold addresses of group functions from the MIB table until the group ends, and a group-depth counter is used to determine when the parsing is completed and

control may return to the process that called the parser. Data objects are copied to or from their places in memory based on their lengths, and group-functions are executed as they are encountered and then closed.

Note that any group object that is used to group data objects and needs perform no function execution simply has a local address in the MIB that points to only a return op-code; in other words, to a "null function".

One function of the MIB table is to isolate and translate local memory data and function addresses from the object-based messages. When a code change and re-assembly moves a function or variable, the MIB values change and the message format and values are unchanged, so message building is unaffected. The MIB provides "re-direction" of data and function addresses.

6.2 MIB Data Object Attributes

The sixth byte in each MIB entry is the attribute byte. These eight flag bits are used to define the attributes of each data object as follows:

0. Read Enable	Bit name symbol:	RE
1. Write Enable		WE
2. Password Protect		PW
3. Scratch Pad Addressing		SC
4. Page bit 0		P0
5. Page bit 1		P1
6. Page bit 2		P2

The **Read Enable** (RE) bit must be set before the parser will allow a variable to be read and returned in the outbound message. If the Read Enable flag is not set, the variable will not be returned, and a zero-length data object for this variable will be ignored.

The **Write Enable** (WE) bit must be set before the parser will allow a variable to be modified by the inbound message. If the Write Enable flag is not set, the variable will not be written, and a non-zero-length data object for this variable will be ignored.

The **Password Protect** (PW) bit, if set, will block all access (read or write) to the data object variable unless a correct password is passed to the unit in a prior data object and password group object in the same message (See Passwords below). (Note that password protected writing along with un-protected reading can be accomplished using two MIB entries, one read-only and one password protected which is write enabled.)

The **Scratch Pad** (SC) bit is set if the variable is located in scratchpad RAM space, which is an alternate data area in the Intel_ based micro-controllers. Scratchpad variables may be handled in the MIB and accessed in the same way as movx RAM variables.

The **Ram page bit 0, 1, and 2** bits represent a page-select value for the memory in the 87C520 based bridge unit. In this unit, there are eight pages of RAM memory each containing 65,536 bytes of working data storage. These bits allow the parser to access data objects in any of the eight pages of memory.

6.3 MIB Group Object Attributes

The MIB entry attribute bytes also have meaning for groups. These flag bits are used to define the attributes of each group object as follows:

2. Password Protect PW

7. Pre-Execute PE

The **Password Protect** bit, when set, specifies that the proper password has to be provided before this group will be recognized. If the password match is not correct, the group is skipped as if it was unknown to the system.

The **Pre-Execute** bit, when set, causes the group function to be performed prior to parsing the group contents, instead of at the group end as is normal.

7.0 A Syntax and Notation for Object Based Messaging

Now that our messages are becoming more powerful, they are also becoming more complex, and harder for humans to read and understand. Next we will introduce a notation and syntax for displaying and describing object based messages which will make the hierarchy and grouping easier to see.

Each data object begins with the type code, or *descriptor*. However, there are both data and group types, so we will tell them apart by adding a prefix **"D:"** to data objects and a prefix **"G:"** to the group objects. Since each object begins with this descriptor, and also a length byte in the case of data objects, we will also contain the object descriptors inside curly-braces "{}" so that we can tell where the descriptor ends and the data begins, and we put the data payload length in parenthesis in data objects.

Data Objects —

Here are some data object examples:

{D: Door Number (4)} 1234
{D: Door Codes (4)} 03 B5 F7 66
{D: Door Name (14)} "South Entrance"

Notice that the descriptor is shown in plain-english and not by the "12" code that we used in previous examples. Since the descriptors are well-known and published, our tools and programs can perform the translation for us and we do not have to see all those number codes. We also display the value in the proper format (i.e. integer, array, and string) because the data object descriptor tells us what the intended format of the item is. And, if we have access to the dictionary, we can display even more about the object and what it represents.

6.2 Group Function Execution

As we touched on earlier, groups may have functions affiliated with them, executed at group-end time. Here is an example message that we will use to discuss group-functions:

```
1.      {G: Send Message} [
2.              {G: One Wire Search} [
3.                      {D: Depth (1)} 65
4.                      {D: Channel (1)} 3
5.                      {D: Line Control (1)} 01001101
6.              ]
7.              {D: Line Status (0)}
8.              {D: Serial Number (0)}
9.              {D: Depth (0)}
10.             {G: Reply} [
11.             ]
12.     ]
```

The line numbers on the left are for reference only. This message, when parsed, will perform as follows:

Line #1 starts a Send Message group. No action takes place yet, but the address of the Send Message subroutine is pushed on the stack and the group depth is incremented.

Line #2 starts a One Wire Search function. Once again, however, no action takes place yet. The One Wire Search subroutine address is now pushed on the stack.

Line #3 contains a data object that sets a variable called *Depth* to 65. This variable is required by the One Wire Search function, and a value of 65 (decimal) causes the search to begin at the top (sometimes called a "First"). No action takes place now except that the variable value in memory is changed to the value provided in the message.

Line #4 contains a data object that sets a variable called Channel to 3. Again, no action occurs yet except that the *channel* variable is set to a value of 3.

Line #5 shows a boolean data object setting a byte of line control flag bits to the value 01001101 (binary). Again, this data object does not cause any action to occur, except that the value of the *line control* variable is changed in the target system memory.

Line #6 is the close of the One Wire Search group, and so a return is performed to the top of the stack, which executes the last function pushed, or the One Wire Search function. The Depth, Channel, and Line Control values that were set inside

the group are used by the One Wire Search function to perform the search action. On return from the search function, the parsing of the message continues.

Lines #7, 8 and 9 show three zero-length variables. When the parser encounters these it obtains the data specified by their descriptors and creates data objects in the output message for their values. In other words, three data objects will be created in the output buffer for the Line Status, Serial Number and Depth variables which have been changed by the Search function when it was executed.

Line #10 is the start of a Reply group which ends promptly at line #11. This empty group will perform a function called Reply that addresses the outbound message to the requestor (in other words, back to us).

Line #12 is the close of the Send Message group, and so the message that we have assembled in the target system is now sent back to us.

So, it can be seen that we have sent the distant node a command that not only changed some variables in it but then performed some functions, assembled a response message, added variables to it, addressed it to us, and then delivered it.

Consider this alternative message:

```
1.      {G: Send Message} [
2.              {G: Reply} [
3.                      {G: One Wire Search} [
4.                              {D: Depth (1)} 65
5.                              {D: Channel (1)} 3
6.                              {D: Line Control (1)} 01001101
7.                      ]
8.                      {D: Line Status (0)}
9.                      {D: Serial Number (0)}
10.                     {D: Depth (0)}
11.             ]
12.     ]
```

This version performs the same way as the first example, but the Reply group *includes* the One Wire Search group. This is also acceptable because the actions occur at the group end (at the "]" symbol) and so the result is the same.

(In the DS5000 based controllers, there is little stack space and so the depth of groups should be minimized.)

7.0 Bridge Operating System MIB

The Bridge Operating System provides 1024 bytes for the system parser MIB table, or about 170 MIB entries. User applications are encouraged to add to this table as they wish by over-storing the BOS MIB end marker, adding more MIB entries, and then providing a MIB end maker of their own. The BOS MIB contains a number of useful data and group objects.

7.1 Reliable Network Protocol Objects

In earlier versions of the Bridge Operating System, a networking layer called DDP (Data Delivery Protocol) was added under UDP/IP to provide reliable messaging. This layer is now replaced by two group functions that perform the same functions using the object based messaging scheme. The example in the previous section would send a UDP/IP message only, and there would be no assurance of message delivery by the operating system (which is the nature of UDP). However, the BOS MIB includes a group-function called "Validate" and a group-function called "ACK" that provide the same services that the DDP layer used to provide.

To provide reliable message delivery under UDP, the sender must transmit a message and then wait for an acknowledgment from the recipient. To make sure that the acknowledgment is for the correct message, each transmitted message must include a sequence number that is returned in the acknowledgment and must match the one that was sent. The sequence number is simply incremented each time a new message is sent. If the sender does not receive a correct acknowledgment, it should re-send the message. This solves the problem of reliable message delivery, but introduces a new problem. If the message gets across but the acknowledgment is lost, the message would be sent again and the recipient would end up with two copies, not knowing if the second is a valid message or simply a copy of the first. To resolve this, the recipient checks the sequence numbers and rejects any message in which the sequence number matches the previous one (although all messages are acknowledged to satisfy the sender).

When a message is to be sent using reliable protocol, the first group in it should be constructed as follows:

```
{G: Validate} [
        {D: Transmit Sequence Number (1)} nn
]
```

When the Validate group function is performed, it sends an acknowledgment message to the sender. Then it checks the sequence number variable (nn) to see if

it is the same as the last one received and, if it is, the remainder of the message is rejected.

The way that a recipient sends an acknowledgment is to build a message with this form:

```
{G: Acknowledgment} [
        {D: Received Sequence Number (1)} nn
]
```

This group function causes the sequence number that was just received to be sent back to the sender to acknowledge receipt of the message. The current outbound message is also addressed as a reply by this group function so that the acknowledgement message will be directed back to the sender of the original message.

If the message that was sent causes other data to be assembled in the outbound message buffer, it all be added after the Acknowledgment group and sent at the same time. This allows acknowledgements and response data to share the same message and reduces network traffic.

The BOS also assumes that data assembled in the network message buffer is to be sent out and so performs the Send Message function when parsing of the inbound message is completed automatically if anything has been assembled in the outbound buffer. This means that the {G: Send Message} [ ] group in the examples above could have been omitted and the message would have been sent anyway by the BOS when the inbound message parsing was finished.

The BOS MIB includes a set of network messaging groups that are used to form and address network messages. One of these is Start Message, which prepares a reliable delivery message (like the one shown above) in the outbound buffer and manages the sequence number issues invisibly. So, to send a reliable message and request the return of data using reliable delivery, one could form a request like this:

```
{G: Start Message} [
]
{D: Serial Number (0)}
{D: Depth (0)}
```

This simple message would cause the node to build a message like this and send it back to us:

```
{G: Ack}[
        {D: Received Sequence Number (1)} xx (whatever was received)
]
{G: Validate} [
        {D: Transmit Sequence Number (1)} nn (next xmit seq number)
]
{D: Serial Number (8)} xx xx xx xx xx xx xx xx  (whatever the serial # is)
{D: Depth (1)} xx (whatever the depth is)
```

The first item that we would get back is the acknowledgement of the message that we sent out. This {G: Ack} group causes us to stop any retries pending of the original request message. The Received Sequence Number value would be the same as the one we sent out in our original message.

The second item {G: Validate} is the request to validate the response message, and would invoke a function to evaluate the enclosed transmit sequence number and see if this message is a duplicate or not. If it is we will abort parsing and ignore the rest of the message.

Then, we get our payload data (serial number and depth values) as requested.

Note that we did not need to use the Reply group because the acknowledgement has already addressed the message as a reply.

Group Objects —

Group objects cause our syntax to impose indenting so group relationships can be readily distinguished. Group affiliation is very important in this scheme and moving a group-end marker can radically change the meaning of a message. An opening square bracket ("[") is used to mark the beginning of a group payload and a closing square bracket ("[") is used to mark the end of a group. Note that this closing bracket implies a group end maker byte in the actual message.

An example is:

```
{G: Access Event} [
        {G: Door Information} [
                {D: Door Number (4)} 1234
                {D: Door Codes (4)} 03 B5 F7 66
                {D: Door Name (14)} "South Entrance"
        ]
        {D: Serial Number, Person (8)} 04 12 B5 6E 00 00 00 C6
        {D: Name, Person (10)} "Joe Schmoe"
        {D: Door Number (4)} 1234
        {D: Time/Date (4)} 12:34PM 1/2/97
]
```

In this example, a single access control event has produced a message which contains door information as well as information about the person attempting to enter, and the time and date. All this is obvious from an examination of the message descriptors and structure. (This entire message is only 73 bytes long, so this a great deal of data to be conveyed in a very few message bytes.)

Note also that the door information is contained in a group within the event report group. This kind of group-in-a-group hierarchy allows data to be associated logically with the other data, or the event, with which it is connected.

This example also shows how indenting on the display screen helps the reader to clearly visualize the group associations.

8.0 The ADMIN Utility

A Windows_ based (16 bit, Win 3.11 or Win 95) utility has been written that will allow users to access networked Bridge devices and manipulate them. This utility can perform the following:

- Read device statistics (statistics page)
- Read/Modify device system variables (administrative page)
- Down-load firmware to the device anywhere in memory
- Up-load code or data from anywhere in the device memory
- Build object-based messages in indented, textual form (as defined in section 7) and send them to any device
- Receive and parse object based messages, showing them in indented textual form (as in the examples in section 7)
- Display the hexadecimal codes generated by the message builder

See the Admin Utility User's Guide for more information on this administrative and development tool.

10.0 An Error Control method for Object Based Messages

When sending data across unreliable or error-prone links there is often a need to impose error an error detecting scheme on the data. Serial links, by their nature, are non-packetized, and so require *bounding* of the data so that the start and end of each message can be determined.

Error control introduces a problem for the object based paradigm because parsing is usually a left-to-right process and error control requires the examination of a message in its entirety before processing begins. To do this, an error control group is used as a message prefix, as in this example:

```
{G: Error Control} [
        {D: CRC16 (2)} 1234
]
```

The Error Control group function will compute the CRC-16 of all of the bytes from the byte following the end of this group to the end marker of the next group in the inbound message buffer and, if they do not match the value provided in the CRC16 variable, will abort parsing. A process that is reading data from an unreliable source would also need to require that the leading object descriptor always be a valid Error Control object, or the message is rejected.

11.0 Other Media and Object-Based Messaging

There is a natural tendency at this point to consider using this same object-based scheme for the information transmitted via the serial port, or that stored in **iButton** devices. Indeed, there are many advantages to this method.

When object-based messages are written in iButton devices, the Dallas Semiconductor TMEX file and directory structure should be used. If the TMEX directory overhead is prohibitive, then at least the TMEX file data record structure should be used. These structures provide for length determination and error control necessary to prevent errors.

When using serial line communications, the TMEX file data record structure is also a convenient method for error control, although it limits any single block to 252 bytes in length. When using the standard data packet on a serial line, use an incrementing sequence number as the continuation pointer, and include this number in the CRC16 computation to retain TMEX compatibility.

11.1 An iButton Example

As an example iButton application, let us propose that the presence of several components of a subsystem must be known to a system before the process will be deemed complete and allowed to operate. Each of these components has an iButton affixed to it to be used to identify it, and each component is tethered (a sustained iButton connection is established) to the controller iButton bus when it is brought into the configuration.

The process that scans the iButton bus for arriving iButton devices would, on detecting a new device, generate a message as follows:

```
{G: Arrival}[
        {D: Serial Number (8)} 01:02:03:04:05:06:07:08
        {D: Channel (1)} 3
]
```

This message is assembled in the inbound message buffer and parsing of it is performed. The Arrival group-function reads the iButton device data and checks to see if the leading bytes represent a valid Error Control group descriptor. If not, the message is discarded and parsing ceases. (If the leading iButton data bytes represent a TMEX file directory structure, perhaps a specific file is sought and read instead.) However, if the contents of the iButton device appear valid, then the remainder of the iButton data is read into the inbound buffer and parsing of it is enabled.

12.0 The Parse-able Message group

One problem with parsed messages is that an arriving message from an unknown source cannot be reliably determined to be object-based. Parsing non-object data would usually result in a large number of undefined data and group types, but could also trigger events that were not intended to occur.

To minimize this possibility and help validate messages prior to parsing them, the most obvious method is to simply place the entire message in a single group and then arrange for the recipient to verify that the group descriptor begins the message before allowing it to be parsed.

Toward this end, we must define a group type descriptor that means, "Parse-able Message". The requirements for the value selected include:

- A value not likely to exist in non-object data
- A value that is non-ASCII in both bytes
- A value that has the MS bit in LS byte set
- A value that is uncommon in test patterns or initialization patterns
- Two bytes are not identical or complementary values

These rules eliminate all byte values below 128, the byte values 0x55, 0xAA, 0x00 and 0xFF. Based on this, the descriptor value 0xCFC2 has been assigned.

-end-