intel

APPLICATION NOTE

AP-109

December 1980

Using Intel Single Board Computers for Serial Distributed Processing Links

Peter L. Andersen
OEM Microcomputer Systems
Applications Engineering

143347

RELATED INTEL PUBLICATIONS

RMX/80™ User's Guide, 9800522

RMX/80™ Interactive Configuration Utility User's Guide, 142603

Multiprocessing Extensions for the RMX/80™ Real-Time Executive, AP-88, 142725

Using the iSBC 544™ Intelligent Communications Controller, AP-53, 9800933

iSBX 351™ Serial MULTIMODULE™ Board Hardware Reference Manual, 9803190

Simplifying Complex Designs Using the iRMX 80™ Nucleus, AP-112, 143349

Using Intel Single Board Computers for Serial Distributed Processing Links

Contents

INTRODUCTION

System designers are satisfying more and more difficult application needs with solutions which use microprocesors in a distributed environment. This distribution of intelligence results in many system benefits, including a simplification of the design and a resulting decrease in the development time of the system solution. An additional feature is the minimization of the installation costs of the system resulting from reduced wiring and from resource sharing. However, to effectively create distributed solutions, a reliable communication scheme must be used which links the various parts of the solution together.

The purpose of this application note is to demonstrate the ease with which Intel iSBC™ boards can be configured to implement distributed solutions. Several approaches are offered which apply standard operating modes of both the hardware and of the software. While certainly not the only solution, a distributed processor communications protocol is developed which will support many typical applications.

Distributed systems generally fall into one of two categories. The first consists of those systems in which the processors are located in such a manner as to allow communications over a common system data bus, such as Intel's MULTIBUS™ system bus. The second category involves systems in which the processors are also geographically distributed. This class of system generally uses a form of serial communications to allow each processor to communicate with other processors in the system.

The emphasis of this application note is on serial multiprocessing links. Before proceeding with a development of application solutions which involve serial communications, this note includes a short discussion of common network designs.

Common Network Designs

Serial networks may be classed into three general categories. These are the LOOP, the MULTIDROP, and the STAR. Each has applicability to certain iSBC products and has distinct advantages and disadvantages in an application. After discussing the characteristics of each network, an application scenario illustrates their uses.

LOOP NETWORKS

The loop represents the most common of the communication schemes. It is derived from the older teletype communication networks in which several printers were connected in series. Any data generated on one machine causes all machines concerned in the network to echo the data. This arrangement is shown in Figure 1. Loop communications normally use a 20-milliamp signal to convey the data. The presence of this current, known as marking, represents a binary 1 and the absence, or spacing, of any circuit (an open circuit) represents a binary 0. Both full and half duplex configurations are commonly used.

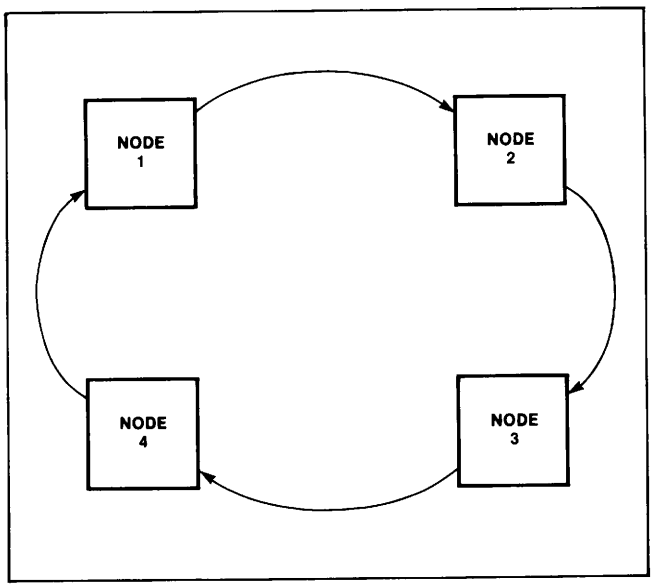


Figure 1. Serial Loop Configuration

The incorporation of loop networks into a system implementation creates both advantages and disadvantages. The strongest argument for using a loop is the inherent noise immunity which can be gained by using the 20-milliamp current to convey the data. The use of a high compliance voltage also allows the transmission of data for large distances at low to moderate transmission rates.

Solid state circuits common to current technology are somewhat less reliable than older mechanical units when many stations are configured onto the loop. This is the result of the requirement to reproduce the information at each node for transmission to the next node. If any node fails, all other nodes downstream in the communication network will not be capable of communicating. Even with this constraint, many good designs incorporate the loop network concepts.

MULTIDROP NETWORKS

Multidrop networks are fast becoming the accepted network for multiprocessor communications. This type of network is shown in Figure 2. As can be seen, this arrangement is similar to that of the loop. The advantage is that all stations are capable of receiving data simultaneously. Multidrop networks are voltage driven since to pass current through a node creates a loop situation. The electrical interface for multidrop networks consists of tri-state drivers and high impedance receivers. The RS422 electrical interface is common for this type of network since it provides high speed data transmission over long distances with good noise immunity.

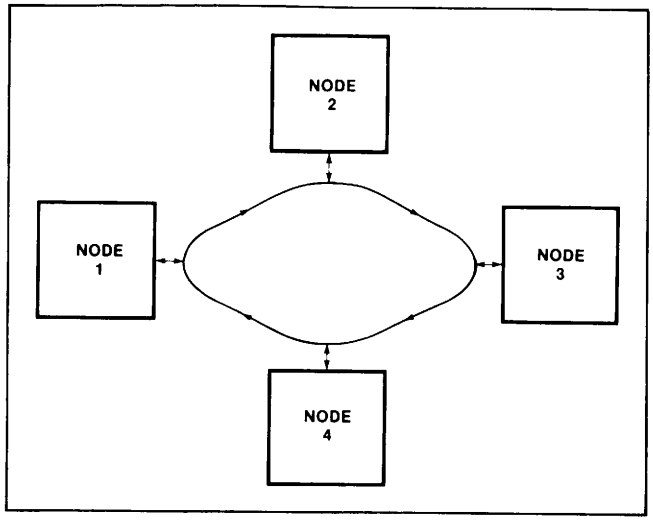


Figure 2. Multidrop Loop Network

The ability to implement networks with a minimum amount of hardware and the potential expansion capabilities provide the multidrop network with a significant advantage in many applications. Since each node in the network can be added by simply adding another tee network, expansion is performed without the need for modification of the communication network.

Intel's iSBX 351™ Serial MULTIMODULE™ Board is an ideal vehicle for the implementation of multidrop networks. Its use is detailed in subsequent paragraphs of this application note.

STAR NETWORKS

Star networks are implemented where either data throughput or hardware limitations prohibit the use of other types of communication arrangements. Figure 3 illustrates a typical star network implementation. Note that a star network is really an example of multiple point to point communications. System throughput is improved since communication can occur with all slave nodes simultaneously without the need to have each node monitor traffic which is not intended for it.

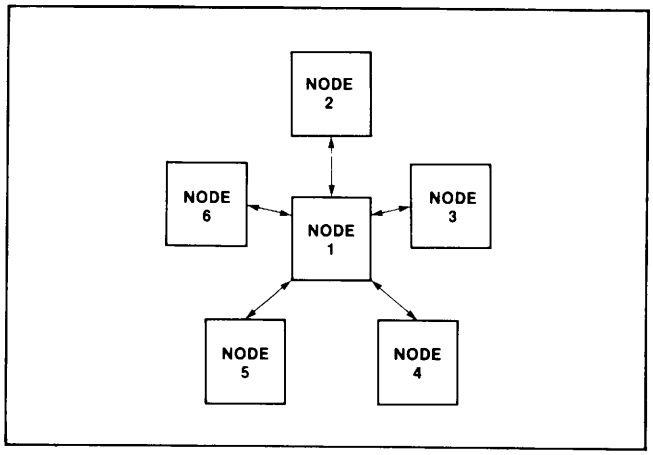


Figure 3. Star Configuration

Intel's iSBC 544™ Intelligent Communications Controller and the iSBC 534™ Four-Channel Communications Board are ideal choices to design star communications networks. The outer points of the star are implemented using any single board computer which has on-board serial communications. An example of this type of network is included in this application note.

Sample Application

An application example is used in this application note to illustrate the techniques which are required to implement various types of serial communication networks. Both hardware and software aspects of the design are described in some detail.

The application discussed in this note involves the creation of an alarm and security system for a multiple building complex. This complex is shown in Figure 4. The solution generated allows monitoring of three existing buildings with provision for the addition of a fourth at a later date. The figure shows that each building is to have a local annunciator panel for reporting activity in those areas served by the building sensors. In addition, a main security station provides monitoring of all buildings in the complex.

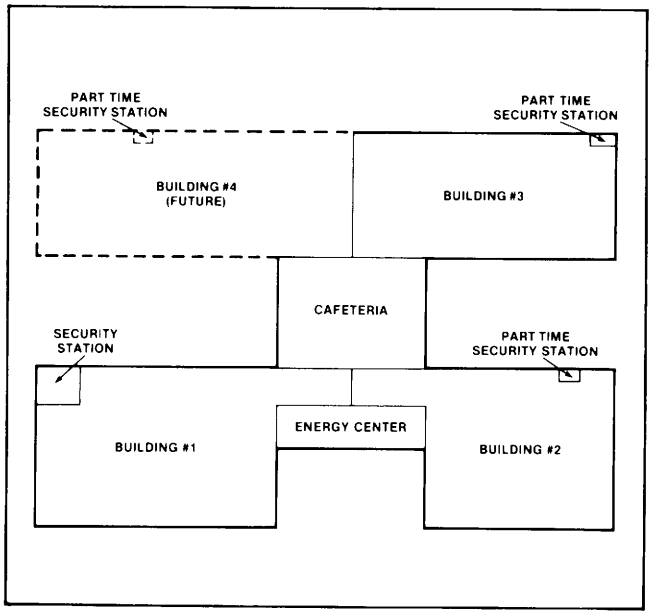


Figure 4. Application Building Complex

Applications such as this are optimized by distributing the system intelligence throughout the complex. This serves two needs; that of minimizing the wiring costs, and of providing an improvement in system reliability. When a system is distributed in this way, a means must be devised which allows communication between the various elements of the network. Both hardware network design and software communication protocol are required before the system can be considered operational.

In order to illustrate various network designs, the example is broken into a distributed system as shown in Figure 5. This is an example of a multidrop communication network. The system design does not require that building nodes communicate with each other; all communications are between building annunciator nodes and the main operator station. This illustrates the concept of a master and a slave. This will be further discussed in a later portion of this application note.

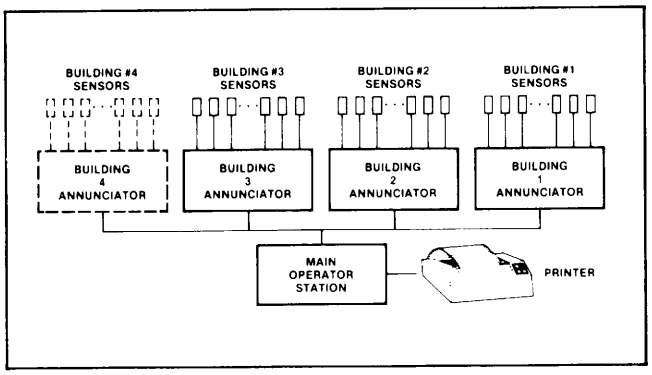


Figure 5. A Multidrop Application

The building annunciator can be further distributed as shown in Figure 6. Here, a star communication network is created which places data gathering intelligence near clusters of sensors. These stations, or nodes, then transmit information regarding local activity to their master node, the building concentrator/annunciator panel. Note that this panel serves both as a master node for the sensor units and as a slave node for the main operator station. This arrangement is common in many system designs.

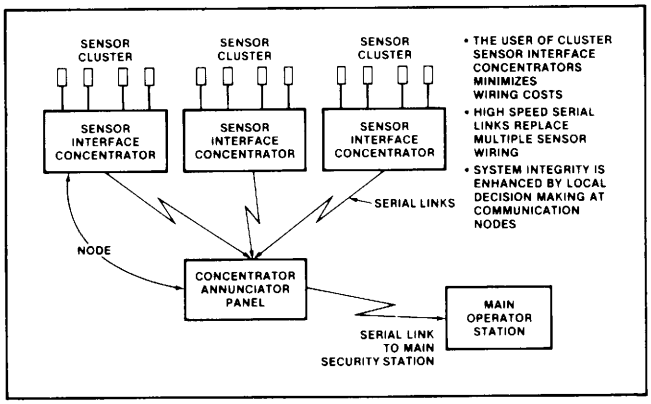


Figure 6. A Star Application

HARDWARE IMPLEMENTATION

The implementation of communication networks using iSBC boards for both star and multidrop techniques is detailed in the following sections. In each case, a master/slave relationship is assumed. For purposes of illustration, actual data from the application example is used in performing the sample calculations. While not the only possible system designs, those shown are considered to be typical and should enable the reader to gain an insight into techniques which can be used to create similar designs.

Star Network Design

A star network can use almost any accepted transmission media. This is because each node communicates with only one other node, creating a multiple point to point link. Since the techniques are identical for all media, this note discusses only a representative example. RS232C data transmission is used to demonstrate the techniques which can be used to establish a star network.

The heart of a star network is the master node. It must provide a means of independent communication with each of its slave nodes. Costs can be minimized if multiple communication drivers are placed onto the same board. The example chosen requires that four slave nodes be controlled by the master. The iSBC 544 Intelligent Communications Controller provides this function. It provides a software selectable baud rate and the ability to offload the host processor of communication activities. If additional slave nodes are required, the system can be expanded by adding iSBC 544 controller boards.

An ideal choice for a slave node is the iSBC 80/10B™ Single Board Computer. This computer provides good processing capabilities with low cost. It includes both an RS232C and 20-milliamp current loop communications interface. For many small applications it makes an ideal choice for a complete single board solution.

Figure 7 shows the implementation for a four-node plus master star communication network for the security application. Both the iSBC 544 communications board and the iSBC 80/10B Single Board Computer require jumpering and component insertion to allow operation in either the RS232C mode or in the EIA mode. EIA operation is electrically identical to RS232C but has specifications which allow transmissions at significant distances when the baud rate is reduced. For the purposes of this application note, the terms are used interchangeably.

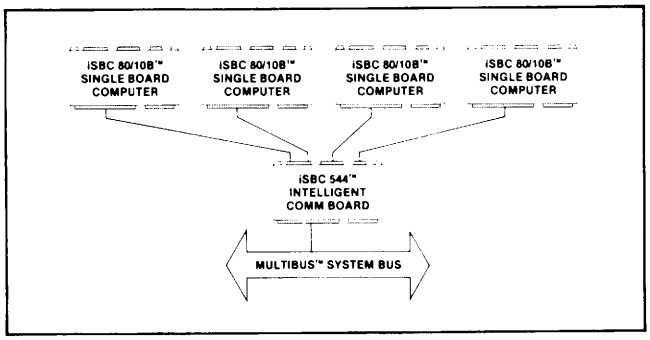


Figure 7. Star Implementation

In the case of the iSBC 544 board, a header plug can be inserted into an on-board connector to enable the correct mode of operation. The wiring for this header is shown in Figure 8. Note that the ready to send and the clear to send signals are jumpered to allow the correct operation of the USART without the control signal of a modem. Also note that the transmit and receive signals are reversed through the header.

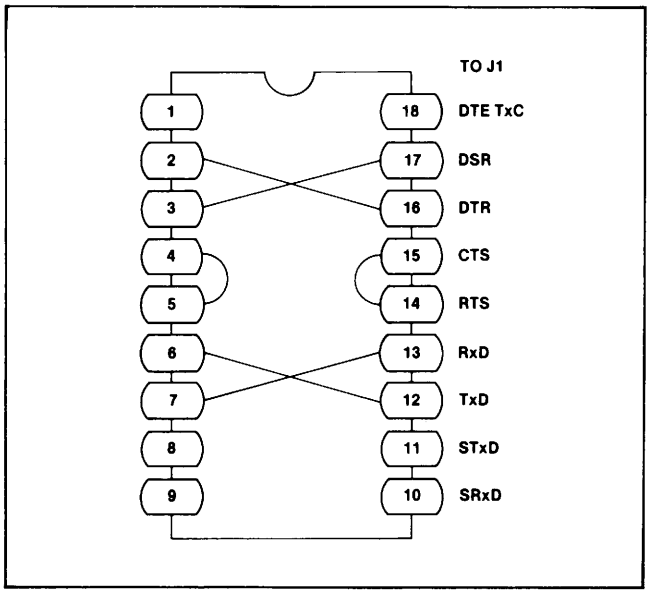


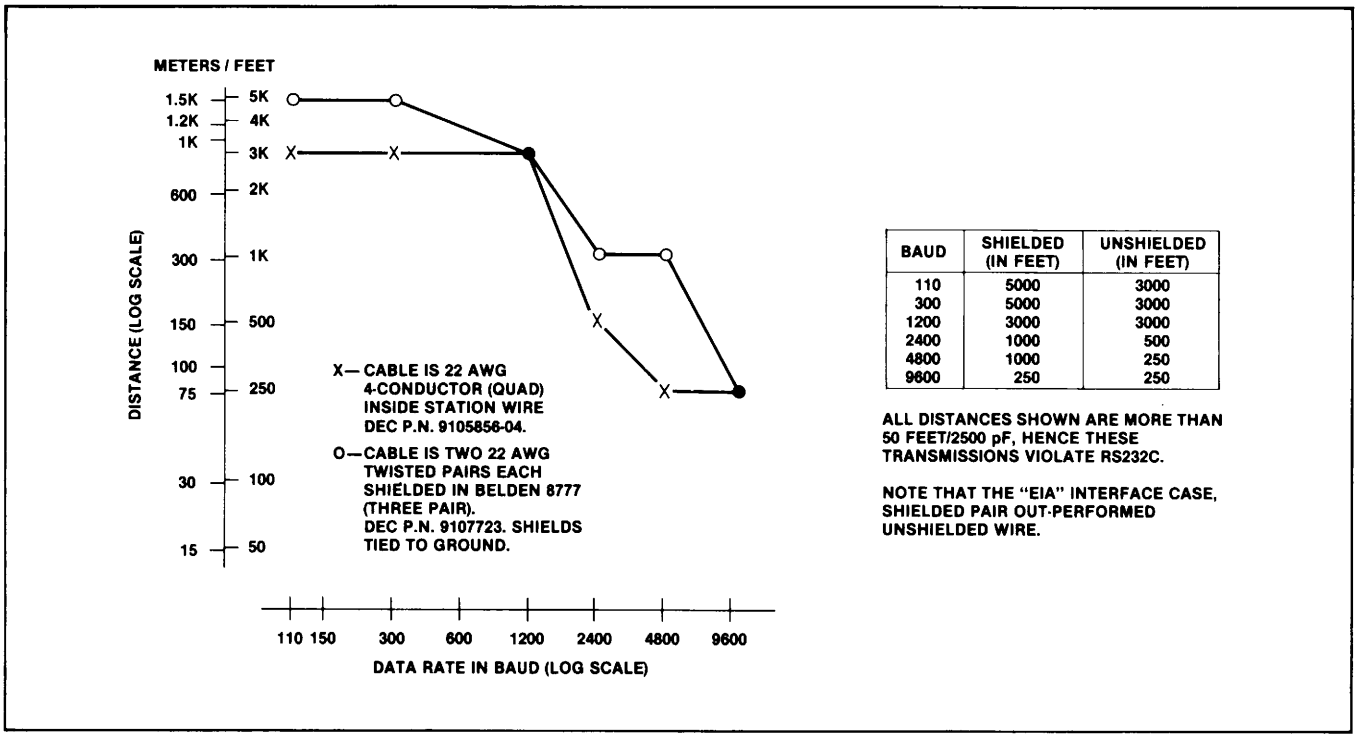
Figure 8. Master to Master Wiring Header

The use of an RS232C signal for long distance communication necessitates the use of a low baud rate for reliable data transmissions. This is shown in Figure 9.

The security system application design allows for minimum communication traffic so a baud rate of 1200 is used for the star networks. The placement of intelligence at the sensor cluster minimizes the amount of data which must be transmitted, providing adequate response time at this data rate.

The hardware design phase includes the selection of the communication scheme to be used between the host processor and the slave node. Fortunately, this task is simplified by the use of built-in master/slave protocol functions contained on the iSBC 544 communications board. These functions involve three hardware features. The first is the incorporation of dual-ported memory onto the board. This memory provides a media for the storage of data and commands for communication between the slave and the host processor. The second feature generates a flag interrupt each time the host board writes into the first location of the dual-ported memory. The interrupt is cleared when the memory location is read by the slave processor. Finally, the execution of the 8085A-2 SOD instruction generates a MULTIBUS interrupt to inform the host board of the need to service the slave. This interrupt can be vectored to the host interrupt matrix to activate a service routine in support of the slave board.

A detailed explanation of the use of these features is found in another application note, AP-53, *Using the iSBC 544™ Intelligent Communications Controller.* The reader should refer to this document for details of the board and its use.



BAUD	SHIELDED (IN FEET)	UNSHIELDED (IN FEET)
110	5000	3000
300	5000	3000
1200	3000	3000
2400	1000	500
4800	1000	250
9600	250	250

ALL DISTANCES SHOWN ARE MORE THAN 50 FEET/2500 pF, HENCE THESE TRANSMISSIONS VIOLATE RS232C.

NOTE THAT THE "EIA" INTERFACE CASE, SHIELDED PAIR OUT-PERFORMED UNSHIELDED WIRE.

Figure 9. RS232 Baud Rate vs Distance

Multidrop Network Design

The complexity of a multinode system is minimized through the use of a multidrop network. Certain limitations are placed on the acceptable hardware transmission media in multidrop applications. A scheme must be used which allows each of the nodes to alternatively gain control of the network in order to communicate its message.

Both half and full duplex configurations are allowable in a multidrop network. While it is simpler from a hardware standpoint, a half duplex connection requires more stringent communications protocol as this system allows communications in only one direction at a time. For all practical purposes, no priority for masters or slaves is provided. All nodes may listen to whomever is using the line at the time. The software must be written to allow only one node to communicate at a given instant. An example of a half duplex network is the Ethernet.

More straightforward software can be written for full duplex communication configurations. In this instance, an assumption is made that only one master is configured into the system and always drives the output lines. Any number of slaves can be placed to drive the input lines, communicating only when queried by the master node. This is the scheme used for the application example which is discussed in this application note.

Intel's iSBX 351 Serial MULTIMODULE Board lends itself well to a multidrop application. It allows a wide variety of system solutions to be created using any of the single board computers which incorporate iSBX bus connectors. Several board options must be enabled to create the multidrop communications configuration.

The first option involves configuring the drivers to use a tri-state driver on the output lines. This allows only the board desiring to control the communications signals to place data on the serial output lines. The board configuration is different for a slave and for a master node. In the Intel implementation, the driver output signal lines are controlled using the DTR (data terminal ready) signal from the USART. The jumper configuration for both a slave and a master is shown in Figures 10 and 11.

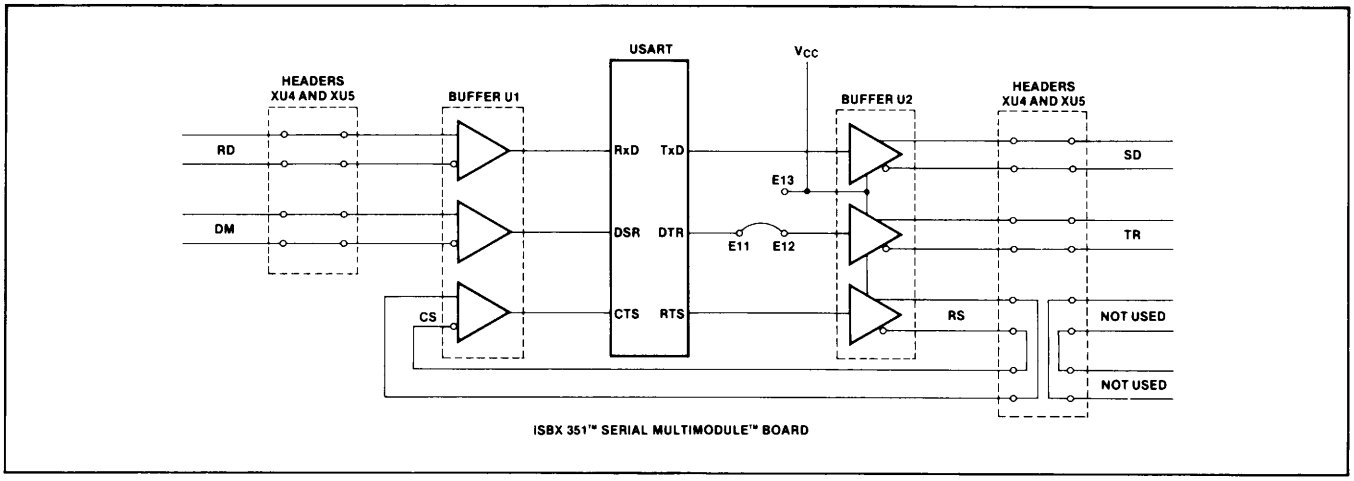


Figure 10. Master Configuration Detail

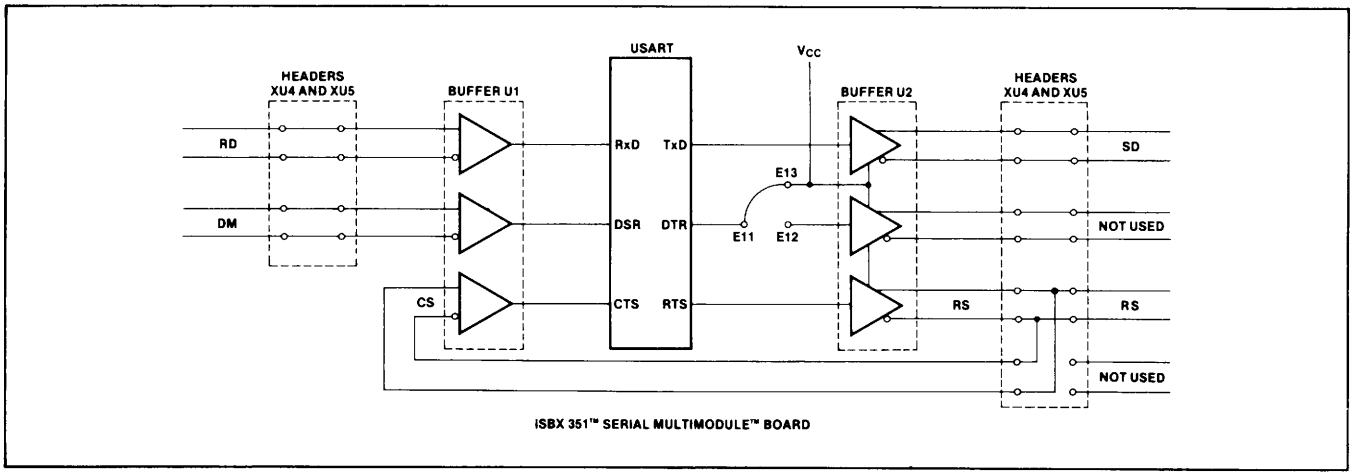


Figure 11. Slave Configuration Detail

Figure 12 illustrates the physical wiring data paths which are used to create a full duplex multidrop network having a master and four slave stations. Each board must be configured using a header block to provide the correct signals. This is done as shown in Figures 10 and 11.

Finally, bias resistors are chosen to terminate the serial communication data paths. A discussion of the formulas for determining the correct values for the two resistors is provided in Appendix A of the *iSBX 351™ Serial MULTIMODULE Board Hardware Reference Manual* (Order Number 9803190). If the equations are solved for the componnts used on the RS422 section of the board, the equations for calculating the correct values become:

$$Rb1 = 4500 / (18.6 - 1.6n)$$

where n is the number of slave nodes

AND

$$Rb2 = 2500 / (18.6 - 1.6n)$$

where n is the number of slave nodes.

The drivers used on the iSBX 351 communications MULTIMODULE board limit the number of slave nodes to 11 for any one network. If greater numbers of nodes are required, a driver component substitution may be required.

For the example application in this note, four slave nodes are used, so the calculated resistor values become 370 ohms for Rb1 and 205 ohms for Rb2.

Figure 13 illustrates the acceptable baud rates for various communications cable lengths using the RS422 interface. The iSBC 351 board is designed to support data transmission rates of up to 19.2 kilobaud. Thus, the system designed using these boards is seen to permit a total multidrop cable length of up to 4000 feet (1.219 km). The sample application discussed in this note incorporated a total communications link length of 1600 feet, well within the performance limits.

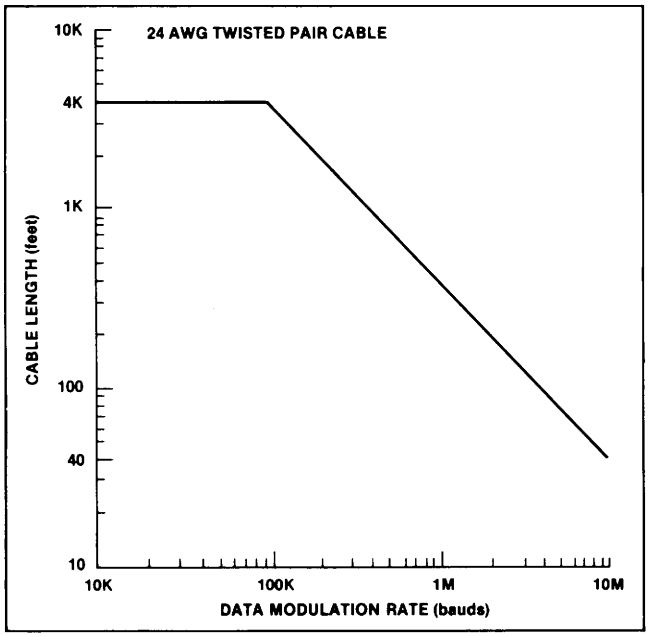


Figure 13. RS422 Data Rate vs Distance

SOFTWARE IMPLEMENTATION

Once the hardware has been defined, the design process moves into a software phase. Here, protocols are defined which provide both data transmission and handshaking between nodes. Handshaking is defined as background communication which maintains synchronization and integrity of the communications link.

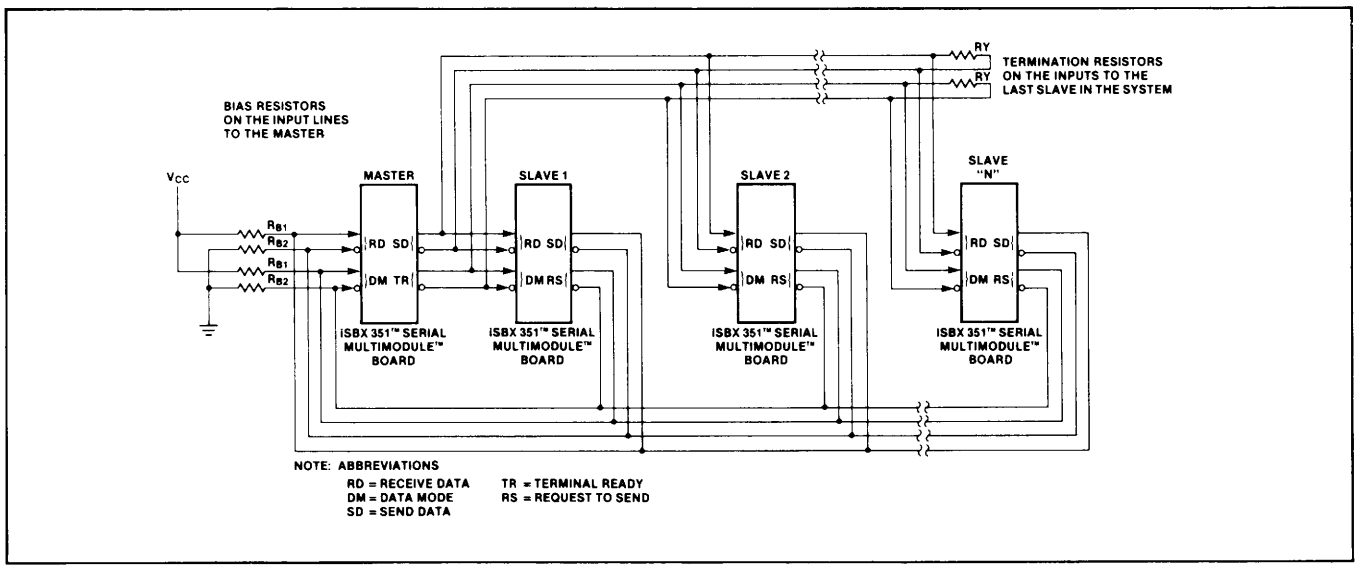


Figure 12. Master/Slave Physical Wiring

Again, standard features of both iSBC board products and of iRMX™ software products simplify the incorporation of communication protocols into the system solution.

Master/Slave Relationships

In order to maintain an orderly flow of data between nodes, software must be created which allocates priority and only allows one node to transmit at any instant. The most straightforward technique which incorporates this function is the master/slave relationship. Here, one node is designated as a master and all others are slaves. Each slave is allocated a time during which it may transmit any information to other nodes. The master node controls the prioritization of all slave nodes. The complexity of the system can be further reduced if a full duplex scheme is implemented. In this mode, all slaves listen only to the master and transmit their messages over a common serial channel to the master. A disadvantage of this technique is that direct slave to slave (virtual channel) transmissions are not permitted.

Figure 14 shows how a master node establishes time slots for windows, during which a slave may control the output channel and transmit data to the master. Each slave node is assigned a unique identification or address.

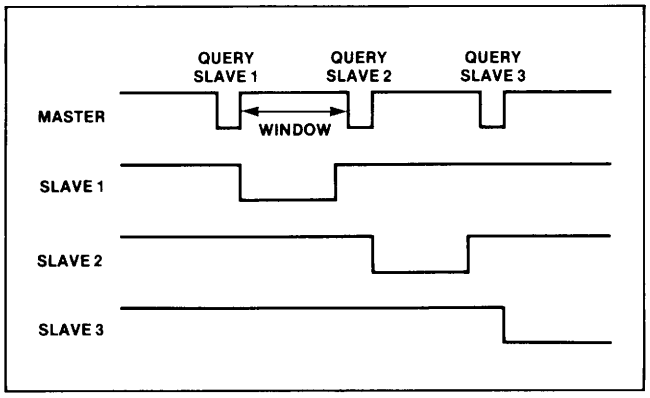


Figure 14. Generation of Windows

Each data packet contains an address as an integral part. When a slave recognizes its address in the packet from the master, it knows that it has fixed time window in which to return its own data packet to the master.

Data Packet Formats

Communications between each slave and its master involve the transmission of data packets. Each packet contains both handshaking information (such as checksums and message counts) and the information which is to pass between nodes. The handshaking portions of the message are used to maintain the integrity of transmissions in the face of such external stimuli as electrical noise.

Debugging and system maintenance is minimized when a means is provided which allows easy viewing and interpretation of the communication messages. The scheme used in this application note uses a fully ASCII compatible communications format. This allows a CRT terminal to tap onto the link and to monitor all communication messages.

Each message packet begins and ends with a unique character. This character cannot exist with a message block. Using the ASCII formats, all messages use the numeric representation of 0 to 9 and the alpha characters from A to Z. A line feed is used to begin a message packet and a carriage return is used to terminate the message. The assignment of these characters assures an easily interpreted message packet.

The layout of the data packet is shown in Figure 15. The justification and description of each field is given in the following paragraphs. Note that much of the message is concerned with maintaining system integrity. The requirement for an address field has already been described. Two characters are used for this field and provide for 256 addresses (0–FF hex). The protocol being described used 0 as the system master and 1 through 255 for possible slave addresses.

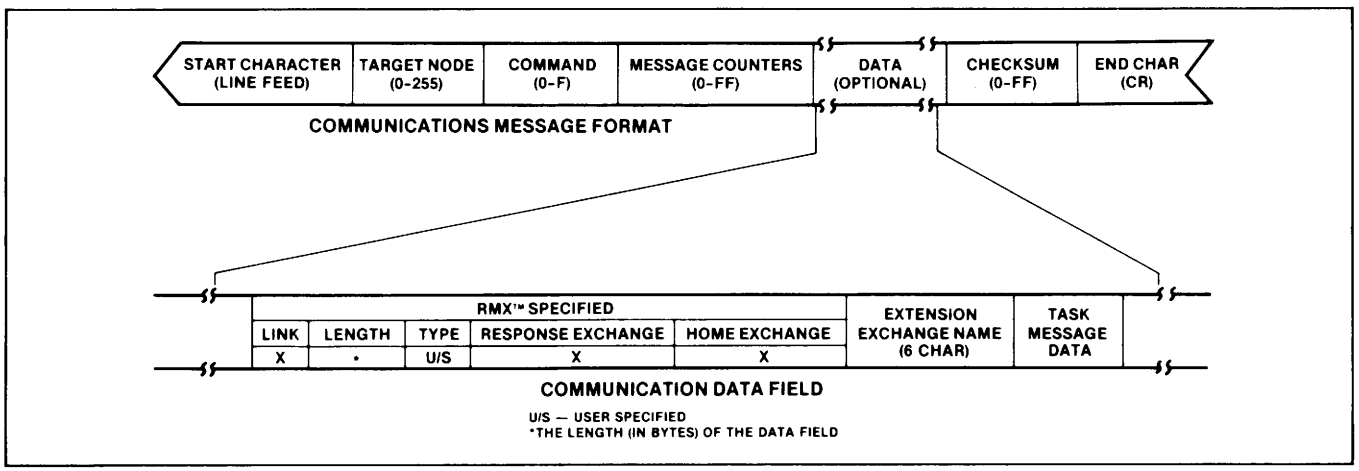


Figure 15. Communications Packet Format

The next field indicates the type of message being sent and the action which is required by the receiver. Analysis of system communication requirements indicate that five message types are sufficient. The use of a single byte of the packet for this field allows a total of 16 message types. This leaves 11 for future expansion. The types being used in this application are:

Type 0 — This is a reset request from the master to a slave. It will cause the target slave to generate a software reset. This command is used only when communications cannot be established using other means.

Type 1 — A type 1 command is used to synchronize the master and the slave. It will cause the message counters (see subsequent paragraphs) to reset to zero. This command is issued only by the master node.

Type 2 — The type 2 message is defined to be that which contains information which is to be moved between nodes of the communications network. The exact format of the data is discussed elsewhere. This is the only message which is not considered to be of a strictly handshaking nature.

Type 3 — A type 3 message is used to indicate that a network node is responsive and ready to handle messages. When normal communications have been established and no data messages are being sent, this message is continuously being sent between the master and the slaves.

Type 5 — The final message type is the type 5. It is used by the slave to respond to the master's request to synchronize. The receipt of this message indicates that the slave has performed all necessary operations with its message counters and is ready to receive messages.

The next field is used to describe the number of messages which contain data and have been received or sent by each node. The first character is used to indicate the number of messages which have been received from the master node to that slave. The second character indicates the number of messages which have been sent from the master to the node. Internal software protocol enables the communications package to use these fields to determine if a data message has been received correctly by the target node. If a response message does not contain the correct counts, the message can be retransmitted. Since only one character is used for each count, the numbers are allowed to recycle each 15 messages.

The data field contains information which is to be sent to the receiving node. Its exact content is discussed later in the application note during the discussion of the iRMX 80 message extensions.

A checksum is included to guarantee the integrity of the transmitted message. This field contains the 8-bit modulo 256 sum of all transmitted ASCII characters, beginning with the line feed and continuing through the last character of the data field. It is used to compare a calculated checksum of received characters with that transmitted by the sending node. If these two numbers are in agreement, the message is considered to be valid and can be used by the receiving node's application software.

The final field is the end of message character. A carriage return is used and provides a unique indicator of the end of a message packet.

The implementation of these communication concepts into a functioning software package is illustrated later during the discussion of the layering of the multiprocessing communications package.

Multitasking Message Concepts

The design of systems which involve distributed processors strongly suggests that a multitasking environment is also present. The ability to segregate an application into tasks allows a considerable simplification of the design and implementation process. In reality, few tasks represent completely isolated functions. Some degree of communications is required between the tasks. This may range from a simple synchronization of tasks to a data transfer. Real-time executives simplify these processes for the system designer. Intel's iRMX 80 nucleus is an excellent example of such an executive. It provides low overhead, small physical size, and operates on almost all 8-bit Intel single board computers. The product has been thoroughly covered in several application notes and it is therefore not necessary to cover it again in this note. However, certain features are useful for developing a distributed processor extension of the basic nucleus.

An iRMX 80 message is analogous to a letter which is mailed to someone. It is sent via a mailroom or post office (called an exchange) and is then picked up by the receiving party. Each part of the iRMX implementation can be thought of in this manner. When a serial communications link is used between the processors, the analogy translates to that of sending a mailgram. The following short discussion deals with the structure and mechanism for sending messages between tasks which reside on different single board computers. For clarity, the mail analogy is used.

There are several distinct parts of a message. Each has a function and a format within the message structure. Before a message can be generated, a medium must be procured to contain that message. In terms of a large office, this medium is paper; in the multitasking system, it is a block of RAM. In order to maintain a continuing supply of memory, the iRMX executive provides the

user with what is known as a free space manager. The purpose of the free space manager is to recycle memory blocks so that they can be again used for generation of additional messages. This service is called each time the sending task requires to generate a message. When the communication programs have successfully transmitted the message to the target node, the memory block is again returned to the free space memory pool. As the target node receives the message it must obtain a block of memory from its free space manager to store the information until it can be processed by the target task. The target task returns the block to the pool when it has taken the appropriate actions.

As with any type of message, the individual to whom the message is to be delivered must be provided. Because the iRMX 80 nucleus is designed to be used by multiple tasks on the same processor, the target address mechanism is not included in the message header itself; instead, the target is specified in the call to the iRMX procedure as a parameter list component. When the target exchange address is not known, which happens when multiprocessor applications are created, it is necessary to create an extension to the executive which allows the assignment of a target exchange name to each message. Fortunately, this is relatively easy to create.

iRMX 80™ Named Exchange Extension

The iRMX 80 nucleus uses a small block of RAM memory to store data regarding the characteristics and status of each exchange. This storage area normally occupies 10 bytes of memory. Its layout is shown in Figure 16. Except to note that a unique identification field does not exist which sets each exchange descriptor apart, it is beyond the scope of this application note to dwell on the meanings of the fields. However, a method must be created which allows the extension of the field to include an identification which is unique to that descriptor.

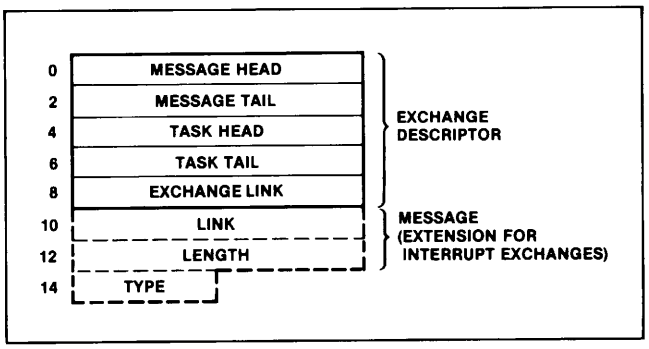


Figure 16. Exchange Descriptor

If one is not to rewrite the nucleus, the format of the exchange descriptor must not be changed and the identification field should be added as additional bytes of the memory block. Two features of the iRMX 80 nucleus make the direct implementation of this concept impossible. First, a special exchange descriptor already exists within the standard real-time executive. This is the interrupt exchange which adds 5 bytes to the nucleus for use as an interrupt message. Second, if an additional extension were to be added, all exchanges would have to be created as interrupt exchanges so that common software could be used. While this would not in itself be prohibitive, the interactive configurator (ICU 80) does not allow addition of fields to either the standard or to the interrupt exchange descriptor. Another method is required to add the extension.

Fortunately, another method exists to create an exchange. The nucleus operation, RQCXCH, creates an exchange at an address which is specified by a parameter of the call instruction. If user software is created to build the named exchanges, a name can be prefixed to each desired name exchange.

In the standard iRMX 80 nucleus six characters are allowed to name each task. A logical extension of this concept provides a six-character name for each named exchange. The exchange descriptor is thus extended by inserting 6 bytes before the basic format. In realty, this is not sufficient because named exchanges are to be created dynamically. The memory blocks representing the set of named exchanges to not necessarily constitute a contiguous block of RAM. This leads to the creation of an additional word of memory to carry a pointer to the next named exchange field. A value of zero in this field indicates that no additional named exchanges exist. If a non-zero value is placed into the field, it is a pointer to the next named exchange. Figure 17 illustrates the structure of the named exchange fields.

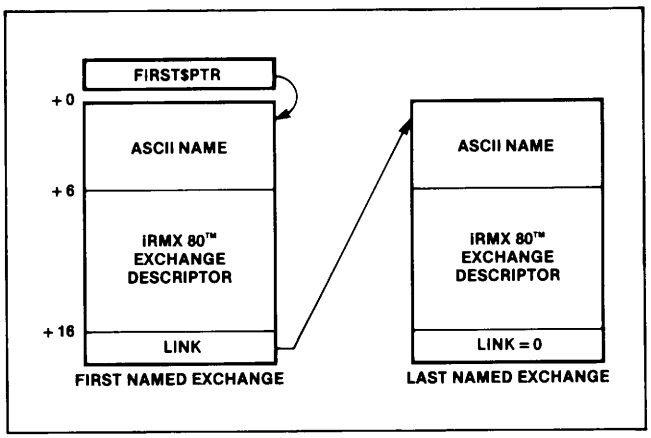


Figure 17. Named Exchange Descriptors

The creation of the software to support the named exchange extension is rather straightforward. Two functions are required; one which creates the named exchanges, and a second which will return the address of an exchange which matches a given name. A complete listing of the software generated to perform these functions is shown in Appendix A. The services of the free space manager are used to allocate the memory segments which are used for the exchange descriptors. A

quick examination of the program techniques provides some insight into the operations involved in creating extensions to the standard nucleus.

Examination of the listing at line 55 shows that a procedure is used to allow user code to determine the absolute address of the iRMX exchange descriptor field from the named exchange lists. This address can in turn be used with either the RQWAIT or the RQSEND instruction in the user's code. If no corresponding name is found in the table, a value of zero is returned by the procedure. An address parameter is passed in the user call to the FIND$EXCH subroutine which points to the 6 bytes containing the ASCII name of the referenced exchange.

An internal address value, identified as FIRST$PTR, contains the address of the first named exchange descriptor. A simple DO loop sequence is used to locate a matching name in the table lists. Either a zero or the location of an actual 10-byte exchange descriptor within the extended memory block is returned.

A task is used to provide the system capability of building named exchanges. A task, rather than a procedure, is included because a task allows the initialization of system pointers such as FIRST$PTR. The named exchange building task, CREATE$COM, waits at its input exchange, COM$CREATE$EXCH until a request for creation of a named exchange is received from a user task. The listing for this task is found in Appendix A, beginning on line 78.

When a message is received, the task obtains a 20-byte block of memory from the free space manager. It then moves the ASCII name from the requesting message into the appropriate fields and uses the iRMX primitive call, RQCXCH, to create an exchange descriptor within the memory block. The pointer field of the last named exchange is updated to point to the new memory field and the pointer field of the newly created named exchange is set to zero.

Finally, the address of the actual exchange descriptor is returned to the requesting program as an updated parameter of the request message.

The creation of named exchanges greatly enhances the capabilities of systems designed around the iRMX 80 nucleus. This extension allows the generation of distributed systems in which tasks may communicate with each other regardless of their geographical locations so long as some type of link exists.

Generation of Multiprocessing Serial Communications Link

The creation of software for a serial communications link provides the capability of allowing true multitasking, multiprocessor capabilities. The need for two types of communications packages, a slave and a master, indicates that two separate tasks are required. A comprehensive examination of the communication requirements indicates that the system can be generated in three layers. Figure 18 shows the layer relationships. The first is defined as the protocol and consists of that code which is required to implement the algorithms for either the slave or the master network nodes. The second level is known as the link level and contains that code which is used to support common operations such as message generation and data queue handling. The third provides a specialized interface to the physical hardware which is being used to generate the communications link. This level, called the physical level, is unique to each configuration.

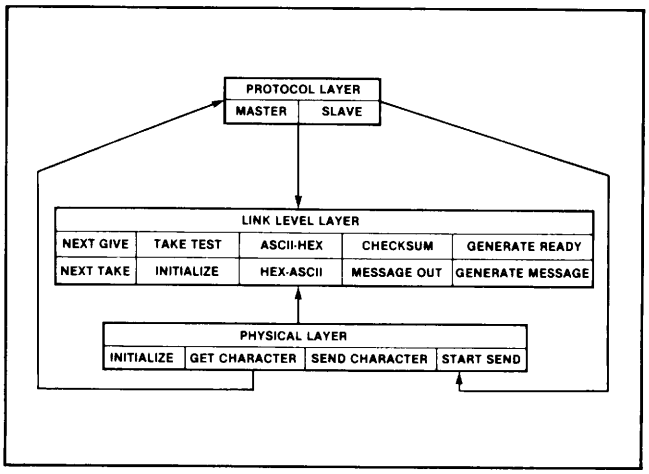


Figure 18.

PROTOCOL LEVEL COMMUNICATIONS PACKAGE

Two protocol level communications packages are included in this application note. One implements the master protocol and is reproduced in Appendix B. The second, the slave protocol, is found in Appendix C.

No attempt is made to detail the operations which are involved in the task generation. The code is commented and is readily followed by the reader who has an understanding of PL/M programming techniques. Some facits relating to the implementation of the packages as iRMX 80 tasks are in order.

The master communications package is designed to use a USART device which is configured to provide interrupts at level 7 each time a character is received. A physical level interrupt handler supports the receipt of each individual character, and, when an end of message character (carriage return) is received, places an interrupt message into the interrupt exchange, RQL7EX. At this point (Appendix B, line 244), the master protocol handler can begin processing the received message. Because the task is associated with interrupt level 7, a task priority in the range between 113 and 128 must be assigned to the task. The example used in this application note uses a priority of 113.

In a similar manner, the slave communications task uses interrupt level 6 to receive messages (Appendix C, line 130). Its priority must lie in the range between 97 and 112.

LINK LEVEL COMMUNICATIONS PACKAGE

The link level communications package provides a common set of procedures which can be used to support both the protocol and the physical layers. Listings of these support programs are found in Appendix D. The programs which make up the package are defined in the following discussion.

Several procedures support the maintenance of the data queues. These are CQQINIT, CQ$NEXT$TAKE, and CQ$NEXT$GIVE. The first, CQQINIT, is used to clear space for a data queue. It sets up the length parameter and the give and take pointers to their initial values.

Data is placed onto the queue by calling the procedure, CQ$NEXT$GIVE, and including the desired data in the paramter list. Conversely, CQ$NEXT$TAKE is used to take data from the queue. Both maintain the queue pointers as data is inserted or removed. For further information about data queues, the reader should refer to AP-52, *Using the iSBC 544™ Intelligent Communications Controller*. This document contains an extensive discussion on the subject of data queues.

Two procedures deal with the transformation of data between printable ASCII and the internal binary format. CQASCHEX transfers data from the data queue into a working buffer. In the process, it converts the format from ASCII into a hex representation usable by the target task. Likewise, CQHEXASC is used to transform data in a user buffer into a transmittable format. In both cases, appropriate start and stop characters are added or deleted as necessary to create the correct format.

One procedure deals with computing the checksum of a message which is in ASCII format. This procedure, CQ$CHECKSUM, returns a zero if the checksum agrees with that in the message. If an error is indicated, a value of −1 (0FF hex) is returned.

Finally, two procedures support the generation of message packets. One, CQGENRDY, is used to build a "ready" message by creating the appropriate data into the fields. The second, CQGENMSG, generates a data message containing an iRMX 80 message to be sent to an exchange on another processor board.

PHYSICAL LEVEL COMMUNICATIONS PACKAGE

The physical level communications package provides the customization for the unique configuration of host processor and USART. Four procedures must be included with this level. These are:

- **CQ$INIT** — This public procedure contains all operations necessary to initialize the timers, counters and USARTs associated with the communications code. In addition, this procedure defines the interrupt service routines for the USART and enables the corresponding interrupt levels.
- **CQ$START$MSG** — A public procedure which places the USART into a mode in which the transmitter is enabled. The execution of this procedure should result in an interrupt being generated by the USART transmitter ready line.
- **CQMIVT** — This is an interrupt service routine associated with the USART receiver ready line. It is entered each time that a character has been received by the communication node. In the case of a slave node, this procedure is known as CQSIVT. The name is unimportant because the location of the routine is passed to the iRMX 80 nucleus at initialization by the CQ$INIT program. When a carriage return (end of message) is encountered, a message is passed to the protocol level task by passing a message to the interrupt exchange, RQL6EX, using the iRMX primitive, RQISND.
- **SEND$CHAR** — Like the CQMIT routine, this is an interrupt handler procedure. It is entered each time the USART signals that it is ready to transmit a character. A new character is obtained from the data queue and it is transmitted to the receiving node. If the character is the end of message, flags are set and the USART transmitter is disabled.

The listing of a sample physical driver is provided in Appendix E. This driver illustrates the use of the iSBX 351 Serial MULTIMODULE Board placed into a socket of an iSBC 80/24 Single Board Computer.

The example from the application implements a multidrop slave node. The enabling of the tri-state drivers is accomplished in line 138 by sending the USART a command of 025H. When the message has been sent, the driver is again placed into a high impedance mode by sending a command of 026H (line 121). These commands control the driver by toggling the DTR or Data Terminal Ready lines of the 8251A USART device.

APPLICATION EXAMPLE

The alarm and security system previously discussed provides a perfect environment in which to use the concepts developed in this application note.

To verify the functionality of the communications package, the transmissions between a master and a slave were monitored using a CRT terminal. The terminal was

connected to one of the RS232 serial paths using a tee network. This tee network is shown in Figure 19. Several scenarios were set up to test the effectiveness of the communications protocol. The results of some of these tests are recorded in the following discussion.

A test of normal communications was performed in which one message is passed between the master and the slave. A short time later, a message is passed to the master from the slave. The CRT display for this action displayed as:

(line 1) 0330000
(line 2) 00300FD
(line 3) 03200091230900BF000000009F
(line 4) 00310FE
(line 5) 0331001
(line 6) 0021082470900870000000008A
(line 7) 0331102
(line 8) 00311FF

Line 1 is a "ready" (character 3) message from the master node to a slave addressed 03 hex (characters 1 and 2). No messages have been sent in either direction at the time this message was generated. On line 2, the slave has responded indicating that it is present and ready to receive messages. It also has not transmitted any messages to the master nor has it received any.

The master node sends the slave a message on line 3. Note that the message count field (characters 4 and 5) is not changed until after the message has been transmitted. The message sent is a type BF and has a total length of 9 bytes. On line 4, the slave acknowledges that is has received one message and has sent none.

Lines 5, 6, and 7 illustrate a sequence in which the slave sends a message to the master node. Prior to the message being sent, the message count field shows that one message has been received. After receiving the message, the master node increments its received counter.

Tests with the alarm and security system verify that the communication module functions correctly. Data integrity is maintained even through intentional disruption of the electrical link. Both the RS232 and RS422 communication paths function as designed.

A detailed discussion of this application and of the multitasking capabilities of the iRMX 80 nucleus can be found in AP-112, *Simplifying Complex Designs Using The iRMX 80™ Executive*.

CONCLUSIONS

This application note illustrates the ease with which a user can add extensions to the iRMX 80 executive to support a distributed multiprocessing application. In addition, the user of standard "off the shelf" single board computers and expansion modules to create a hardware solution is explained.

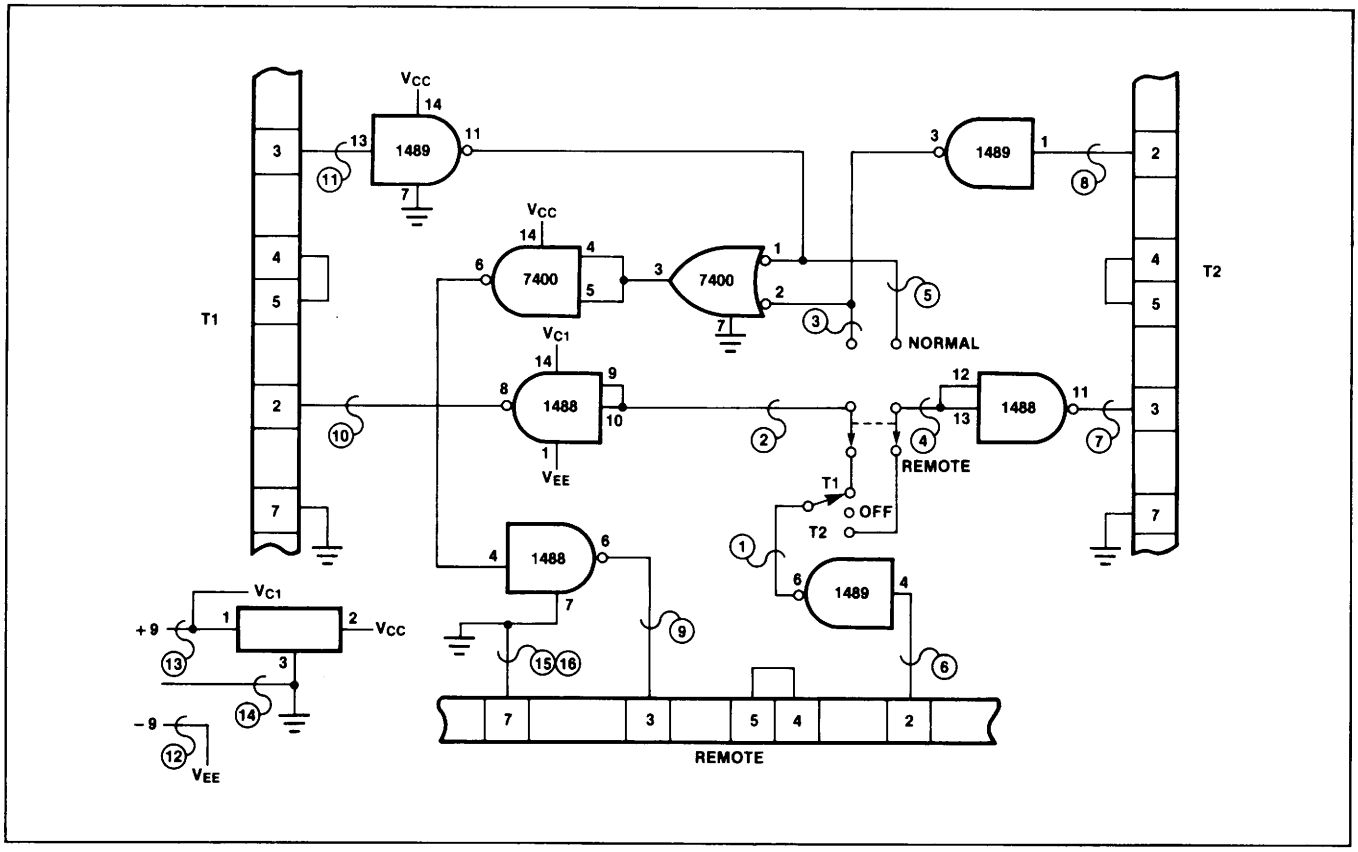


Figure 19. Communications Tee Schematic

The ability to break applications into small tasks, either functionally or geographically, aids the system designer by allowing his concentration on each task. A message transfer capability allows these tasks to again be tied together to form a complete solution to the application. Where the tasks reside on the same single board computer, the standard iRMX nucleus provides this ability. When different host boards are used, extensions to the nucleus allow the same functions to be performed. Both multiple processors on the same MULTIBUS chassis (refer to AP-88 and AP-112) and in different chassis using the serial links described in this application note can be supported with extensions to the nucleus.

Many concepts are used to create serial communication networks. Intel's single board computer products simplify the design process. Among these products, several stand out in this application. For example:

- **iRMX 80™ Executive** — This outstanding product simplifies the design of multitasking systems and provides a foundation for the implementation of extensions to create many varied configurations. Such things as task synchronization, interrupt handling, and message transfers allow multitasking. The free space manager allocates RAM and is an integral part of the serial communicaitons link software. The terminal handler and the disk operating system simplify the software design by providing ready to use I/O drives which can be accessed by the user.

- **iSBC 80/10B™ Single Board Computer** — This microcomputer allows a low cost solution to many small to medium applications. The ability to operate the board in an iRMX 80 environment enhances its capabilities by allowing it to multiprocess. Its onboard serial communications link allows it to be used as a slave node in either EIA or current loop communications networks. The iSBX MULTIMODULE connector further enhances its capabilities by allowing customization of I/O to meet a wide variety of user applications.

- **iSBC 544™ Intelligent Communications Board** — The use of this board allows much of the protocol and physical drivers to be off-loaded from the host processor. The board provides an ideal master communicatons node for star type networks. By placing the handshaking operations on the communication board, the host is free to perform application-oriented functions with a much higher throughput rate.

- **iSBC 80/24™ Single Board Computer** — This powerful 8085A-2 based microcomputer board provides the capability to implement most of the system functions without the need to expand to additional memory or I/O expansion MULTIBUS boards. It fully supports the iRMX 80 executive. Its ability to act as a full bus master allows even greater flexibility through the implementation of MULTIBUS multiprocessor solutions. The two iSBX MULTIMODULE expansion connectors provide the user with considerable flexibility in the design of his system. The use of one of these sockets as a carrier for the serial communications MULTIMODULE board makes a multidrop serial communications link practical.

- **iSBX 351™ Serial MULTIMODULE Board** — The implementation of multidrop communications requires the use of an electrical interface which supports more than one communicaitons node to share the link. The use of the RS422 operating mode of the board easily implements this feature. A well-defined hardware protocol also assists in the implementation of a serial multidrop communications link. Up to 11 slave nodes can be designed into a system using this powerful board. In addition, the iSBX 351 board can function in either the RS232C or the EIA mode for linking into terminals or for creating a point to point network.

The assistance of Judy McMillan in the implementation and testing of the security system and its communication links is greatly appreciated.

```
                $title('RMX/80 EXTENSION TO NAME EXCHANGES, VERSION 1.3')
 1              create$com$exch:
                do;
               /*************************************************************
               CREAT$EXCHANGE  creates a list of exchanges associating
               the name of an exchange with its location, for the purpose
               of other tasks wishing to find the location of an exchange.
               FIND$EXCHANGE, when given the name of an exchange, polls
               down the list and, when it finds a match, returns the
               address to the requesting task.
               *************************************************************/
               $nolist

                 /* declare exchanges used by the task */

26   1           declare COM$CREATE$EXCH exchange$descriptor public;
27   1           declare CREATE$EXCH     exchange$descriptor public;

                  /* declare pointers and messagges used by the task */

28   1           declare first$ptr address;
29   1           declare last$ptr  address;
30   1           declare msg$ptr   address;
31   1           declare exch$ptr  address;
32   1           declare n         byte;

33   1           declare request based msg$ptr structure(
                     msg$hdr,
                     exch$name(6) byte);

34   1           declare fs$req structure(
                     msg$hdr,
                     msg$length  address);

35   1           declare exch based exch$ptr structure(
                     name(6)      byte,
                     exchange(10) byte,
                     link         address);

36   1           declare last$exch based last$ptr structure(
                     name(6)      byte,
                     exchange(10) byte,
                     link         address);
37   1           declare response based msg$ptr structure (
                     msg$hdr,
                     exchange address );

38   1           NAMEX:
                 Procedure(exptr) address public;
39   2                declare addr address;
40   2                declare exptr address;
41   2                declare (ex based exptr) (6) byte;

                 /* declare the returning message with the address */
```

```
42   2      declare ret$msg based addr  structure(msghdr,
                exch$adr    address);

            /* declare message to send to create a named exchange */

43   2      declare req structure(msg$hdr,
                name(6) byte);

            /* create an exchange to communicate with
               the comm create exchange */

44   2       declare fsx exchangedescriptor;
45   2       call rqcxch(.fsx);

             /* build and send the request message */

46   2         req.length = 15;
47   2         call move(6, .ex, .req.name);
48   2         req.type = 200;
49   2         req.response$exchange  = .fsx;
50   2         call rqsend(.comcreate$exch, .req);
51   2         addr = rqwait(.fsx, 0);
52   2         addr = ret$msg.exch$adr;

             /* return with the address of the exchange */

53   2         return(addr);
54   2       end;

55   1       FIND$EXCH:
          procedure (name$ptr) address public;

          /*****************************************************
          This procedure finds the exchange having a name specified
          by the passed parameter pointer.  If a match is found the
          exchange address is returned.  If no match is found, a
          zero is returned.
          *****************************************************/

56   2           declare name$ptr address;
57   2           declare match byte;
58   2           declare (name based name$ptr)(6) byte;

                /* test for no exchanges */
59   2           if first$ptr = 0
                then return 0;

                /* test for a name match */
61   2           else do;
62   3               exch$ptr = first$ptr;
63   3               do while 1;
64   4                    match = 0ffh;
65   4                    do n = 0 to 5;
66   5                         if name(n) <> exch.name(n)
```

```
                                then match = 0;
68   5                        end;
69   4                        if match then return .exch.exchange;
71   4                        if exch.link = 0 then return 0;
73   4                        exch$ptr = exch.link;
74   4                    end;
75   3               end;
76   2           return 0;
77   2           end;

78   1           CREATE$COM:
                 procedure public;

                 /*************************************************************
                 This procedure creates exchange extensions when asked to
                 do so by a task that wants its exchange made completely
                 accessible.  It saves the name of the exchange then
                 creates an exchange to save its address. It may be updated
                 at any time.
                 *************************************************************/

                      /* initialize exchanges */
79   2                   call rqcxch(.com$create$exch);
80   2                   call rqcxch(.create$exch);

                      /* initialize pointers */
81   2                   last$ptr = 0;
82   2                   first$ptr = 0;

83   2                   do while 1;

                            /* get a request for creation of an exchange */
84   3                      msg$ptr = rqwait(.com$create$exch, 0);

                            /* get some memory for the exchange */
85   3                      fs$req.length = 11;
86   3                      fs$req.type   =  5;
87   3                      fs$req.response$exchange = .create$exch;
88   3                      fs$req.msg$length = 20;
89   3                      call rqsend(.rqfsax, .fs$req);
90   3                      exch$ptr = rqwait(.create$exch, 0);
91   3                      exch$ptr = fs$req.msg$length;

                            /* store name in the exchange structure */
92   3                      call move( 6, .request.exch$name(0),
                                 .exch.name(0));

                            /* create an exchange */
93   3                      call rqcxch(.exch.exchange(0));

                            /* update the link field */

94   3                      if last$ptr > 0
                            then do;
96   4                           last$exch.link = exch$ptr;
```

```
 97    4                    last$ptr = exch$ptr;
 98    4                    exch.link = 0;
 99    4                end;
100    3                else do;
101    4                    exch.link = 0;
102    4                    first$ptr = exch$ptr;
103    4                    last$ptr = exch$ptr;
104    4                end;

                        /* return the exchange pointer */
105    3                response.exchange = .exch.exchange;
106    3                call rqsend(request.response$exchange, msg$ptr);

107    3            end;
108    2        end;
109    1        end;

MODULE INFORMATION:
     CODE AREA SIZE     = 01D3H     467D
     VARIABLE AREA SIZE = 0048H      72D
     MAXIMUM STACK SIZE = 0004H       4D
     244 LINES READ
     0 PROGRAM ERROR(S)
```

```
   $title('Master Communications Protocol Driver, Version 8.1')
1            MASTER$DRIVER: do;
             /*
             The task contained in this listing supports the master
             communications protocol for extablishing network commun-
             ications.

             The task must be configured using the ICU/8C as follows:
                     TASK NAME: MASTER
                     TASK ENTRY POINT: MLINK
                     TASK PRIORITY: 113
                     TASK STACK LENGTH: 1CC
                        .
                              .
                        .
                     EXCHANGE: RQL7EX
                       SCOPE: EXTERNAL
                       INTERRUPT: YES
                     EXCHANGE: TIMEX
                               SCOPE: PUBLIC
                               INTERRUPT: NO
                        .
                     LINK: :Fn:CQMSTR.OBJ
                        .
                        .
                        .
                     LINK: :Fn:CQCOM.LIB
                     LINK: :Fn:CQRSLV.LIB

             Tasks desiring to send a message to a node should
             send a message to the exchange CQMIEX(node).

                 Certain include files are used by the task.
             */
             $include (:f0:exch.elt)
2    1  =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        =     MSG$HEAD ADDRESS,
        =     MSG$TAIL ADDRESS,
        =     TASK$HEAD ADDRESS,
        =     TASK$TAIL ADDRESS,
        =     NEXT$EXCH ADDRESS)';
             $include (:f0:ied.elt)
3    1  =    DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        =     MESSAGE$HEAD ADDRESS,
        =     MESSAGE$TAIL ADDRESS,
        =     TASK$HEAD ADDRESS,
        =     TASK$TAIL ADDRESS,
        =     EXCHANGE$LINK ADDRESS,
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE)';
              $include (:f0:msg.elt)
4    1  =    DECLARE MSG$HDR LITERALLY '
```

```
          =     LINK ADDRESS,
          =     LENGTH ADDRESS,
          =     TYPE BYTE,
          =     HOME$EXCHANGE ADDRESS,
          =     RESPONSE$EXCHANGE ADDRESS';
          =
 5    1   =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
          =     MSG$HDR,
          =     REMAINDER(1) BYTE)';
                $include (:f0:common.elt)
 6    1   =    DECLARE TRUE LITERALLY '0FFH';
 7    1   =    DECLARE FALSE LITERALLY '00H';
 8    1   =    DECLARE BOOLEAN LITERALLY 'BYTE';
 9    1   =    DECLARE FOREVER LITERALLY 'WHILE 1';
                $include (:f0:synch.ext)
10    1   =    RQSEND:
          =      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
11    2   =        DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
          =
12    2   =      END RQSEND;
          =
13    1   =    RQWAIT:
          =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
14    2   =        DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
          =
15    2   =      END RQWAIT;
          =
16    1   =    RQACPT:
          =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
17    2   =        DECLARE EXCHANGE$POINTER ADDRESS;
          =
18    2   =      END RQACPT;
          =
19    1   =    RQISND:
          =      PROCEDURE (IED$PTR) EXTERNAL;
20    2   =        DECLARE IED$PTR ADDRESS;
          =
21    2   =      END RQISND;
                $include (:f0:intrpt.ext)
22    1   =    RQENDI:
          =      PROCEDURE EXTERNAL;
          =
23    2   =      END RQENDI;
          =
24    1   =    RQELVL:
          =      PROCEDURE (LEVEL) EXTERNAL;
25    2   =        DECLARE LEVEL BYTE;
          =
26    2   =      END RQELVL;
          =
27    1   =    RQDLVL:
          =      PROCEDURE (LEVEL) EXTERNAL;
28    2   =        DECLARE LEVEL BYTE;
          =
29    2   =      END RQDLVL;
```

```
                =
30    1   =     RQSETV:
              =       PROCEDURE (PROC,LEVEL) EXTERNAL;
31    2   =         DECLARE PROC ADDRESS;
32    2   =         DECLARE LEVEL BYTE;
              =
33    2   =       END RQSETV;
              =
34    1   =     RQSETP:
              =       PROCEDURE (PROC,LEVEL) EXTERNAL;
35    2   =         DECLARE PROC ADDRESS;
36    2   =         DECLARE LEVEL BYTE;
              =
37    2   =       END RQSETP;
                $include (:f0:fsmgr.ext)
38    1   =     DECLARE RQFSAX EXCHANGE$DESCRIPTOR EXTERNAL;
39    1   =     DECLARE RQFSRX EXCHANGE$DESCRIPTOR EXTERNAL;
40    1   =     DECLARE RQFREE EXCHANGE$DESCRIPTOR EXTERNAL;
                $include (:f1:mesmsg.elt)
41    1   =         declare msg$format literally 'structure (
              =             trgt      byte,
              =             cmnd      byte,
              =             seq$lo    byte,
              =             seq$no    byte,
              =             text(250) byte )';
              =
42    1   =         declare queue$format literally 'structure (
              =             end$ptr         byte,
              =             give$ptr        byte,
              =             take$ptr        byte,
              =             data$byte(254)  byte )';
              =
43    1   =         declare par$format literally 'structure (
              =             no        byte,
              =             lo        byte,
              =             run       byte,
              =             stop      byte,
              =             que$pt    byte )';
                $include (:f0:objman.ext)
44    1   =     RQCTSK:
              =       PROCEDURE (STATIC$PTR) EXTERNAL;
45    2   =         DECLARE STATIC$PTR ADDRESS;
              =
46    2   =       END RQCTSK;
              =
47    1   =     RQCXCH:
              =       PROCEDURE (EXCHANGE$PTR) EXTERNAL;
48    2   =         DECLARE EXCHANGE$PTR ADDRESS;
              =
49    2   =       END RQCXCH;
                $include (:f1:comcom.ext)
50    1   =     CQ$NEXT$GIVE:
              =         Procedure (q$ptr,give$byte) byte external;
51    2   =         Declare q$ptr address;
52    2   =         Declare give$byte byte;
```

```
53    2  =        end cq$next$give;
54    1  =    CQ$TAKE$TEST:
          =         Procedure (q$ptr) byte external;
55    2  =        Declare q$ptr address;
56    2  =        end cq$take$test;
57    1  =    CQ$NEXT$TAKE:
          =        Procedure (q$ptr) byte external;
58    2  =        Declare q$ptr address;
59    2  =        end cq$next$take;
60    1  =    CQ$Q$INIT:
          =        Procedure (q$ptr,q$size) external;
61    2  =        Declare q$ptr address;
62    2  =        Declare q$size byte;
63    2  =        end cq$q$init;
64    1  =    CQ$ASC$HEX:
          =        Procedure (q$ptr,msg$ptr) external;
65    2  =        Declare (q$ptr,msg$ptr) address;
66    2  =        end cq$asc$hex;
67    1  =    CQ$CHECKSUM:
          =        Procedure (q$ptr) byte external;
68    2  =        Declare q$ptr address;
69    2  =        end cq$checksum;
70    1  =    CQ$HEX$ASC:
          =        Procedure (q$ptr,hex$byte) external;
71    2  =        Declare q$ptr address;
72    2  =        Declare hex$byte byte;
73    2  =        end cq$hex$asc;
74    1  =    CQ$MSG$OUT:
          =        Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
75    2  =        Declare msg$size byte;
76    2  =        Declare (q$ptr,par$ptr,msg$ptr) address;
77    2  =        end cq$msg$out;
78    1  =    CQ$GEN$RDY:
          =        Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
79    2  =        Declare trget byte;
80    2  =        Declare (msg$ptr,q$ptr,par$ptr) address;
81    2  =        end cq$gen$rdy;
82    1  =    CQ$GEN$MSG:
          =        Procedure (msg$pointer,trget,msg$ptr,q$ptr,
                              par$ptr,save$flg) external;
83    2  =        Declare (trget,save$flg) byte;
84    2  =        Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
85    2  =        end cq$gen$msg;

86    1       USRINT:
                  Procedure external;
87    2         end usrint;
88    1       FIND$EXCH:
                  Procedure(name$ptr) address external;
89    2         declare name$ptr address;
90    2         end find$exch;
91    1       NAMEX:
                  Procedure(exptr) address external;
92    2         declare exptr address;
93    2         end namex;
```

```
94   1      return$ram:
                procedure(pointer) external;
95   2          declare pointer address;
96   2      end return$ram;

            /*

                Certain literals are used to define the network's
                physical characteristics. These are:
            */
97   1          Declare N$NODE literally '4';     /* number of nodes */
98   1          Declare NODE$ARRAY literally '5';

            /*
                A structure provides the data queues for the
                transmission of data to each node. It is defined
                and is available as a public
            */
99   1          Declare CQMSTQ queue$format public at (811ch);

            /*
                A single data queue is used to support the input of
                data from a node since only one slave is given a
                window at a time:
            */
100  1          Declare CQMSIQ queue$format public at (801bh);

            /*
                Each node has an associated set of flags which
                indicate the operational mode of that node. The
                function of each bit is defined as:
                    bit 0 - request synchronization (synch$request)
                    bit 1 - request initialization (init$request)
                    bit 2 - repeat last message (repeat$request)
                    bit 3 -
                    bit 4 -
                    bit 5 -
                    bit 6 -
                    bit 7 - error flag (error$flag)
            */
101  1          Declare CQCMDF(node$array) byte public;
102  1          Declare SYNCH$REQUEST literally '01H';
103  1          Declare INIT$REQUEST literally '02H';
104  1          Declare REPEAT$REQUEST literally '04H';
105  1          Declare ERROR$FLAG literally '80H';


            /*
                A counter is used to indicate the number of attempts
                to establish communications with a node. When it
                reaches a maximum value, a synch will be sent to the
                node. when the maximum value of synchs are sent to a
                node with no effect, a reset will be sent to the node.
            */
```

```
106   1          Declare ERROR$COUNT(node$array) byte;
107   1          Declare MAX$COUNT literally '5';
108   1          Declare SYNCH$COUNT(node$array) byte;

                 /* a pointer is used to store the address of exchanges
                  used byincomming messages*/

109   1          Declare target$exch address;
               /*
                  A set of exchanges is used to hold a message until it
                 has been acknowledged by the slave.
               */
110   1          Declare HOLD$EXCH(node$array) exchange$descriptor;

               /*
                 A set of exchanges is provided to accept messages which
                 are to be sent to any of the nodes.
               */
111   1          Declare COMIEX(node$array) exchange$descriptor public;

               /*
                 The task uses various sets of data structures
               */
112   1          Declare req$msg structure (
                     msg$hdr,
                     msg$length address );
113   1          Declare (m,n,node$cnt,outmode,ram$size) byte;
114   1          Declare msg$ptr address;
115   1          Declare msg msg$format;
116   1          declare start$544 byte public at (8000h);
117   1          Declare data$block based msg$ptr structure (
                     msg$hdr);
118   1          Declare COMPAR(node$array) par$format public
                     at (8002h);
119   1          Declare RAM$MSG based msg$ptr structure (
                     msg$hdr);
120   1          Declare CQ$ACTIVE$NODE byte public at (8001h);
121   1              Declare fre$mem address;
122   1              Declare fre$msg based fre$mem structure(msg$hdr);
               /*
                 The task uses certain exchanges for internal
                 communications
               */
123   1          Declare COMSEX exchange$descriptor;
124   1          Declare RQL7EX int$exchange$descriptor public;
            $eject
            /*************************************************************
                                        error$test
            This short procedure is used to increment the error
            counters and to signal an initialization or synch command
            when required.  It will order a re-transmission of the
            last message each time an error is detected.

            *************************************************************/
```

```
125   1       error$test: procedure;

126   2           If (error$count(n):=error$count(n)+1) > max$count
                  then do;
128   3               error$count(n)=0;
129   3               if (synch$count(m):=synch$count(m)+1) > max$count
                      then do;
131   4                   synch$count(m) = 0;
132   4                   cqcmdf(n) = cqcmdf(n) or error$flag
                                      or init$request;
133   4               end;
134   3               else cqcmdf(n) =cqcmdf(n)
                                      or error$flag or synch$request;
135   3           end;
136   2           else cqcmdf(n) = cqcmdf(n) or repeat$request;
137   2           return;
138   2       end error$test;
             $eject
139   1       MLINK: Procedure public;

                  /* Initialize the system and the task */
140   2           Do n=0 to N$NODE;
141   3               CQCMDF(n)=synch$request or error$flag;
142   3               CQMPAR(n).run,CQMPAR(n).stop,CQMPAR(n).que$pt=0;
143   3               ERROR$COUNT(n)=0;
144   3               SYNCH$COUNT(n)=0;
145   3           end;

                  /* Initialize the node pointer */
146   2           NODE$CNT=0;

                  /* Initialize the input exchanges */
147   2           Do N=0 to n$node;
148   3               call RQCXCH(.CQMIEX(n));
149   3               call RQCXCH(.HOLD$EXCH(n));
150   3           end;

151   2               call RQCXCH(.CQMSEX);

                  /* Initialize the queues */
152   2           call cq$q$init(.cqmstq, 254);
153   2           call cq$q$init(.cqmsiq, 254);


                  /* Create the named exchanges and give ram to fsm*/
154   2           call usrint;

                  /* initialize the level 7 interrupt exchange */
155   2           call rqelvl(7);

                  /* Begin main task loop */
156   2           Do forever;

                  /* Begin support of one nodal channel */
157   3               Do n=1 to n$node;
```

```
                        /* reset queue pointers */
158   4                 camstq.give$ptr, camstq.take$ptr = 0;

                        /* Compute nodal mode number */
159   4                 If (cacmdf(n) and synch$request)>0
                        then outmode = 1;
161   4                 else outmode = 0;
162   4                 If (cacmdf(n) and init$request)>0
                        then outmode = 2;
164   4                 ca$active$node = n;

                        /* if comm failure, kill all messages */
165   4                 if (cacmdf(n) and error$flag) > 0
                        then do;
167   5                     do while (msg$ptr:=raacpt(.camiox(n))) > 0;
168   6                         call return$ram(msg$ptr);
169   6                     end;

170   5                     do while (msg$ptr:=raacpt(.hold$exch(n)))
                                > 0;
171   6                         call return$ram(msg$ptr);
172   6                     end;
173   5                 end;

                        /* Operate on nodal mode */
174   4                 Do case outmode;

                            /* case 0, routine communications */
175   5                     Do;
176   6                         If (cacmdf(n) and repeat$request)>0
                                then do;

                                /* support of retransmission request */
178   7                         cacmdf(n)=cacmdf(n)
                                        and not repeat$request;
179   7                             msg$ptr=raacpt(.hold$exch(n));
180   7                             if msg$ptr>0

                                /* retransmit old message */
                                then do;
182   8                                if campar(n).na = 0
                                       then campar(n).na = 15;
184   8                                else campar(n).na=campar(n).na
                                            - 1;
185   8                                call ca$gen$msg(msg$ptr,n,.msg,
                                           .camstq,.campar(n),0);
186   8                                call raaend(.hold$exch(n),
                                           msg$ptr);
187   8                                if campar(n).na = 15
                                       then campar(n).na = 0;
189   8                                else campar(n).na
                                            =campar(n).na+1;
190   8                             end;
```

```
                        /* no old message, send ready */
191   7                 else do;
192   8                     msg.trgt=n;
193   8                     msg.cmnd=3;
194   8                     call cq$msg$out(f,.cqmstq,
                                .cqmpar(n),.msg);
195   8                 end;
196   7           end;   /* end of retransmission */

                  /* support of next message */
197   6           else do;

                        /* clear hold exchange */
198   7                 msg$ptr=rqacpt(.hold$exch(n));
199   7                 if msg$ptr>0
                        then do;

                            /* free space return size */
201   8                     ramsize=data$block.length;
202   8                     if (ramsize mod 4) > 0
                            then data$block.length
                              =data$block.length
                              +(4-(ram$size mod 4));
204   8                     call RQSEND(.rqfsrx,msg$ptr);
205   8                 end;

                        /* test for a message output */
206   7                 msg$ptr=rqacpt(.cqmiex(n));

                        /* when a message exists */
207   7                 if msg$ptr>0
                         then do;
209   8                     if (target$exch:
                                =find$exch(msg$ptr +9))> 0
                            then call
                               rqsend(target$exch,msg$ptr);
211   8                     else do;
212   9                     call cq$gen$msg(msg$ptr,n,.msg,
                                .cqmstq,.cqmpar(n),0);
213   9                     call rqsend(.hold$exch(n),
                                msg$ptr);

                            /* increase the na flag */
214   9                     if cqmpar(n).na = 15
                            then cqmpar(n).na = 0;
216   9                     else cqmpar(n).na
                                = cqmpar(n).na + 1;

217   9                 end;
218   8                 end;

                        /* when no message exists */
219   7                 else do;
220   8                     msg.trgt=n;
221   8                     msg.cmnd=3;
```

```
222   8                        call cq$msg$out(0,.camstq,
                                   .campar(n),.msg);
223   8                    end;
224   7                end;   /* end of routine message */

                       /* start the message transmission */
225   6                start$544 = n;

226   6            end;                    /* end of case 0 */

                   /* case 1, synch request */
227   5            do;

228   6                campar(n).na,campar(n).la = 0;
229   6                msg.trgt=n;
230   6                msg.cmnd=1;
231   6                call cq$msg$out
                           (0,.camstq,.campar(n),.msg);
232   6                start$544 = n;
233   6                cacmdf(n)= cacmdf(n)
                           and not synch$request;

234   6            end;

                   /* case 2, initialize request */
235   5            do;

236   6                cacmdf(n)=(cacmdf(n) and
                       not init$request) or synch$request;
237   6                campar(n).na, campar(n).la = 0;
238   6                msg.trgt=n;
239   6                msg.cmnd=0;
240   6                call cq$msg$out
                           (0,.camstq,.campar(n),.msg);
241   6                start$544 = n;
242   6            end;

243   5        end;           /* end of do case blocks */

               /* wait for a response from the slave */
244   4        msg$ptr = rqwait(.rq17ex,25);
245   4        start$544 = 0;

               /* test for a message or a timeout */
246   4        if ram$msg.type=3

               /* support of no valid response */
               then
                   /* test for max number of errors to node */
247   4            call error$test;

               /* support of good response return */
248   4        else do;

               /* test for good checksum */
```

```
249   5                 if ca$checksum(.cqmsiq)=0
                        then do;

                            /* convert message to hex format */
251   6                     call ca$asc$hex(.comsiq,.msg);

                            /* test for data message receipt */
252   6                     if msg.cmnd = 2

                            /* support receipt of data message */
                            then do;

                                /* get ram from free space manager */
254   7                         req$msg.length=11;
255   7                         req$msg.type = 5;
256   7                         req$msg.response$exchange=.ca$ms$ex;
257   7                         req$msg.msg$length=msg.text(2);
258   7                         call rqsend(.rqfsax,.req$msg);
259   7                         msg$ptr=rqwait(.ca$ms$ex,0);
260   7                         msg$ptr=req$msg.msg$length;

                                /* move message into memory */
261   7                         call move(msg.text(2),.msg.text(0)
                                        ,msg$ptr);

                                /* if target is another node.. */
262   7                         if msg.trgt > 0
                                then call rqsend(.cqmiex(msg.trgt)
                                        ,msg$ptr);

                                /* if target is master board */

264   7                         if (target$exch:= find$exch(msg$ptr +
9))
                                        > 0
                                then
265   7                             call rqsend(target$exch, msg$ptr);
266   7                         else do;
267   8                             ramsize = msg.text(2);
268   8                             if (ramsize mod 4) > 0
                                    then msg.text(2)=msg.text(2)
                                      + (4 -(ramsize mod 4));
270   8                             call rqsend(.rqfsrx, msg$ptr);
271   8                         end;


                                /* increment message counter */
272   7                             if cqmpar(n).la = 15
                                    then cqmpar(n).la = 0;
274   7                             else cqmpar(n).la=cqmpar(n).la+1;


275   7                             end;
                            /* respond to non-sequence acknowledge */
276   6                     if msg.cmnd = 5
```

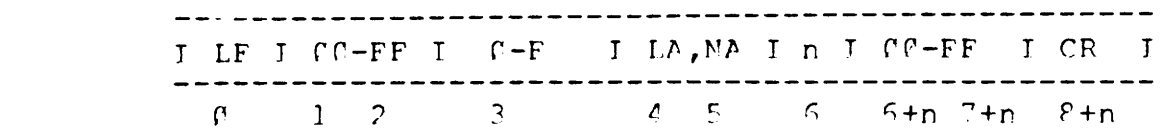
```
                                then cam$par(n).la,cam$par(n).na = 0;

                                /* test for a good slave LA response */
278   6                         if campar(n).na = msg.seqla + 1
                                then call error$test;
280   6                         else do;
281   7                              if msg.seqla <> campar(n).na
                                     then cacmdf(n) = cacmdf(n)
                                               or synchrequest;
283   7                              else do;
284   8                                   error$count(n) = 0;
285   8                                   cacmdf(n) = cacmdf(n)
                                               and not error$flag;
286   8                              end;
287   7                         end;
288   6                   end;    /* end of response return */
                          /* support of bad checksum */
289   5                   else call error$test;
290   5                   end;

291   4                   end;    /* end of message arrived options */
292   3             end;                          /* end of one node */
293   2         end;                                 /* end of task */
294   1   end
```

```
                $title('Slave Communications Package  Version 3.2')
  1             CQMSLV$MODULE: do;

                /*
                    This is the slave communication main task.
                    It should be configured into the system
                    having the following parameters:

                        Task name- CQSLAV
                        Task entry point- CQSLAV
                        Task priority- as required
                        Task stack length- 100


                    The LINK and LOCATE commands of the user
                    must include the following statements:

                        Link ...
                        ...
                        :Fn:CQSLAV.OBJ
                        :Fn:CQS351.OBJ
                        :Fn:CQCOM.LIB
                        :Fn:CQRSLV.LIB
                        ...

                    Several system parameters must be defined
                    in the user code to define the mode
                    characteristics. These parameters are defined
                    as:
                        cq$slad- a byte value providing the
                            nodal address of the commun-
                            ication node.
                        fre$mem- an address value which provides
                                a pointer to a block of RAM to
                                be assigned via the free space
                                manager to the communications.
                        ram$len- an address value which indicates
                                the size of the above block.
                */
  2   1         declare cqslad  byte external;

                $nolist
                $include(:f1:commsg.elt)
 48   1   =        declare msg$format literally 'structure (
      =                 trgt      byte,
      =                 cmnd      byte,
      =                 seq$la    byte,
      =                 seq$na    byte,
      =                 text(250) byte )';
      =
 49   1   =        declare queue$format literally 'structure (
      =                 end$ptr         byte,
      =                 give$ptr        byte,
      =                 take$ptr        byte,
```

```
                =            data$byte(254) byte )';
                =
50    1         =        declare par$format literally 'structure (
                =            la         byte,
                =            na         byte,
                =            run        byte,
                =            stop       byte,
                =            que$pt     byte )';
                    $include(:f0:objman.ext)
51    1         =    RQCTSK:
                =      PROCEDURE (STATIC$PTR) EXTERNAL;
52    2         =        DECLARE STATIC$PTR ADDRESS;
                =
53    2         =      END RQCTSK;
                =
54    1         =    RQCXCH:
                =      PROCEDURE (EXCHANGE$PTR) EXTERNAL;
55    2         =        DECLARE EXCHANGE$PTR ADDRESS;
                =
56    2         =      END RQCXCH;
                    $include(:f1:comcom.ext)
57    1         =    CQ$NEXT$GIVE:
                =        Procedure (q$ptr,give$byte) byte external;
58    2         =        Declare q$ptr address;
59    2         =        Declare give$byte byte;
60    2         =        end cq$next$give;
61    1         =    CQ$TAKE$TEST:
                =        Procedure (q$ptr) byte external;
62    2         =        Declare q$ptr address;
63    2         =        end cq$take$test;
64    1         =    CQ$NEXT$TAKE:
                =        Procedure (q$ptr) byte external;
65    2         =        Declare q$ptr address;
66    2         =        end cq$next$take;
67    1         =    CQ$Q$INIT:
                =        Procedure (q$ptr,q$size) external;
68    2         =        Declare q$ptr address;
69    2         =        Declare q$size byte;
70    2         =        end cq$q$init;
71    1         =    CQ$ASC$HEX:
                =        Procedure (q$ptr,msg$ptr) external;
72    2         =        Declare (q$ptr,msg$ptr) address;
73    2         =        end cq$asc$hex;
74    1         =    CQ$CHECKSUM:
                =        Procedure (q$ptr) byte external;
75    2         =        Declare q$ptr address;
76    2         =      . end cq$checksum;
77    1         =    CQ$HEX$ASC:
                =        Procedure (q$ptr,hex$byte) external;
78    2         =        Declare q$ptr address;
79    2         =        Declare hex$byte byte;
80    2         =        end cq$hex$asc;
81    1         =    CQ$MSG$OUT:
                =        Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
82    2         =        Declare msg$size byte;
```

```
 83   2  =        Declare (q$ptr,par$ptr,msg$ptr) address;
 84   2  =        end cq$msg$out;
 85   1  =    CQSGEN$RDY:
         =        Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
 86   2  =        Declare trget byte;
 87   2  =        Declare (msg$ptr,q$ptr,par$ptr) address;
 88   2  =        end cq$gen$rdy;
 89   1  =    CQ$GEN$MSG:
         =        Procedure (msg$pointer,trget,msg$ptr,q$ptr,par$ptr,sav
          -    e$flg) external;
 90   2  =        Declare (trget,save$flg) byte;
 91   2  =        Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
 92   2  =        end cq$gen$msg;
 93   1       Declare RQL6FX int$exchange$descriptor public;
 94   1       Declare CQSIEX exchange$descriptor public;
 95   1       Declare cmrqex exchange$descriptor;
 96   1       declare mold$slave$ex exchange$descriptor public;
 97   1       Declare cqtime exchange$descriptor public;

 98   1       cq$start$msg$351:
                 procedure external;
 99   2           end cq$start$msg$351;

100   1       cq$init$351:
                 procedure external;
101   2           end cq$init$351;

102   1       find$exch:
                 procedure(name$ptr) address external;
103   2           declare name$ptr address;
104   2       end find$exch;

105   1       cq$startup:
                 procedure external;
106   2       end cq$startup;

107   1       declare cq$in$sl queue$format public;
108   1       declare cq$out$sl queue$format public;
109   1       declare msg msg$format;
110   1       declare cq$pars par$format public;
111   1       declare fre$mem address;
112   1       declare cqcmdf(5) byte external;

113   1       declare fre$msg based fre$mem structure (
                     msg$hdr );
```

```
/*
    The message which is transmitted between nodes
    follows the description below:

    The complete message structure is defined as:

      STX  TARGET  COMMAND   SEQ  TEXT CHECKSUM  EOT
```



With the exception of the start of text (a line feed) and the end of transmission (a carriage return), all characters transmitted in the message string will consist of printable ASCII characters.

The target field consists of two frames which represent the hex address of the device to which the message is addressed. The protocol allows for up to 256 unique device addresses.

The command field indicates the type of message which is being sent in the network. It consists of a single frame representing a hex number in the range from 0 to 0FFH. The current message definitions are:

command	label
0	INIT
1	SYNCH
2	DATA
3	READY
4	NOT READY
5	NON SEQ ACK
6	reserved
.	"
.	"
.	"
F	"

This is the main link level RMX/80 communications task which supports slave operations of a single board computer. Several high level functions are supported by this task.

Messages containing data may be sent or received by this task. To send a message to another processor on the communications loop, a message of the format shown is placed into the communications exchange, CQSIEX. When the message has been transmitted, the RAM area in which the message was located will be returned to the free space manager.

Messages may also be received by the board when they are addressed to the communications address assigned to this board. When a message has been received, it will be transferred to a block of RAM obtained from the free space manager and then sent to an exchange which is designated in the name field of the message.

```
                The format for a data message is:
                    msg$hdr  targetexchangename    message

                Data may be up to 250 bytes in length.

                A special command, INIT, can be received by the
                communications driver task which will cause a
                software reset of the single board computer.

            */

114   1     CC$SLAV: Procedure public;

115   2         declare target$exch address;
116   2             declare name        address;
117   2         Declare msg$ptr     address;
118   2         declare ramsize     address;

119   2         Declare RAM$msg based msg$ptr structure (
                    msg$hdr );

120   2         Declare req$msg structure (
                    msg$hdr,
                    msg$length address );
                /* initialize the task at power up */
121   2         call cq$q$init(.cq$in$sl, 250);
122   2         call cq$q$init(.cq$out$sl, 250);
123   2         cq$pars.run, cq$pars.stop, cq$pars.que$pt = 0;

                /* build required exchanges */
124   2         call rqcxch(.rql6ex);
125   2         call rqcxch(.cqsiex);
126   2         call rqcxch(.cmrqex);
127   2         call rqcxch(.cqtime);
128   2         call cqinit$351;

129   2         do forever;

                /* wait for a window from the master */
130   3         msg$ptr = rqwait (.RQL6EX, 400);

                /*test for good communications with iMOS*/

131   3         if ram$msg.type <> 3 then do;

                    /* test for the receipt of a valid message */
133   4             if cq$checksum(.cq$in$sl) = 0
                    then do;

                        /* convert the message into hex format */
135   5                 call cq$asc$hex(.cq$in$sl,.msg);

                        /* see if message is for this slave board */
136   5                 if msg.trgt = caslad
                        then do;
```

```
138   6                    name = .msg.text + 9;
139   6                    cqcmdf(0) = cqcmdf(0) and 7fh;

                           /* handle each message type by case */
140   6                    do case msg.cmnd;

                               /* case 0, INIT command */
141   7                        call cqstartup;

                               /* case 1, SYNCH command */
142   7                        do;

                                   /* reset counters */
143   8                            cq$pars.la,cq$pars.na = 0;

                                   /* initailize data queues */
144   8                            call cq$q$init (.cq$in$sl,250);
145   8                            call cq$q$init (.cq$out$sl,250);
                                   /* return non seq ack message */
146   8                            msg.trgt = 0;
147   8                            msg.cmnd = 5;
148   8                            call cq$msg$out (0,.cq$out$sl,.cq$
          -    pars,.msg);
149   8                            call cq$start$msg351;
150   8                        end;

                               /* case 2, DATA command */
151   7                        do;

                                   /* test for correct NA */
152   8                            if cqpars.na = msg.seq$na
                                   then do;

                                       /* clear old messages */
154   9                                msg$ptr = rqacpt(.hold$slave$e
          -    x);
155   9                                if msg$ptr > 0
                                       then call return$ram(msg$ptr);

                                       /* increment LA counter */
157   9                                if (cq$pars.la:=cq$pars.la+1)
          -    > 15
                                       then cq$pars.LA = 0;

                                       /*verify that exchange exists*
          -    /

159   9                                TARGET$EXCH = FIND$EXCH(name);
160   9                                if TARGET$exch > 0
                                       then do;

                                           /* get RAM and store messa
          -    ge */
162  10                                    req$msg.length = 11;
163  10                                    req$msg.type = 5;
```

```
164  1C                                          req$msg.response$exchange
          -    = .cm$rq$ex;
165  1C                                          req$msg.msg$length = msg.t
          -    ext(2);
166  1C                                          call RQSEND (.rqfsax,.req$
          -    msg);
167  1C                                          msg$ptr = RQWAIT (.cm$rq$e
          -    x, 0);
168  1C                                          msg$ptr = req$msg.msg$leng
          -    th;
169  1C                                          call move (msg.text(2),.ms
          -    g.text(0), msg$ptr);
170  1C                                          call RQSEND (target$exch,
          -    msg$ptr);
171  1C                                      end;

                                             /* test for data out request *
          -    /
172   9                                      msg$ptr = RQACPT (.cqsiex);
173   9                                      if msg$ptr = 0
                                             then call cq$gen$rdy (0,.msg,.
          -    cq$out$sl,.cq$pars);
175   9                                      else do;
176  1C                                          if( target$exch:= find$exc
          -    h(msgptr + 9)) > 0
                                                  then call rqsend(target$ex
          -    ch, msg$ptr);
178  1C                                          else do;
179  11                                             call cqgen$msg(msgptr,
          -    0,.msg,.cq$out$sl,.cq$pars,0);
180  11                                             call rqsend(.hold$slav
          -    e$ex, msg$ptr);
181  11                                          end;
182  1C                                      end;
183   9                                  end;

                                         /* if bad na, then retransmit last
          -     msg */
184   8                                  else do;
185   9                                      cqpars.na = msg.seq$na;
186   9                                      msg$ptr = rqacpt(.hold$slave$e
          -    x);
187   9                                      if msg$ptr > 0
                                             then do;
189  1C                                          call cq$gen$msg(msg$ptr,0,
          -    .msg,.cq$out$sl,.cq$pars,0);
190  1C                                          call rqsend(.hold$slave$ex
          -    , msg$ptr);
191  1C                                      end;
192   9                                  end;
```

```
                                        /* send message */
193   8                                 call cq$start$msg$351;
194   8                             end;

                                      /* case 3, RDY command */
195   7                               do;

                                        /* verify na is correct */
196   8                                 if cq$pars.na = msg.seq$na
                                        then do;

                                          /* clear old messages */
198   9                                   msg$ptr = rqacpt(.hold$slave$e
          -   x);
199   9                                   if msg$ptr > 0
                                          then call return$ram(msg$ptr);

                                          /* test for a message waiting
          -   to be sent */
201   9                                   msg$ptr = RQACPT (.cqsiex);
202   9                                   if msg$ptr = 0
                                          then call cq$gen$rdy (0,.msg,.
          -   cq$out$sl,.cq$pars);
204   9                                   else do;
205   10                                      if (target$exch:=find$exch
          -   (msg$ptr + 9)) > 0
                                              then call rqsend(target$ex
          -   ch, msg$ptr);
207   10                                      else do;
208   11                                        call cq$gen$msg(msg$pt
          -   r,0,.msg,.cq$out$sl,.cq$pars,0);
209   11                                        call rqsend(.hold$slav
          -   e$ex,msg$ptr);
210   11                                      end;
211   10                                  end;

212   9                               end;

                                      /* if bad na, then retransmit last
          -    */
213   8                               else do;
214   9                                   cqpars.na = msg.seq$na;
215   9                                   msg$ptr = rqacpt(.hold$slave$e
          -   x);
216   9                                   if msg$ptr > 0
                                          then do;
218   10                                      call cq$gen$msg(msg$ptr,0,
          -   .msg,.cq$out$sl,.cq$pars,0);
219   10                                      call rqsend(.hold$slave$ex
          -   , msg$ptr);
220   10                                  end;
221   9                               end;

                                      /* start transmission */
```

```
222   8                         call ca$start$msg$351;
223   8                     end;

                            /* case 4, NONRDY command */
224   7                     do;end;

                            /* case 5, NONSEQACK command */
226   7                     do;end;

                            /* case 6, reserved */
228   7                     do;end;

                            /* case 7, reserved */
230   7                     do;end;

                            /* case 8, reserved */
232   7                     do;end;

                            /* case 9, reserved */
234   7                     do;end;

                            /* case A, reserved */
236   7                     do;end;

                            /* case B, reserved */
238   7                     do;end;

                            /* case C, reserved */
240   7                     do;end;

                            /* case D, reserved */
242   7                     do;end;

                            /* case E, reserved */
244   7                     do;end;

                            /* case F, reserved */
246   7                     do;end;

248   7                 end; /* do case */
249   6             end; /* good target */
250   5         end; /* good checksum */
251   4     end; /*good communications with iMOS*/

          /* clear out messages if com failure exists */
252   3     else do;
253   4         do while (msg$ptr:=rqacpt(.casiex)) > 0;
254   5             call return$ram(msg$ptr);
255   5         end;

256   4         do while (msg$ptr:=rqacpt(.hold$slave$ex)) > 0;
257   5             call return$ram(msg$ptr);
258   5         end;

                /* set failure indicator */
```

```
259   4             cqcmdf(f) = cqcmdf(f) or 80h;
260   4         end;

261   3         end; /* do forever */
262   2     end cc$slav;
263   1     end comslv$module;
```

```
                $title('QUEUE INITIALIZATION PROCEDURE')
 1              Q$INIT$MODULE: Do;

 2    1           declare eom literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =      declare msg$format literally 'structure (
      =              trgt       byte,
      =              cmnd       byte,
      =              seq$la     byte,
      =              seq$na     byte,
      =              text(250)  byte )';
      =
 4    1    =      declare queue$format literally 'structure (
      =              end$ptr         byte,
      =              give$ptr        byte,
      =              take$ptr        byte,
      =              data$byte(254)  byte )';
      =
 5    1    =      declare par$format literally 'structure (
      =              la         byte,
      =              na         byte,
      =              run        byte,
      =              stop       byte,
      =              que$pt     byte )';
                $include(:f0:synch.ext)
 6    1    =    RQSEND:
      =           PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =           DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =         END RQSEND;
      =
 9    1    =    RQWAIT:
      =           PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =           DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =         END RQWAIT;
      =
12    1    =    RQACPT:
      =           PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =           DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =         END RQACPT;
      =
15    1    =    RQISND:
      =           PROCEDURE (IED$PTR) EXTERNAL;
16    2    =           DECLARE IED$PTR ADDRESS;
      =
17    2    =         END RQISND;
                $include(:f0:exch.elt)
18    1    =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =           MSG$HEAD ADDRESS,
      =           MSG$TAIL ADDRESS,
      =           TASK$HEAD ADDRESS,
      =           TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER (1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;
23   1        CQ$Q$INIT: Procedure (queue$ptr, queue$size) reentrant pub
          -     lic;
             /*
                 This procedure initializes the queue pointers to
                 indicate an empty queue.
             */

24   2          Declare queue$ptr address;
25   2          Declare queue$size byte;

26   2          Declare queue based queue$ptr queue$format;

                /* set up the size of the queue into the structure */

27   2              queue.FND$PTR = queue$size - 1;

                /* reset the queue offset pointers */

28   2              queue.GIVE$PTR,queue.TAKE$PTR = 0;

                /* return to calling program */

29   2              return;

30   2        end cq$q^init;
31   1        end q$init$module;
```

```
                $title('NEXT GIVE QUEUE PROCEDURE, VERSION 2.1')
 1              NEXT$GIVE$MODULE: Do;

 2    1           declare eom literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =      declare msg$format literally 'structure (
           =           trgt      byte,
           =           cmnd      byte,
           =           seq$la    byte,
           =           seq$na    byte,
           =           text(250) byte )';
           =
 4    1    =      declare queue$format literally 'structure (
           =           end$ptr         byte,
           =           give$ptr        byte,
           =           take$ptr        byte,
           =           data$byte(254) byte )';
           =
 5    1    =      declare par$format literally 'structure (
           =           la        byte,
           =           na        byte,
           =           cur       byte,
           =           stop      byte,
           =           que$pt    byte )';
                $include(:f0:synch.ext)
 6    1    =    RQSEND:
           =      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =        DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
           =
 8    2    =      END RQSEND;
           =
 9    1    =    RQWAIT:
           =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =        DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
           =
11    2    =      END RQWAIT;
           =
12    1    =    RQACPT:
           =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =        DECLARE EXCHANGE$POINTER ADDRESS;
           =
14    2    =      END RQACPT;
           =
15    1    =    RQISND:
           =      PROCEDURE (IED$PTR) EXTERNAL;
16    2    =        DECLARE IED$PTR ADDRESS;
           =
17    2    =      END RQISND;
                $include(:f0:exch.elt)
18    1    =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
           =     MSG$HEAD ADDRESS,
           =     MSG$TAIL ADDRESS,
           =     TASK$HEAD ADDRESS,
           =     TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;

23   1      CQ$NEXT$GIVE: Procedure(QUEUE$PTR,GIVE$BYTE) byte reentran
          -   t public;
              /*
                  This procedure places a byte into the queue if room
                  exists in the queue for that byte of data. If no room
                  exists, a queue full indicator will be returned by
                  the procedure.
              */


24   2          Declare QUEUE$FULL literally '0FFH';
25   2          Declare QUEUE$OK   literally '000H';

26   2          Declare QUEUE$PTR address;
27   2          Declare (GIVE$BYTE,RSLT) byte;

28   2          declare queue based queue$ptr queue$format;


                /* Test for queue full condition and if it is full,
                   then insert end of message */

29   2              RSLT = queue$ok;
30   2              If (queue.GIVE$PTR+1 > queue.END$PTR)
                    then do;
32   3                  rslt = queue$full;
33   3                  queue.date$byte(queue.give$ptr) = eom;
34   3              end;

35   2              else do;

                        /* Store the byte into the next queue location
          -         */

36   3                  queue.DATA$BYTE(queue.GIVE$PTR) = GIVE$BYTE;

                        /* Increment the give pointer */

37   3                  If ((queue.GIVE$PTR:=queue.GIVE$PTR+1)
```

```
                                > queue.END$PTR)
                            then queue.GIVE$PTR = 0;
39    2                 end;

                        /* Return to calling program with good complete */

40    2                 Return rslt;

41    2           end cq$next$give;
42    1           end next$give$module;
```

```
                $title('GET CHARACTER FROM QUEUE PROCEDURE')
1               NEXT$TAKE$MODULE: Do;

2     1             declare eom literally '0DH';
                $include(:f1:commsg.elt)
3     1   =         declare msg$format literally 'structure (
          =               trgt        byte,
          =               cmnd        byte,
          =               src$la      byte,
          =               seq$no      byte,
          =               text(250)   byte )';
          =
4     1   =         declare queue$format literally 'structure (
          =               end$ptr           byte,
          =               give$ptr          byte,
          =               take$ptr          byte,
          =               data$byte(254)    byte )';
          =
5     1   =         declare par$format literally 'structure (
          =               la          byte,
          =               no          byte,
          =               run         byte,
          =               stop        byte,
          =               que$pt      byte )';
                $include(:f0:synch.ext)
6     1   =     RQSEND:
          =       PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
7     2   =         DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
          =
8     2   =       END RQSEND;
          =
9     1   =     RQWAIT:
          =       PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2   =         DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
          =
11    2   =       END RQWAIT;
          =
12    1   =     RQACPT:
          =       PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2   =         DECLARE EXCHANGE$POINTER ADDRESS;
          =
14    2   =       END RQACPT;
          =
15    1   =     RQISND:
          =       PROCEDURE (IED$PTR) EXTERNAL;
16    2   =         DECLARE IED$PTR ADDRESS;
          =
17    2   =       END RQISND;
                $include(:f0:exch.elt)
18    1   =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =      MSG$HEAD ADDRESS,
          =      MSG$TAIL ADDRESS,
          =      TASK$HEAD ADDRESS,
          =      TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;
23   1        CQ$NEXT$TAKE: Procedure (queue$ptr) byte reentrant public;
             /*
                 This typed procedure gets the next byte from the
                 indicated queue and returns it to the calling
                 program. The queue take pointer is incremented.
             */

24   2           Declare queue$ptr address;
25   2           Declare take$byte byte;

26   2           declare queue based queue$ptr queue$format;

                 /* store next data byte in queue */

27   2               take$byte = queue.DATA$BYTE(queue.TAKE$PTR);

                 /* increment the take pointer to next location */

28   2               If ((queue.TAKE$PTR:=queue.TAKE$PTR+1)
                        > queue.END$PTR)
                    then queue.TAKE$PTR = 0;

                 /* return the data byte to caller */

30   2               return take$byte;

31   2        end cq$next$take;
32   1        end next$take$module;
```

```
                $title('ASCII to HEX CONVERSION PROCEDURE')
 1              ASCS$HEX$MODULE: Do;

 2    1             declare eom literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =        declare msg$format literally 'structure (
      =                  trgt       byte,
      =                  cmnd       byte,
      =                  seq$lo     byte,
      =                  seq$hi     byte,
      =                  text(250)  byte )';
      =
 4    1    =        declare queue$format literally 'structure (
      =                  end$ptr          byte,
      =                  give$ptr         byte,
      =                  take$ptr         byte,
      =                  data$byte(254)   byte )';
      =
 5    1    =        declare par$format literally 'structure (
      =                  lo         byte,
      =                  hi         byte,
      =                  run        byte,
      =                  stop       byte,
      =                  que$pt     byte )';
                $include(:f0:synch.ext)
 6    1    =     RQSEND:
      =       PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =         DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =       END RQSEND;
      =
 9    1    =     RQWAIT:
      =       PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =         DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =       END RQWAIT;
      =
12    1    =     RQACPT:
      =       PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =         DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =       END RQACPT;
      =
15    1    =     RQISND:
      =       PROCEDURE (IED$PTR) EXTERNAL;
16    2    =         DECLARE IED$PTR ADDRESS;
      =
17    2    =       END RQISND;
                $include(:f0:exch.elt)
18    1    =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =      MSG$HEAD ADDRESS,
      =      MSG$TAIL ADDRESS,
      =      TASK$HEAD ADDRESS,
      =      TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;
23   1        cq$next$take:
               procedure (q$ptr) byte external;
24   2          declare q$ptr address;
25   2        end cq$next$take;

26   1        CQ$ASC$HEX: Procedure(q$ptr,msg$ptr) reentrant public;

               /*
                  This procedure transforms ASCII data in the input
                  queue into a hex string in the system message storage
                  area.
               */

27   2          Declare (nchar,char,n) byte;
28   2          Declare (q$ptr,msg$ptr) address;

29   2          Declare msg based msg$ptr msg$format;

                  /* throw away the start of message */
30   2          char = cq$next$take(q$ptr);

                  /* convert message target address */
31   2          char = cq$next$take(q$ptr);
32   2          nchar = cq$next$take(q$ptr);

33   2          if char > 40h
                  then char = char - 37h;
35   2          else char = char - 30h;

36   2          if nchar > 40h
                  then nchar = nchar - 37h;
38   2          else nchar = nchar - 30h;

39   2          msg.trgt = shl(char,4) + nchar;

                  /* convert command byte */
40   2          char = cq$next$take(q$ptr);
41   2          if char > 40h
                  then char = char - 37h;
43   2          else char = char - 30h;
```

```
44   2          msg.cmnd = char;

                /* convert LA sequence */
45   2          char = cq$next$take(q$ptr);

46   2          if char > 40h
                then char = char - 37h;
48   2          else char = char - 30h;

49   2          msg.seq$la = char;

                /* convert NA sequence */
50   2          char = cq$next$take(q$ptr);

51   2          if char > 40h
                then char = char - 37h;
53   2          else char = char - 30h;

54   2          msg.seq$na = char;

                /* do operations until end of message */

55   2          n = 0;
56   2          Do while (char:=cq$next$take(0$ptr)) <> EOM;

                /* get next ASCII character */

57   3             nchar = cq$next$take(0$ptr);

                /* convert to hex format */

58   3             if char > 40H
                   then char = char - 37H;
60   3             else char = char - 30H;

61   3             if nchar > 40H
                   then nchar = nchar - 37H;
63   3             else nchar = nchar - 30H;

64   3             msg.text(n) = shl(char,4) + nchar;
65   3             n = n + 1;
66   3          end;

67   2       end cq$asc$hex;
68   1       end asc$hex$module;
```

```
                $title('HEX TO ASCII CONVERSION PROCEDURE')
 1              HEX$ASC$MODULE: Do;

 2    1             declare eom literally 'ODH';
                $include(:f1:commsg.elt)
 3    1    =        declare msg$format literally 'structure (
      =                  trgt       byte,
      =                  cmnd       byte,
      =                  seq$la     byte,
      =                  seq$na     byte,
      =                  text(250) byte )';
      =
 4    1    =        declare queue$format literally 'structure (
      =                  end$ptr         byte,
      =                  give$ptr        byte,
      =                  take$ptr        byte,
      =                  data$byte(254) byte )';
      =
 5    1    =        declare par$format literally 'structure (
      =                  la         byte,
      =                  na         byte,
      =                  run        byte,
      =                  stop       byte,
      =                  que$pt     byte )';
                $include(:f0:synch.ext)
 6    1    =     RQSEND:
      =       PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =         DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =       END RQSEND;
      =
 9    1    =     RQWAIT:
      =       PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =         DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =       END RQWAIT;
      =
12    1    =     RQACPT:
      =       PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =         DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =       END RQACPT;
      =
15    1    =     RQISND:
      =       PROCEDURE (IED$PTR) EXTERNAL;
16    2    =         DECLARE IED$PTR ADDRESS;
      =
17    2    =       END RQISND;
                $include(:f0:exch.elt)
18    1    =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =      MSG$HEAD ADDRESS,
      =      MSG$TAIL ADDRESS,
      =      TASK$HEAD ADDRESS,
      =      TASK$TAIL ADDRESS,
```

```
         =     NEXT$EXCH ADDRESS)';
              $include(:f0:msg.elt)
19    1  =     DECLARE MSG$HDR LITERALLY '
         =     LINK ADDRESS,
         =     LENGTH ADDRESS,
         =     TYPE BYTE,
         =     HOME$EXCHANGE ADDRESS,
         =     RESPONSE$EXCHANGE ADDRESS';
         =
20    1  =     DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
         =     MSG HDR,
         =     REST (1) BYTE)';

21    1           declare timex exchange$descriptor external;
22    1           declare rqfsrx exchange$descriptor external;

23    1       cq$next$give:
                  procedure(que$ptr,data$byte) byte external;
24    2           declare que$ptr address;
25    2           declare data$byte byte;
26    2       end cq$next$give;

27    1       CQ$HEX$ASC: Procedure (q$ptr,hex$byte) reentrant public;

              /*
                  This procedure is used to convert a hex byte into two
                  ASCII characters and store them into the next two loc-
                  ations of the Q$out data area.
              */
28    2           Declare (high$byte,hex$byte,low$byte,status) byte;
29    2           Declare q$ptr address;

30    2           Declare q$out based q$ptr queue$format;

                  /* separate low and high order bits */

31    2           If (high$byte:=(shr(hex$byte,4) and 0FH)) > 9
                  then high$byte = high$byte + 37H;
33    2           else high$byte = high$byte + 30H;

34    2           If (low$byte:=(hex$byte and 0FH)) > 9
                  then low$byte = low$byte + 37H;
36    2           else low$byte = low$byte + 30H;

                  /* store ASCII conversions into the queue */

37    2           status = cq$next$give(.Q$OUT, high$byte);
38    2           status = cq$next$give(.Q$OUT, low$byte);

39    2           return;
40    2       end cq$hex$asc;
41    1       end hex$asc$module;
```

```
                $title('CHECKSUM CALCULATION PROCEDURE')
 1              CHECKSUM$MODULE: Do;

 2    1            declare com literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =       declare msg$format literally 'structure (
      =                 trgt      byte,
      =                 cmnd      byte,
      =                 seq$lo    byte,
      =                 seq$no    byte,
      =                 text(250) byte )';
      =
 4    1    =       declare queue$format literally 'structure (
      =                 end$ptr         byte,
      =                 give$ptr        byte,
      =                 take$ptr        byte,
      =                 data$byte(254) byte )';
      =
 5    1    =       declare par$format literally 'structure (
      =                 lo        byte,
      =                 no        byte,
      =                 run       byte,
      =                 stop      byte,
      =                 que$pt    byte )';
                $include(:f0:synch.ext)
 6    1    =    RQSEND:
      =           PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =             DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =           END RQSEND;
      =
 9    1    =    RQWAIT:
      =           PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =             DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =           END RQWAIT;
      =
12    1    =    RQACPT:
      =           PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =             DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =           END RQACPT;
      =
15    1    =    RQISND:
      =           PROCEDURE (IED$PTR) EXTERNAL;
16    2    =             DECLARE IED$PTR ADDRESS;
      =
17    2    =           END RQISND;
                $include(:f0:exch.elt)
18    1    =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =          MSG$HEAD ADDRESS,
      =          MSG$TAIL ADDRESS,
      =          TASK$HEAD ADDRESS,
      =          TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;
23   1      CQ$CHECKSUM: Procedure(q$ptr) byte reentrant public;

            /*
                This procedure is used to determine the checksum of
                a message which has been received and stored into
                the Q$in buffer. A checksum will be calculated on all
                characters beginning at the start of text and
                continuing until the checksum bytes are encountered.
                This value will be tested with the value transmitted
                and if they agree, a value of zero will be returned
                to the calling program.  If not in agreement, a
                non-zero value will be returned.
            */

24   2          Declare (ptr,end$ptr,q$ptr) address;
25   2          Declare (checksm,string$sum,char,n) byte;
26   2          Declare last$chars(3) byte;

27   2          declare q$in based q$ptr queue$format;

                /* get queue pointer values for reference */

28   2              PTR = Q$IN.take$ptr;
29   2              END$PTR = Q$IN.end$ptr;

                /* initialize checksum value */

30   2              checksm = 0;

                /* compute checksum of string */

31   2              char = Q$IN.data$byte(ptr);
32   2              do while char <> EOM;
33   3                  checksm = checksm + char;
34   3                  if ptr = end$ptr
                        then ptr = 0;
35   3                  else ptr = ptr + 1;
37   3                  char = Q$IN.data$byte(ptr);
38   3              end;
```

```
                    /* last three characters in the message are
                       not included in the checksum, so they must
                       be subtracted....*/

                    /* first remember the last queue location */
39    2                 if ptr = end$ptr
                        then char = 0;
41    2                 else char = ptr+1;

42    2                 do n = 0 to 2;
43    3                     checksm = checksm - (last$chars(n):=
                                        Q$IN.data$byte(ptr));
44    3                     if ptr = 0
                            then ptr = end$ptr;
46    3                     else ptr = ptr - 1;
47    3                 end;
48    2                 checksm = checksm + last$chars(0);

                    /* convert transmitted checksum into a hex value */

49    2                 Do n = 1 to 2;
50    3                     if last$chars(n) > 40H
                            then last$chars(n) = last$chars(n) - 37H;
52    3                     else last$chars(n) = last$chars(n) - 30H;
53    3                 end;
54    2                 string$sum = shl(last$chars(2),4) + last$chars(1);

                    /* test for validity of transmission */

55    2                 if string$sum = checksm
                        then return 0;

                    /* if bad checksum, clear data queue of message */

57    2                 else Q$IN.take$ptr = char;
58    2                 return -1;

59    2           end co$checksum;
60    1           end checksum$module;
```

```
                $title('GENERATE READY MESSAGE PROCEDURE')
 1              GEN$RDY$MODULE: Do;

 2    1            declare eom literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =       declare msg$format literally 'structure (
      =    =           trgt       byte,
      =    =           cmnd       byte,
      =    =           seq$la     byte,
      =    =           seq$na     byte,
      =    =           text(250) byte )';
      =
 4    1    =       declare queue$format literally 'structure (
      =    =           end$ptr          byte,
      =    =           give$ptr         byte,
      =    =           take$ptr         byte,
      =    =           data$byte(254) byte )';
      =
 5    1    =       declare par$format literally 'structure (
      =    =           la         byte,
      =    =           na         byte,
      =    =           run        byte,
      =    =           stop       byte,
      =    =           que$pt     byte )';
                $include(:f0:synch.ext)
 6    1    =   RQSEND:
      =    =     PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =       DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =     END RQSEND;
      =
 9    1    =   RQWAIT:
      =    =     PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =       DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =     END RQWAIT;
      =
12    1    =   RQACPT:
      =    =     PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =       DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =     END RQACPT;
      =
15    1    =   RQISND:
      =    =     PROCEDURE (IED$PTR) EXTERNAL;
16    2    =       DECLARE IED$PTR ADDRESS;
      =
17    2    =     END RQISND;
                $include(:f0:exch.elt)
18    1    =   DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =    =    MSG$HEAD ADDRESS,
      =    =    MSG$TAIL ADDRESS,
      =    =    TASK$HEAD ADDRESS,
      =    =    TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;

23   1      cq$msg$out:
             procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
24   2          declare msg$size byte;
25   2          declare (q$ptr,par$ptr,msg$ptr) address;
26   2      end cq$msg$out;

27   1      CQ$GEN$RDY: Procedure (trget,msg$ptr,q$ptr,par$ptr) reentr
        -     ant public;

             /*
                 This procedure generates a "ready" message onto the
                 communication network. The target address is passed
                 as the parameter in the call.

                 On entry to the procedure:
                     trget is a byte which contains the address
                         of the target node.
                     msg$ptr is an address which points to
                         the RAM work area.
                     q$ptr is an address which points to
                         the output queue.
                     par$ptr is an address which points to
                         the communication flags.
             */

28   2          Declare trget byte;
29   2          Declare rdy$msg structure (
                     TARGET     byte,
                     COMAND     byte,
                     SEQ$LA     byte,
                     SEQ$NA     byte  )
                 data (  0,3,0,0 );

30   2          declare (msg$ptr,q$ptr,par$ptr) address;

31   2          declare msg based msg$ptr msg$format;

                 /* move format into message block */
```

```
32   2          call move (4,.rdy$msg,msg$ptr);
                /* insert current parameters */
33   2          msg.trgt = trget;
                /* send message */
34   2          call cq$msg$out (0, c$ptr, par$ptr, msg$ptr);
35   2          return;
36   2      end cq$gen$rdy;
37   1      end gen$rdy$module;
```

```
                $title('DATA MESSAGE GENERATOR PROCEDURE')
 1              GEN$MSG$MODULE: Do;

 2    1             declare eom literally '0DH';
                $include(:f1:commsg.elt)
 3    1    =        declare msg$format literally 'structure (
      =                  trgt       byte,
      =                  cmnd       byte,
      =                  seq$la     byte,
      =                  seq$na     byte,
      =                  text(250)  byte )';
      =
 4    1    =        declare queue$format literally 'structure (
      =                  end$ptr          byte,
      =                  give$ptr         byte,
      =                  take$ptr         byte,
      =                  data$byte(254)   byte )';
      =
 5    1    =        declare par$format literally 'structure (
      =                  la         byte,
      =                  na         byte,
      =                  run        byte,
      =                  stop       byte,
      =                  que$pt     byte )';
                $include(:f0:synch.ext)
 6    1    =    RQSEND:
      =           PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
 7    2    =             DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
 8    2    =           END RQSEND;
      =
 9    1    =    RQWAIT:
      =           PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10    2    =             DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
11    2    =           END RQWAIT;
      =
12    1    =    RQACPT:
      =           PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13    2    =             DECLARE EXCHANGE$POINTER ADDRESS;
      =
14    2    =           END RQACPT;
      =
15    1    =    RQISND:
      =           PROCEDURE (IED$PTR) EXTERNAL;
16    2    =             DECLARE IED$PTR ADDRESS;
      =
17    2    =           END RQISND;
                $include(:f0:exch.elt)
18    1    =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =           MSG$HEAD ADDRESS,
      =           MSG$TAIL ADDRESS,
      =           TASK$HEAD ADDRESS,
      =           TASK$TAIL ADDRESS,
```

```
        =    NEXT$EXCH ADDRESS)';
             $include(:f0:msg.elt)
19   1  =    DECLARE MSG$HDR LITERALLY '
        =     LINK ADDRESS,
        =     LENGTH ADDRESS,
        =     TYPE BYTE,
        =     HOME$EXCHANGE ADDRESS,
        =     RESPONSE$EXCHANGE ADDRESS';
        =
20   1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =     MSG$HDR,
        =     REMAINDER(1) BYTE)';

21   1          declare timex exchange$descriptor external;
22   1          declare rqfsrx exchange$descriptor external;

23   1        cq$msg$out:
               procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
24   2          declare msg$size byte;
25   2          declare (q$ptr,par$ptr,msg$ptr) address;
26   2        end cq$msg$out;

27   1        CQ$GEN$MSG: Procedure (msg$pointer, trget, msg$ptr, q$ptr,
         -      par$ptr,save$flg) reentrant public;

                /*
                    This procedure generates a data message and places
                    it onto the system communications network.

                    On entry to the procedure:
                        msg$pointer is an address which points to
                            the data to be transmitted.
                        trget is an byte which contains the
                            address of the target node.
                        msg$ptr is an address which points to
                            the RAM work area.
                        q$ptr is an address which points to
                            the output queue.
                        par$ptr is an address which points to
                            the communications flags.
                */
28   2          Declare (msg$pointer,msg$ptr,q$ptr,par$ptr,ram$size) a
         -    ddress;
29   2          Declare (trget,save$flg) byte;

30   2          Declare msg based msg$ptr msg$format;

31   2          Declare data$msg structure (
                    target      byte,
                    comand      byte,
                    seq$LA      byte,
                    seq$NA      byte,
                    text        byte )
                    data (0,2,0,0,0);
```

```
32   2          declare data$block based msg$pointer structure (
                    msg$hdr );
                /* move message format into message buffer */

33   2          call move (5, .data$msg, msg$ptr);

                /* insert target address */

34   2          msg.trgt = trget;

                /* append data to communication message */

35   2          call move (data$block.length, msg$pointer, .msg.text(0
         -    ) );

                /* transmit message onto network */

36   2          ram$size = data$block.length;
37   2          call cq$msg$out (ram$size,q$ptr,par$ptr,msg$ptr);

                /* return memory to free space manager */

38   2          if save$flg
                then do;
40   3              if ramsize mod 4 > 0
                    then data$block.length = data$block.length + (4-(r
         -    am$size mod 4));
42   3              call RQSEND (.rqfsrx, msg$pointer);
43   3          end;

44   2          return;
45   2      end cq$gen$msg;
46   1      end gen$msg$module;
```

```
                $title('iSBX 351 Communications Driver, Version 1.?')
                $nointvector
1               CQS351: Do;

                /*************************************************************

                    This module contains the physical interface for
                    support of the Intel iSBX 351 communications expansion
                    board.  The board is inserted into socket J6 and has a
                    base address of FCH with the interrupt from the
                    receiver strapped to the interrupt level 6. The
                    transmitter interrupt is wired to interrupt level 5.

                        Multi-drop communication options are supported by
                        the physical software package.

                *************************************************************/


                $include (:f0:exch.elt)
2     1   =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =      MSG$HEAD ADDRESS,
          =      MSG$TAIL ADDRESS,
          =      TASK$HEAD ADDRESS,
          =      TASK$TAIL ADDRESS,
          =      NEXT$EXCH ADDRESS)';
                $include (:f0:ied.elt)
3     1   =     DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =      MESSAGE$HEAD ADDRESS,
          =      MESSAGE$TAIL ADDRESS,
          =      TASK$HEAD ADDRESS,
          =      TASK$TAIL ADDRESS,
          =      EXCHANGE$LINK ADDRESS,
          =      LINK ADDRESS,
          =      LENGTH ADDRESS,
          =      TYPE BYTE)';
4     1         Declare RQL6EX int$exchange$descriptor external;
5     1         Declare RQL7EX int$exchange$descriptor external;

                $include (:f1:commsg.elt)
6     1   =         declare msg$format literally 'structure (
          =             trgt      byte,
          =             cmnd      byte,
          =             seq$la    byte,
          =             seq$na    byte,
          =             text(250) byte )';
          =
7     1   =         declare queue$format literally 'structure (
          =             end$ptr         byte,
          =             give$ptr        byte,
          =             take$ptr        byte,
          =             data$byte(254) byte )';
          =
8     1   =         declare par$format literally 'structure (
```

```
        =               ja          byte,
        =               na          byte,
        =               run         byte,
        =               stop        byte,
        =               que$pt      byte )';
 9  1       Declare CQINSL queue$format external;
10  1       Declare CQOUTSL queue$format external;
11  1       Declare CQPARS par$format external;

            $include (:f1:comcom.ext)
12  1   =   CQ$NEXT$GIVE:
        =       Procedure (q$ptr,give$byte) byte external;
13  2   =       Declare q$ptr address;
14  2   =       Declare give$byte byte;
15  2   =       end cq$next$give;
16  1   =   CQ$TAKE$TEST:
        =       Procedure (q$ptr) byte external;
17  2   =       Declare q$ptr address;
18  2   =       end cq$take$test;
19  1   =   CQ$NEXT$TAKE:
        =       Procedure (q$ptr) byte external;
20  2   =       Declare q$ptr address;
21  2   =       end cq$next$take;
22  1   =   CQ$Q$INIT:
        =       Procedure (q$ptr,q$size) external;
23  2   =       Declare q$ptr address;
24  2   =       Declare q$size byte;
25  2   =       end cq$q$init;
26  1   =   CQ$ASC$HEX:
        =       Procedure (q$ptr,msg$ptr) external;
27  2   =       Declare (q$ptr,msg$ptr) address;
28  2   =       end cq$asc$hex;
29  1   =   CQ$CHECKSUM:
        =       Procedure (q$ptr) byte external;
30  2   =       Declare q$ptr address;
31  2   =       end cq$checksum;
32  1   =   CQ$HEX$ASC:
        =       Procedure (q$ptr,hex$byte) external;
33  2   =       Declare q$ptr address;
34  2   =       Declare hex$byte byte;
35  2   =       end cq$hex$asc;
36  1   =   CQ$MSG$OUT:
        =       Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
37  2   =       Declare msg$size byte;
38  2   =       Declare (q$ptr,par$ptr,msg$ptr) address;
39  2   =       end cq$msg$out;
40  1   =   CQ$GEN$RDY:
        =       Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
41  2   =       Declare trget byte;
42  2   =       Declare (msg$ptr,q$ptr,par$ptr) address;
43  2   =       end cq$gen$rdy;
44  1   =   CQ$GEN$MSG:
        =       Procedure (msg$pointer,trget,msg$ptr,q$ptr,par$ptr,sav
         -    e$flg) external;
45  2   =       Declare (trget,save$flg) byte;
```

```
46   2  =        Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
47   2  =        end cq$gen$msg;
                $include (:f0:synch.ext)
48   1  =    RQSEND:
        =      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
49   2  =        DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
        =
50   2  =      END RQSEND;
        =
51   1  =    RQWAIT:
        =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
52   2  =        DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
        =
53   2  =      END RQWAIT;
        =
54   1  =    RQACPT:
        =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
55   2  =        DECLARE EXCHANGE$POINTER ADDRESS;
        =
56   2  =      END RQACPT;
        =
57   1  =    RQISND:
        =      PROCEDURE (IED$PTR) EXTERNAL;
58   2  =        DECLARE IED$PTR ADDRESS;
        =
59   2  =      END RQISND;
                $include (:f0:intrpt.ext)
60   1  =    RQENDI:
        =      PROCEDURE EXTERNAL;
        =
61   2  =      END RQENDI;
        =
62   1  =    RQELVL:
        =      PROCEDURE (LEVEL) EXTERNAL;
63   2  =        DECLARE LEVEL BYTE;
        =
64   2  =      END RQELVL;
        =
65   1  =    RQDLVL:
        =      PROCEDURE (LEVEL) EXTERNAL;
66   2  =        DECLARE LEVEL BYTE;
        =
67   2  =      END RQDLVL;
        =
68   1  =    RQSETV:
        =      PROCEDURE (PROC,LEVEL) EXTERNAL;
69   2  =        DECLARE PROC ADDRESS;
70   2  =        DECLARE LEVEL BYTE;
        =
71   2  =      END RQSETV;
        =
72   1  =    RQSETP:
        =      PROCEDURE (PROC,LEVEL) EXTERNAL;
73   2  =        DECLARE PROC ADDRESS;
74   2  =        DECLARE LEVEL BYTE;
```

```
      =
75    2 =     END RQSETP;

76    1       Declare EOM literally '0DH';
                /* defines end of message    */
77    1       Declare RTS literally '00100000b';
                /* ready to send to USART    */
78    1       Declare RXE literally '00000100b';
                /* ready to receive to USART */
79    1       Declare DTR literally '00000010b';
                /* data set ready to USART   */
80    1       Declare TXEN literally '00000001b';
                /* transmit enable to USART  */
81    1       Declare badint byte;
                /* flag for interrupt 5   */

                $eject
                /*************************************************************

                    This procedure initializes the hardware required to
                    operate the iSPX 351 expansion board.

                *************************************************************/

82    1       CQ$INIT$351:
                    Procedure public;
83    2         Declare  n  byte;

                    /* initialize the timer/counters */
84    2         output(0fbh) = 0b6h;          /* select counter 2 */
85    2         output(0fah) = 32;            /* lsb for 2400 baud */
86    2         output(0fah) = 0;             /* msb for 2400 baud */

                    /* initialize the USART */
87    2         output(0f1h) = 80H;            /* clear out garbage */
88    2         output(0f1h) = 0;             /* ... more garbage  */
89    2         output(0f1h) = 40h;           /* reset the USART   */
90    2         output(0f1h) = 0ceh;          /* 8 bit, no parity, 2 st
          -    op */
91    2         output(0f1h) = 26h;           /* enable receive mode */

                        /* tell the nucleus the vector location*/

92    2         CALL rqsetv(.cqmivt$351, 6);
93    2         call rqsetv(.send$char$351, 5);
                      call rqelvl( 5);
95    2         call rqelvl(6);
96    2         return;
97    2       end cq$init$351;
                $eject
                /*************************************************************

                    This procedure stores a character from the USART into
                    the receiver data input queue.
```

```
                ************************************************************/

 98    1        CQMIVT$351:
                    Procedure interrupt 5 public;
 99    2          Declare (GIVE$STATUS,CHAR) byte;

                    /* get character from USART */
100    2          char = input(0F0h) and 07Fh;
101    2          give$status = cq$next$give(.cqinsl, char);
102    2          if char = eom
                   then call rqisnd(.rql6ex);
104    2          else call rqendi;

105    2        end cqmivt$351;
                $eject
                /************************************************************
                     This procedure sends out the next character to the
                     selected USART device. It will disable the interrupt
                     when all desired characters have been transmitted.

                ************************************************************/
106    1        SEND$CHAR$351:
                    Procedure interrupt 5 public;
107    2          Declare char byte;

                    /*testing if interrupt is really for 5 and not noise*/

108    2          if badint > 0 then do;

                    /* enable drivers at beginning of message */
110    3          If not cqpars.run
                   then do;
112    4               cqpars.run = 1;
113    4               cqpars.stop = 0;
114    4          end;

                    /* disable drivers at end of message */
115    3          If cqpars.stop
                   then do;
117    4               cqpars.run = 0;
118    4               Do while (input(0F1h) and 4) = 0;
119    5               end;
120    4               badint = 0;
121    4               output(0F1h) = 26h;
122    4          end;

                    /* if message is in progress, send next character */
123    3          else do;
124    4               char = cq$next$take(.cqoutsl);
125    4               do while (input(0f1h) and 4) = 0;
126    5               end;
127    4               output(0F0h) = char;

                        /* test for end of message */
128    4               if (char and 0fh) = eom
```

```
                        then cqpars.stop = 1;
130    4                else cqpars.stop = 0;
131    4            end;
132    3            end;

                    /* re-enable interrupts */
133    2            call rqendi;

134    2            return;

135    2        end send$char$351;
                $eject
                /*************************************************************

                This procedure begins the transmission to a selected node.

                *************************************************************/
136    1        CQ$START$MSG$351:
                    Procedure public;

137    2            badint = 0ffh;
138    2            output(0F1h) = 25h;
139    2            return;
140    2        end cq$start$msg$351;

141    1        end cqs351;
```

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

Fold Line ----------

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OREGON

POSTAGE WILL BE PAID BY ADDRESSEE

INTEL CORPORATION
3585 S.W. 198th
Aloha, Oregon 97005

Attention: OMS Applications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA • (408) 734-8102 x598

Printed in U.S.A./B-354/0181/10K/BA

AFN-01805A