A Fault Tolerant Transportation Controller

by

Scott D. MacGregor

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Science
at the Massachusetts Institute of Technology

May 28, 1997

Copyright 1997 Scott D. MacGregor. All rights reserved

The author hereby grants to M.I.T. permission to reproduce
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

OCT 29 1997

Author ....
Department of Electrical Engineering and Computer Science
May 28, 1997

Certified by ....
Richard D. Thornton
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A
Fault Tolerant
Transportation Controller

by
Scott D. MacGregor

Submitted to the
Department of Electrical Engineering and Computer Science

May 28, 1997

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A distributed fault tolerant controller is designed and implemented which could be used for controlling systems such as personal rapid transit (PRT) or baggage delivery systems. The emphasis is placed on designing a communication backbone capable of quickly detecting and reacting to faults among the distributed components. A simple set of control protocols are introduced to verify and test the functionality of the communication backbone. A small scale prototype has been implemented to develop the ideas introduced in the communication backbone and controller design. The prototype system has been shown to handle multiple failures of communication lines and processors.

Thesis Supervisor: Richard D. Thornton
Title: Professor, Electrical Engineering and Computer Science

To my grandfather
Richard Fisher

and

In memory of
David A. MacGregor

Table of Contents

1. INTRODUCTION 8

1.1 THE TRANSPORTATION CONTROLLER 8
1.2 THE PROBLEM 9
1.3 FAULT TOLERANCE 10
1.4 MAKING IT FAULT TOLERANT 10
1.4.1 Zone Controller 11
1.4.2 The Watchdog Circuit 12
1.4.3 Neighbors 13
1.4.4 Communication Backbone 13
1.4.5 Robust Protocols 13
1.5 ORGANIZATION 13

2. DESIGN 15

2.1 NETWORK TOPOLOGY 15
2.1.1 Multidrop Communication Network 15
2.1.2 Point to Point Networks 17
2.2 COMMUNICATIONS BACKBONE 17
2.2.2 Messages 19
2.2.3 The Router 19
2.2.4 Message Handlers 22
2.2.5 Port Handlers 23
2.2.6 Ping Protocol 26
2.3 DEVELOPMENT TOOLS 28
2.3.1 Event Handler 28
2.3.2 Vehicle Manager 29
2.4 PROTOCOLS 30
2.4.1 Virtual Vehicle Protocol 31
2.4.2 Vehicle Exchange 32

3. IMPLEMENTATION 35

3.1 HARDWARE 35
3.1.1 Main Processor 36
3.1.2 Auxiliary Processor 38
3.1.3 DUART 38
3.1.4 LCD Display 39
3.1.5 DIP Switches 39
3.2 SOFTWARE IMPLEMENTATION OF THE COMMUNICATION BACKBONE 40
3.2.1 The Router 40
3.2.2 Message Handler 41
3.2.3 SCI Port Handler 44
3.2.4 DUART Port Handlers 46
3.2.5 Ping Protocol 49
3.3 DEVELOPMENT TOOLS 51
3.3.1 Event Handler 51
3.3.2 Vehicle Manager 53
3.4 CONTROL PROTOCOLS 56
3.4.1 Virtual Vehicle Create 56
3.4.2 Vehicle Exchange 57
3.5 MAIN LOOP 59

4. FURTHER STUDY 60

4.1 Expanding the Controller Algorithms 60
4.2 Zone Controller Voting 60
4.3 Adding a Central Controller 62
4.4 Statistics 62
4.5 Dynamically Download New Code 62
4.6 Adding the Watchdog Circuit 63
5. CONCLUSION 64
6. REFERENCES 66
7. **ACKNOWLEDGMENTS 67**
8. APPENDIX A: PROTOTYPE SCHEMATICS 68
9. APPENDIX B: CODE LISTING 76

List of Figures

FIGURE 1-1. A BASIC DISTRIBUTED TRANSPORTATION CONTROLLER 9
FIGURE 1-2. COMMUNICATION FAILURE BETWEEN BLOCKS 12
FIGURE 2-1. MULTIDROP COMMUNICATION NETWORK TOPOLOGY 16
FIGURE 2-2. POINT TO POINT NETWORK TOPOLOGY 17
FIGURE 2-3. OVERVIEW OF COMMUNICATION BACKBONE SYSTEM 18
FIGURE 2-4. MESSAGE FORMAT 19
FIGURE 2-5. BLOCK CONTROLLER ROUTING TABLE 21
FIGURE 2-6. UPDATED ROUTING TABLE 21
FIGURE 2-7. TWO LAYER PORT HANDLER 24
FIGURE 2-8. THE PING PROTOCOL 27
FIGURE 2-9. SNAPSHOT OF AN EVENT LIST 29
FIGURE 2-10. UPDATE EVENT LIST 29
FIGURE 2-11. VIRTUAL VEHICLE CREATION PROTOCOL 32
FIGURE 3-1. CONTROLLER BLOCK DIAGRAM 36
FIGURE 3-2. DEFAULT ROUTING TABLE 41
FIGURE 3-3. VIRTUAL VEHICLE CREATE MESSAGE 56
FIGURE 4-1. VOTING ON THE ZONE CONTROLLER. 61
FIGURE 4-2. SYSTEM HIERARCHY WITH A CENTRAL CONTROLLER 62

List of Tables

TABLE 3-1. DIP SWITCH SIGNALS 40
TABLE 3-2. VALID PING SOURCE AND DESTINATION PAIRINGS 49
TABLE 3-3. SUMMARY OF POLLED VARIABLES FOR LOOP DRIVER 59
TABLE 3-4. SUMMARY OF INTERRUPT SOURCES 59

1. Introduction

A lot of recent research and funds have been funneled into designing automated transportation systems. As this push continues, more emphasis will be placed on designing fault tolerant, safe transportation control systems. Current transportation control systems are still very primitive. For example, conventional trains today rely on mechanical switches known as track circuits to determine if a stretch of track has been cleared ahead. These systems could have a better throughput if train engineers had the ability to precisely determine the distance to the train in front of them. [2]. A good transportation controller could also be useful in efforts to build autonomously controlled cars. The controller system can make sure a car does not enter the next segment of the road unless the road is clear. This thesis is aimed at developing a communication backbone which would support the construction of complex distributed controllers. The backbone and the controller must be fault tolerant and flexible enough to be useful in various applications such as subway systems, baggage delivery or autonomously controlled cars.

1.1 The Transportation Controller

A common technique for real-time control systems involving continuous control and observation uses distributed systems where individual processing units are responsible for monitoring / controlling a part of the system [6]. This technique can be applied to the transportation controller. Consider a distributed system with multiple processors where each processor is associated with a segment of track. The processor controls any vehicle on its track segment and is responsible for communicating with the processor in charge of adjacent segments of track to coordinate the vehicle's move from one segment to the next. The communication between the two processors ensures that a vehicle does not prematurely enter the next track segment. For instance, Figure 1-1 shows a basic distributed controller. If processor A wants to move the vehicle in its track into the track segment of processor B, A must communicate with B to determine if B's track segment is clear. In this case, B would inform A to slow its vehicle down because B's track is not clear. Thus, the system depends on bi-directional communication between processors which in turn control vehicles on the tracks.

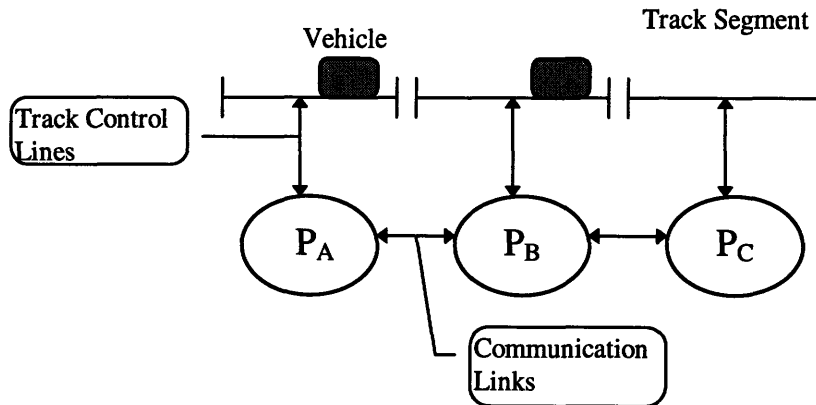


Figure 1-1. A Basic Distributed Transportation Controller

Each processor receives information from the track. The processor also has the ability to control vehicles on its segment of the track. The communication link between neighboring processors allows them to coordinate the vehicles.

It is important to note that the idea of a "vehicle" and a "track" can be abstracted away from this general transportation system. The vehicles could be as diverse as automated cars, subway trains or even bags of luggage. In the same sense, the "tracks" could be highways, subway tracks or luggage conveyers. Thus, the details associated with objects which are moved and the medium used to move them can be removed from the discussion and analysis of the system for the purposes of this thesis.

1.2 The Problem

A system which meets just the above criteria is very easy to design. However, what happens when one or more of the processors fails or becomes inoperative? Suddenly, a processor cannot ask its neighbor if the next track segment is open or not. For the case of a luggage system, the consequence might be no more than an inconvenience. However, in the case of a subway system the results could be disastrous. Imagine the loss of life in a severe subway crash during rush hour! Clearly, changes must be made to the transportation model we have introduced.

This thesis will attempt to introduce a high level of fault tolerance into the system. The system will be able to:

- Quickly detect processor failures.
- Quickly detect communication failures.

- When dealing with a few processor failures, gracefully keep the system in operation while notifying a central unit.
- Gracefully handle disruptions in the communication lines between the processors by re-routing messages when possible.
- In the face of many processor failures, safely but not necessarily gracefully, handle the situation.
- Minimize the number of false alarms.
- Avoid collisions.
- Maintain service.

For instance, if a single processor fails, the system should have the ability to move vehicles out of the bad processor's track and continue service. If many processors suddenly fail (a large power failure or accident) then the system could slow all vehicles to a halt. This event would be an example of a non-graceful but safe handling of a serious failure.

1.3 Fault Tolerance

Some of the different applications for transportation controllers have high availability needs (e.g. a subway system). High availability systems in general have the properties that they can quickly detect faults, report them, mask them and continue service while the fault is fixed [3]. As we can see, these are goals of our system as well. In order to meet these goals, the system must be fault tolerant.

Fault tolerant design typically incorporates the following techniques [3]:

- Modularity - The system is composed of modules.
- Independent failure modes - faults in one module should not cause faults in other modules.
- Redundancy and repair - make it easy to replace modules in the system.

The fault tolerant transportation controller design discussed in this thesis demonstrates these fundamental principles of fault tolerant design.

1.4 Making it Fault Tolerant

Taking the basic distributed model introduced in Section 1.1, the system must be modified to introduce multiple ways of detecting failures and dealing with them in order to become fault tolerant. The four techniques introduced in this thesis are:

We are going to need to introduce multiple ways of detecting failures in order to make the system fault tolerant. The primary ways are:

- Zone Controller
- Using neighbors to detect failures.
- Fault Tolerant Communication Backbone
- Watchdog
- Robust Protocols

1.4.1 Zone Controller

Consider an additional layer of processing units which would sit on top of the basic processing units introduced in section 1.1. These new processing units are considered zone controllers and they manage the basic processing units which act as block controllers. Every zone controller can communicate with the blocks in its zone. Furthermore, block controllers keep their zone controller up to date with the status of their track segment. For example, a block controller notifies the zone controller whenever vehicles enter or leave its track segment. The block controllers have a very detailed localized view of the vehicles and the track. With the information the zone controller receives from its blocks, it has a general, global view of the system. Adding zone controllers to the system can improve the fault tolerance of the system by providing redundancy and independent failure modes. In addition zone controllers can be used for actually controlling the system.

The zone controller provides redundancy in the communication links. If the communication link between two blocks in a zone goes down, messages between the two blocks can be re-routed through the zone controller since each block has a separate means of communicating with the zone as illustrated in Figure 1-2.

The zone controller also provides an independent failure mode by maintaining a global view of the system. Without the zone controller, if a block processor faults, then a neighbor block could not determine if the track ahead was clear or not. Since the neighbor has no knowledge about the status of the crashed block's state, it must stop moving vehicles from its block to the crashed block. This in turn backlogs vehicles coming into the neighbor processor. Thus, the failure of one block controller eventually succeeds in bringing a large portion of the transportation system down. However, with a zone controller, when one block fails, the neighbor asks the zone controller about an appropriate behavior. The zone controller, using its global state has knowledge with regards to

the track of the failed block being cleared or not to instruct the block controller. If the track is actually clear then the zone can disable the bad controller, and the vehicle can be accelerated and allowed to coast through the bad block. Now, the failure of a block controller does not cause the entire system to fail by losing service. It is important to note that the zone controller must have the ability to diagnose, reboot, reload and disable block controllers.

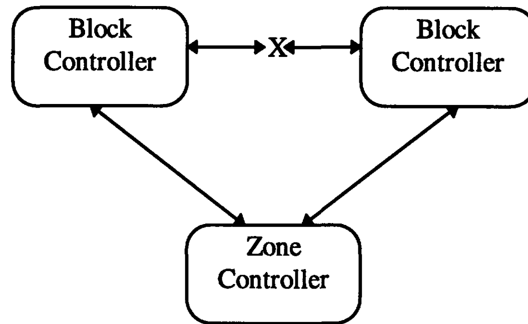


Figure 1-2. Communication failure between blocks

1.4.2 The Watchdog Circuit

Watchdog circuits can be used to help quickly detect block controller processors which have faulted. A traditional watchdog circuit, as the name suggests, watches a processor, making sure it is alive. In fact, sophisticated watchdogs can actually monitor correct processor behavior.

Traditional watchdogs usually work by monitoring a signal from the processor. If the processor does not "hit" the signal every fixed number of time units then the watchdog assumes the processor has faulted. On the processor side, a common mistake is to hit the watchdog via an interrupt. An interrupt goes off every fixed number of time units causing the processor to stop whatever it is doing and "hit" the watchdog signal. Unfortunately, this simple technique alone is not a good way to monitor the status of the processor. The interrupt process might be functioning while the processor program has faulted (i.e. caught in an infinite loop). Thus, the watchdog in this form might think the processor is alive and well even when it has faulted. However in severe cases where the processor has completely halted all operations, it is a reliable means to test the status of the processor.

So the watchdog circuit needs to be more powerful than this. Another way to check proper functioning of the processor is to have a more complicated exchange between the watchdog and the block controller. For example, the watchdog circuit might exchange information with the processor. The processor is then required to reply in some form to this message. If the watchdog does not receive a proper response from the processor, then it concludes that the processor has faulted and notifies the zone controller. Again, the purpose of the watchdog is to help quickly detect block controller failures.

1.4.3 Neighbors

Effective use of neighboring block controllers is critical towards detecting faults. If a block controller sends a message to one of its neighbors and does not receive a reply, it can send a higher priority message (just in case the other processor just hadn't had time to respond to the first message). If that fails, then the neighbor concludes that the processor has failed.

1.4.4 Communication Backbone

A communication backbone must be established between the block and zone controllers not only to send and receive messages but also to quickly detect at faults in the communication lines. A fault tolerant communication backbone has the capability of continuing service by automatically re-routing messages over different communication links to the original destination. When the communication lines are later repaired, the communication backbone must automatically detect the connectivity of the link and route messages back though the link.

1.4.5 Robust Protocols

In addition to providing fault tolerant communication, a transportation controller must also provide robust control protocols. These protocols are used to perform high level communication and coordination of vehicle exchanges between block controllers. The complexity of these protocols depends on the complexity of the desired control algorithms. These protocols must be fault tolerant with regards to messages not being delivered. Since the communication backbone does not guarantee message delivery, protocols must be written using acknowledgments and other techniques to handle messages that are dropped by the communication backbone.

1.5 Organization

Chapter 2 introduces the design of a fault tolerant transportation controller. The design incorporates issues such as the network topology which supports communication between block and zone controllers, the fault tolerant communication backbone and a simple robust set of

protocols for controlling vehicles. Chapter 3 focuses on an actual prototype that was developed using this design. Chapter 4 discusses areas of future work followed by concluding remarks in Chapter 5.

2. Design

The design is broken down into four main components. The first involves defining the network topology to be used. Second is the construction of the communication backbone which lies on top of the network topology. The communication backbone is a set of constructs used by block and zone controllers to provide fault tolerant communication via message passing. The third component is a set of support tools for high level protocol design. Finally, several high level protocols are developed for the system. These protocols actually provide the actual controller for the system whereas the first three components are geared towards producing a fault tolerant communication system. These protocols lie on top of the communication backbone and protocol support tools. This chapter is broken down into discussing these four modules. Combined, they form a fault tolerant transportation controller.

2.1 Network Topology

A well chosen topology for the network is critical for designing a fault tolerant communication system. In our system, the topology specifies the number and type of communication links between block and zone controllers. A good topology must be able to handle multiple communication link failures while still keeping blocks and zones in communication with each other. Furthermore, a good topology should be able to scale as the number of elements (blocks and zones) in the system increases. There are at least two possible topologies which could be used in the network controller: a multidrop communication network and a point to point network. Multidrop communication networks such as Ethernet function on the principle of a single broadcast medium which all network elements broadcast on. Point to point networks rely on point to point connections between the elements. We will now examine the advantages and disadvantages of each one.

2.1.1 Multidrop Communication Network

Implementations of multidrop communication networks are found on many computer networks (e.g. Ethernet) used today. As illustrated in Figure 2-1, in this topology, all network elements share the same communications link. Ethernet controllers (ECs) are used with each element to interact with the medium. Data needing to be communicated is broadcast on the Ethernet. Every network element listens to the message on the medium and determines if they are the intended recipient. If it is not the intended recipient, the network element ignores the message. Since every element shares the Ethernet, only one network element can broadcast a message at a time [4]. On some multidrop

communication networks, network collisions occur when two or more elements broadcast at the same time. In the case of these collisions, both network elements stop, wait a random delay time and attempt to transmit again.

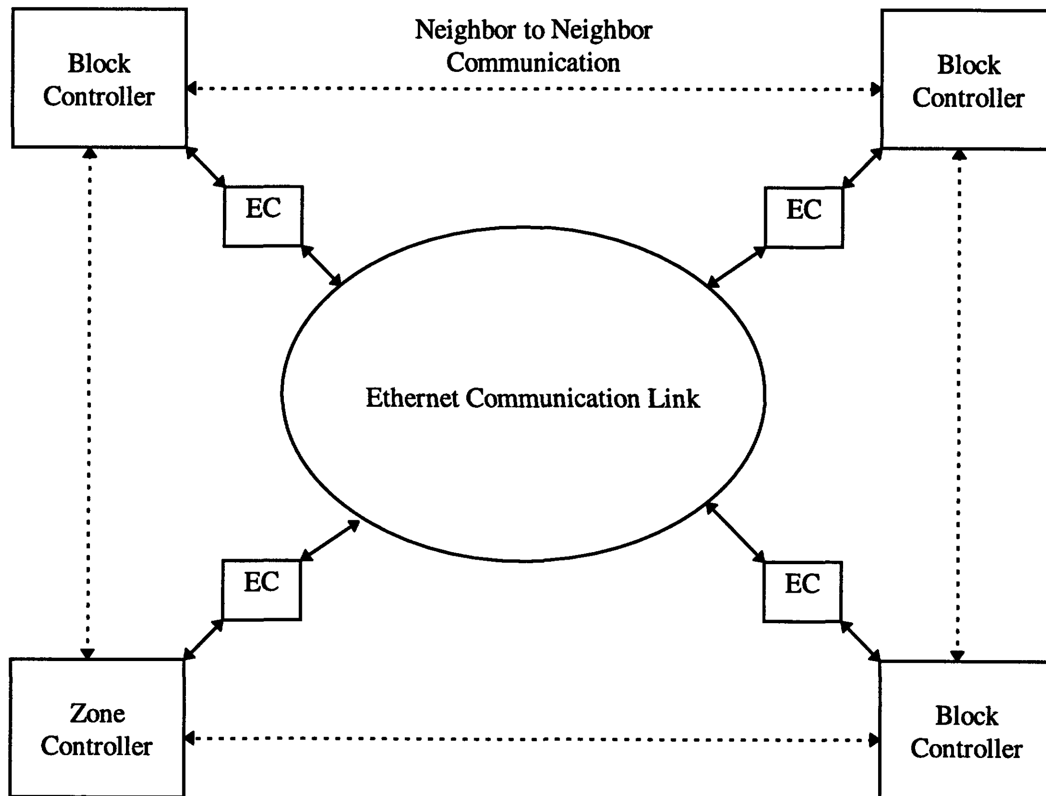


Figure 2-1. Multidrop communication network topology

This topology has a very nice property. It scales very well as the number of network elements grows since no structural changes need to be made. You simply add one more element to the Ethernet.

Unfortunately, the multidrop communication network has several serious drawbacks. The nature that allows it to scale so well also makes it very poor with regards to fault tolerance. Since every network element shares the same medium for communication, if the medium goes down then all communication with the zone controller ceases. Furthermore, having an Ethernet as part of the topology introduces a high hardware cost as each network unit must be equipped with an Ethernet controller. The effective bandwidth is constrained by the number of elements in the system. As the number of elements increases, so do the number of collisions.

2.1.2 Point to Point Networks

A point to point network provides a point to point communication link between the zone controller and each block controller. Figure 2-2 demonstrates a point to point network as it applies to our network. The solid lines represent the point to point communication lines. The dotted lines are the already existing block to block communication lines. The point to point network is very easy to get started as it does not require extra hardware like the multidrop communication network. Moreover, communication between a block and zone controller is isolated. Failure of a link does not effect the communication between other blocks and the zone controller.

Unfortunately, a point to point network does not scale very well. The number of communication links which must be added to the zone increases directly with the number of blocks added to the zone. Adding a communication link incurs a cost at the zone controller end as the zone controller must add more hardware to handle the new link.

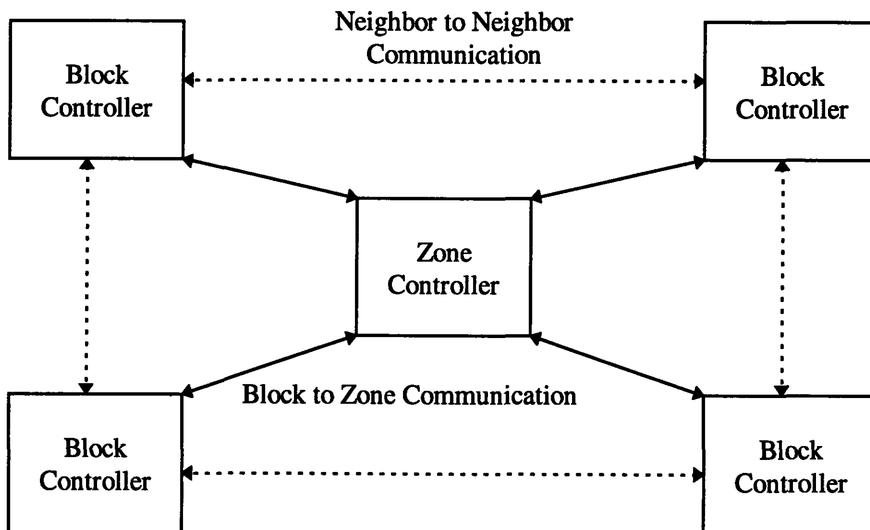


Figure 2-2. Point to Point network topology

The dependency of the multidrop communication network on a single medium for communication of all the network elements was deemed to high for a risk for a fault tolerant system. As such, the star network was chosen as the network topology for communication between blocks and zones. The subsequent sections on the communication backbone will assume this topology.

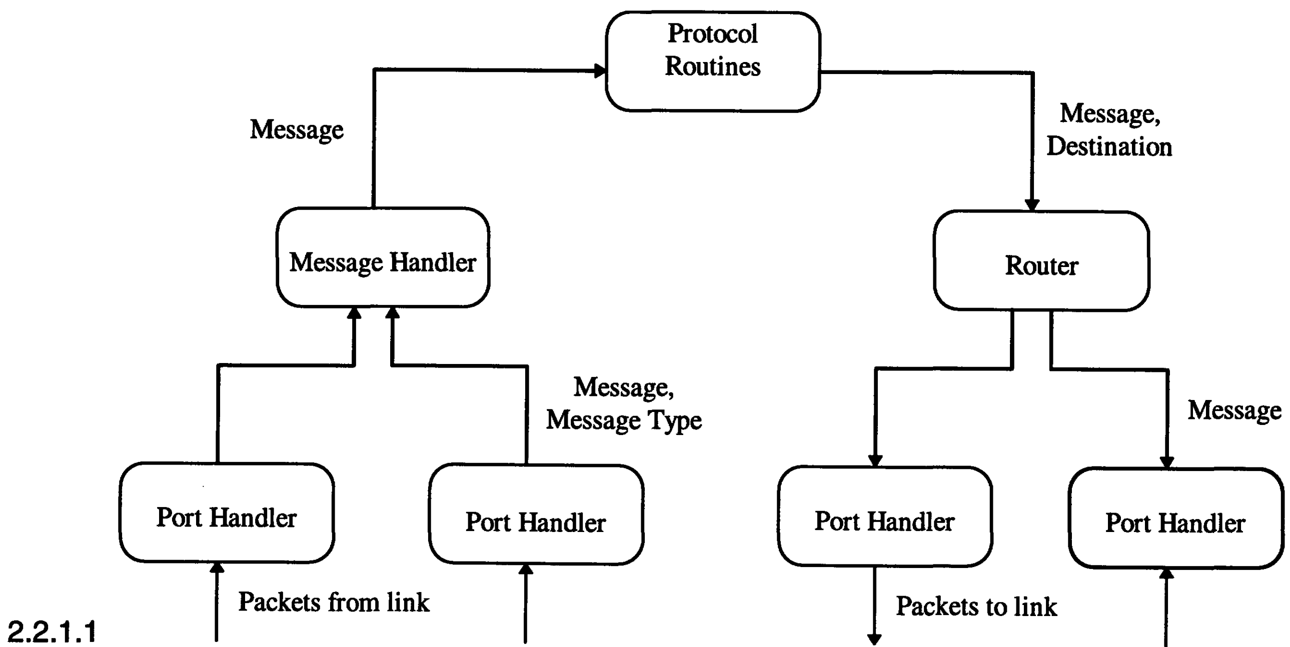
2.2 Communications Backbone

The communication backbone is responsible for sending messages, receiving messages, and passing received messages to appropriate protocol routines. It also has the responsibility of

providing fault tolerant communication between block and zone controllers. If a communication link between two elements in the network fails, it is the responsibility of the backbone to detect the failure and reroute future messages. The following subsections will describe how the backbone accomplishes each of these goals by constructing routers, message handlers, and port handlers in conjunction with the development of a ping protocol.

All communication within the system comes in the form of messages. Unformatted data is never exchanged between the different components. These messages are passed between the processors found in the block and zone controllers.

By forcing all data to be transmitted in the form of messages, we can abstract the specifics of message passing between the components from the actual protocols and action routines. The communication backbone can be decomposed into: the protocol level, the routing level, the message handler and the port handler level. Figure 2-3 illustrates the relationships between the different components.



2.2.1.1

Figure 2-3. Overview of Communication Backbone system

A port handler is in charge of a particular communication link. When a complete message is received from a particular port, the message is placed into the message handler. The message handler examines the type of message received and calls an appropriate protocol routine to deal with the message. On the other side of the system, when a protocol routine has generated a message, it passes the message and a destination (Zone Controller, Left Neighbor, Right Neighbor) to the router. The router decides which port handler should actually transmit the message based on the condition of the systems communication links. The port handler transmits the message over its associated communication link. In addition, the port handler adds error detection code to allow detection of errors which might occur during transmission of the message. The following sections discuss messages, the router, the message handler and port handlers in greater detail.

2.2.2 Messages

As a message passes through the different layers of the communication backbone it is modified and extended. At the very top level (protocol layer) a message is created by a protocol routine. A protocol fills the message, giving it a message code (*MessCode*), destination ID (*MessDest*), source ID (*MessSrc*), message size (*MessSize*), and any message specific information. Figure 2-4 summarizes this basic message format.

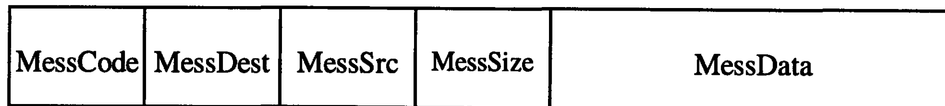


Figure 2-4. Message Format

As each subsequent layer of the communication backbone is analyzed, the layers added or removed to this basic message format will be discussed. But this portion of the message structure will be here after referred to as the *end-message* as it contains the message data for the end layers (the protocols).

2.2.3 The Router

Block and zone controllers must be able to correctly route new messages they have created or messages that may have been received and need to be forwarded to another destination The first case is the common case during normal operation. The latter case only occurs when one or more communication link failures occur, forcing communication between two components to be re-routed.

The router's primary duty is to map a message destination to a particular port. It takes in a message and a destination as inputs. The router then maps the destination to a particular port on the processor, handing the message off to the appropriate Port Handler (discussed later) for transmission. This section describes how the router directs messages to destinations through different ports. It introduces the concept of a routing table which is used to map message destinations to communication ports on the processor. Finally, updating the routing table is discussed.

2.2.3.1 Routing Table

Mapping destinations to ports must be a dynamic mechanism as message destinations and physical ports cannot be bound together since the mappings may change in the presence of failures. A routing table can be generated such that every possible message destination is a column entry and the source forms the row. Each entry in the table refers to a particular communication link available to the processor. Now, when a message needs to be routed, the destination ID and source ID are examined in the table. Their corresponding entry refers to the specific port on the processor which should be used to transmit the message.

This routing table solution can be simplified by taking into consideration an important facts. First, block controllers only have four communication links with the outside world. Namely, the

- Left Neighbor
- Right Neighbor
- Zone Controller
- Auxiliary Processor

Now the number of table columns is reduced to four entries.

With no communication link failures, a normal routing table for the main processor in a block controller might look something like Figure 2-5. The ports specified happen to correspond to those used in the prototype.

If a message is intended for the zone controller, the router looks up the zone controller ID in the table and determines to send it via the SCIPort. The router then passes the message to the SCIPort Handler which is responsible for actually transmitting the message.

	Zone Controller	Left Neighbor	Right Neighbor	Aux. Processor
Source	**SCI Port**	**UART 1 Port**	**UART 2 Port**	**SPI Port**

Figure 2-5. Block controller routing table

This routing scheme only works if the routing table is actually kept up to date. If failures force messages to be re-routed to different destinations, the routing table must reflect these changes. Consider the case where the physical link for the SCI Port has been damaged. Messages for the zone controller must be re-routed. The routing table must be updated to reflect this new information where messages intended for the zone controller are now forwarded to the left neighbor. It may now look like Figure 2-6.

	Zone Controller	Left Neighbor	Right Neighbor	Aux. Processor
Source	**UART 1 Port**	**UART 1 Port**	**UART 2 Port**	**SPI Port**

Figure 2-6. Updated Routing Table

Now, the block controller will route any messages intended for the zone controller through the UART 1 Port.

2.2.3.2 Updating Routing Information

The routing table for the block controller is designed to be small and simple. Since a block controller only has four communication links which are static, it is not necessary to provide the ability to add additional destinations to the routing table. However, it is essential that the entries in the routing table be kept current to reflect the current state of the communication network.

The routing table is updated after a communication link failure is detected. Detection of communication failures is handled by the ping protocol discussed in Section 2.2.6. However, once the failure is detected, the routing table must have the capability of updating entries. These update messages include the destination ID and a field which corresponds to the appropriate communication link to be used in forwarding messages to the destination. In addition, the router has the ability to restore entries when the a link becomes available again.

Updating the routing tables is always handled through the zone controller. The zone controller is responsible for sending update messages to block controllers under its zone. These update messages include the destination ID and a field which corresponds to the appropriate communication link to be used in forwarding messages to this destination. The zone controller has more information available to it with regards to the overall state of the communication network. More specifically, the status of all the blocks in the zone is available to the zone controller. Permitting the zone controller to handle all routing updates allows complex routing schemes to be installed to take advantage of this information.

One final note with regards to the routing table. It is possible in the case of severe communication failures that a block controller is presented with a message to forward whose destination ID is not in its routing table. The default behavior is to always forward the message on to the zone controller as the zone controller has the most information available to it for properly routing the message.

2.2.4 Message Handlers

The message handler has three main responsibilities: to place received messages from the different port handlers into priority message queues and to examine the messages at the head of the queues, routing them to appropriate protocol routines. In addition, the message handler forwards messages whose destination ID do not match that of the controller.

Since messages are arriving from multiple sources, and can be arriving faster than they can be processed, the message handler must have a way to store received messages until they can be processed. One solution is to require each port handler to have a large buffer which stores multiple messages. This way, the port handlers have the responsibility of storing messages and not the message handler. Unfortunately, this requires large buffer structures for each port handler since large bursts of messages can be received from each port handler.

A more memory efficient solution involves priority queues. The message handler has multiple queues ordered by priority. High priority messages received by the port handlers are placed in the high priority queue while low priority messages are placed in the lower priority queues. Messages are always placed at the tail of the queue. The priority of a message is determined by the type of message. Certain messages have a higher priority than others.

The other responsibility of the message handler involves routing received messages to protocol routines. Using the priority queues, this involves taking a message from the head of the highest, non-empty priority queue. The message type is extracted and used to determine the appropriate protocol routine to call by the message handler. The message is then given to the protocol routine.

2.2.4.1 Forwarding Messages

In the face of communication link failures, it is possible that a controller receives a message not intended for it. Forwarding messages is a critical component of the communication backbone as it helps insure connectivity between the modules when the original communication line between the modules has gone down. Before a message is passed on to the appropriate handler, the message handler examines the destination ID. If the destination ID does not match that of the controller, then the message must be forwarded to the actual destination. A call to the router with the message destination ID and the message contents actually forwards the message.

2.2.5 Port Handlers

Port handlers are used to separate the mechanics of the specific communication port being used (SPI, SCI, off chip UARTs, etc) from the rest of the software. This separation allows the message handler and router to be implemented without any dependency on the specifics of the actual ports being used for communication. This abstraction is particularly useful as only the port handlers need to be modified if the system uses different processors with different ports. As such, there is a port handler for each communication port on a processor. A port handler must support a common interface for the message handler and router to interact with. This common interface allows the router to transmit messages and the message handler to receive message. Since the port handler has responsibility for interacting with the communication lines, it is their responsibility to detect transmission errors.

Port handlers can be decomposed into two separate layers. The bottom layer deals solely with port specific information, providing packets to the top layer. A message packet is defined as the unit of data received or transmitted from the port at a time. For most ports, a packet is typically a byte.

The top layer, or management layer has three responsibilities. First, it handles message packets as they arrive, assembling them into complete message and notifying the message handler. In addition, the management layer takes a complete message from the router, breaks it down into packets and passes each packet down to the bottom layer. Finally, the management layer must add error

detection code to messages before they are transmitted. Figure 2-7 illustrates the relationship between the two layers in conjunction with the message queues which are discussed later. Receive and transmit buffers will be discussed in the following two sections followed by the error detection techniques employed by the port handlers.

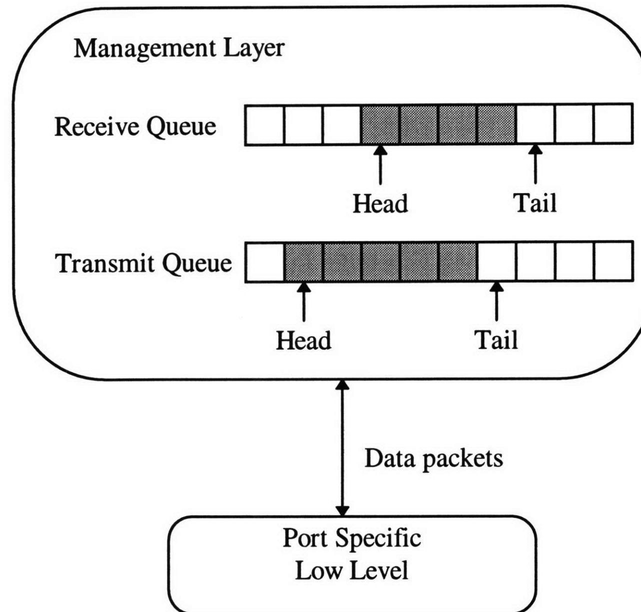


Figure 2-7. Two Layer Port Handler

2.2.5.1 Transmitting

When the router is ready to have a message transmitted to a destination, it instructs the particular port handler to transmit the message. The port handler is then responsible for storing the message, breaking it down into packets which can be transmitted over the particular port and inserting a header byte as discussed in Section2.2.5.3.

The port handler maintains a small queue for storing messages. Messages are added to the end of the queue and transmitted from the head. This transmit message queue (TMQ) allows the router to send multiple messages to the port handler without waiting for the first message to actually be transmitted. Without a TMQ, performance of the system would be degraded as the router could never finish its job until the particular port handler had finished transmitting the message. With a TMQ, the router can add messages to the end of the queue without waiting for messages to be transmitted. The port handler then transmits message packets while the system as a whole can

move forward. Figure 2-7 shows a possible TMQ state. The grayed portion indicates packets that still need to be transmitted. The white portion of the buffer signifies that it is empty.

During actual message transmission, the port handler inserts a header byte to the beginning of each message. The header byte is used by the receiver to determine the start of a message and is an important tool for detecting errors coming over the transmission link.

2.2.5.2 Receiving

In order to receive multiple messages without requiring the message handler to process the first message, the management layer maintains a receive message queue (RMQ). Where the handler queue stores messages, the RMQ stores message packets as they are received from the lower layer. Figure 2-7 shows a possible state of the RMQ. Grayed packet slots indicate packets that have been received, the white slots are empty. Eventually enough packets will be received to complete a message. At this time, the port handler informs the message queue that it has at least one completed message. The message handler then copies the message into the larger message handler queue.

In addition, the management layer responsible for receiving messages must also watch the packets coming in. Upon receipt of a header packet, signifying the start of a new message it must determine if the previous message being received was completed or not. If the message was not complete, then an error during the transmission must have occurred. The port handler then discards the remains of the partial message. Thus, the port handlers (which form the link/transport layer) do not guarantee delivery of messages. The higher level protocols (which are discussed later) must take responsibility for this.

2.2.5.3 Error Detection

By interacting with the physical communication links, the port handler accepts responsibility for detecting transmission errors. These transmission errors come in two forms: bit errors and incomplete messages. As such, the port handler employees two techniques to detect these errors: checksums and header bytes.

Incomplete messages can occur when the communication line goes down during reception of a message. At some later point, data is received again. However, this data may not be part of the first message that was partially destroyed. As such, the partial message should be discarded to prevent the high level protocol routines from dealing with them. Port handlers deal with this problem by

inserting header packets in front of each message before transmission. The header packet signifies the start of a new message and allows the receiving layer of the port handler to discard any partial data received before the header packet.

While a useful tool, header packets complicate the data transmission process. Since the header packet must have an identifiable value, message packets cannot share the same value or they will be confused with header packets. Port handlers solve this problem by breaking each message byte into two packets. For example, if the packet size is a byte then the high and low nibble of the byte would be transmitted separately. This leaves the high nibble empty for all the data bytes transmitted. The header byte is chosen such that it fills the high nibble of a data byte. Now the header packet can be distinguished from data packets. However, this does incur a cost in that the amount of data transmitted is doubled.

Unfortunately, using header packets to discard partially received messages is not good enough to ensure that the end layers will never see erroneous messages. Consider the case of a message corrupted during transit. That is a complete message which is received but during transit the data is altered. To handle this scenario, port handlers attach checksums on the messages before they are transmitted. When a message is received, the message handler computes a checksum for the message and compares the result with the checksum attached to the message. If they do not match, then the port handler discards the message. Otherwise, the message is treated as a complete and verified message.

There are various levels of complexity for implementing a checksum algorithm. The simplest is to add up the value of each byte in the message. The end result is a checksum. If any one bit in the message is changed, the checksum will not add up properly and an error is detected. However, in the case of multiple bit failures, it is possible that the checksum will fail to notice that the message is corrupted. More involved checksum algorithms could be implemented to handle multiple bit failures in a message.

2.2.6 Ping Protocol

In order to help insure proper message delivery, the underlying communication backbone on each controller must verify the correctness of the routing table. Thus, the controller must periodically examine the port connections. A ping protocol was developed for this purpose. The ping protocol is

actually a special low level protocol tied to the communication backbone. Unlike any of the control protocols (discussed later), the ping protocol actually has a direct access to the port handlers. The abstraction between destination (e.g. zone controller, left neighbor) and the message route (e.g. SCI Port, SPI Port) is broken. The ping protocol sends messages to particular routes and not destinations. This abstraction violation is justified by the fact that the goal is to test the connection of the route and not the connection with a particular destination.

As summarized in Figure 2-8, the ping protocol begins when a controller issues a ping request message for each port handler. With a valid connection, the processor at the other end of the link, receives the ping request, acknowledging the request with a ping response message. The initiator of the ping request then processes the response, noting that the port connection is active. After a determined amount of time, the protocol generates a time-out at which point all the ping responses are examined.

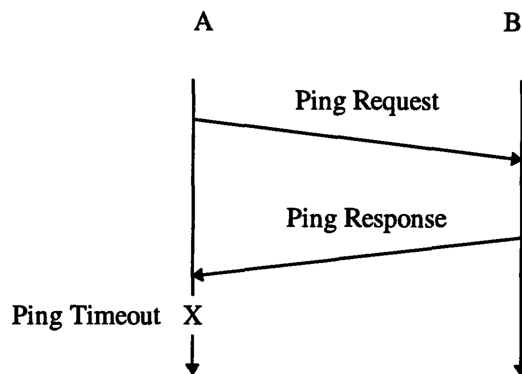


Figure 2-8. The Ping Protocol

2.2.6.1 Ping Time-out

After a pre-determined time interval starting from the time of the initial ping request, the ping requester must determine which routes have returned ping responses. At this time, any route which has not received a ping response is considered down. The routing table must be updated with a new route to be used for the destination that was previously associated with the route.

Once a communication line has been determined to be down, the ping protocol notifies the zone controller of the failure. The zone controller may then chose to update the block's routing tables with a forwarding locations.

2.3 Development Tools

To facilitate the construction of the control protocols, event and vehicle handlers are provided. The event handler serves as a timing abstraction barrier. Using an event handler, the control protocols do not have to worry about interrupts when attempting to determining time-outs or other protocol events. The event handler takes care of these details when a protocol routines creates a pending event. The vehicle handler keeps track of vehicles currently in the block. The following sections describe these two mechanisms in more detail.

2.3.1 Event Handler

The event handler is used to manage events such as a time-outs for protocols or vehicle position updates. The basic idea is to keep a list of all pending events in the system. Pending events are defined as events which will occur at a specific time in the future. Pending events are added to the list by protocol routines which specify the time for the event to occur relative to a processor specific time source and a flag. When it is time for a pending event, the event handler sets an appropriate flag to notify the protocol routine which created the event.

Without an event handler, each protocol would require its own time measuring mechanism so that it could set and monitor time for its event. The event handler elegantly abstracts the timing mechanism away from the protocols. Now, protocols can be written without worrying about monitoring interrupts or timer modules which are processor specific. The following two sections describe adding and updating events.

2.3.1.1 Adding an Event

When adding an event to the event handler, a protocol routine provides the event handler with a flag to be set when the event occurs and a time value which denotes when the event will occur. The time value is a number relative to the time till the event handler polls pending events. Every time the pending events list is polled, the time count is decremented by one. When the time count reaches zero, the flag is set for that event. The event handler takes the flag and the time count value and constructs an event record. This record is then placed in the event handler list.

For example, the ping protocol discussed in Section 2.2.6 uses the event handler to manage ping time-outs. If a ping time-out with a time till event count of one and a flag called *PingTimeout* is added to the event list which already has a pending event with a time count of 5 and a flag called *FooTimeout*, it might look as follows:

Event Record 1 TimeCnt: 1 Flag: PingTimeout	Event Record 2 TimeCnt: 5 Flag: FooTimeout	Any other event Records

Figure 2-9. Snapshot of an Event List

2.3.1.2 Updating Event List

At periodic time intervals, the event handler parses the event list and decrements the time count for each active record. Eventually the time count for a particular pending event will reach zero. At this time, the event handler sets the flag which it was provided with when the event was added. If the flag is later poled by the appropriate protocol routine, it will see that the event has occurred and it is then up to the protocol routine to act accordingly.

Consider the previous example, but now one time count has gone by. The event handler has just decremented the time count for each record in the event list. This leaves us with the event list shown in Figure 2-10.

Event Record 1 TimeCnt: 0 Flag: PingTimeout	Event Record 2 TimeCnt: 4 Flag: FooTimeout	Any other event Records

Figure 2-10. Update Event List

The time count for the ping time-out has reached zero. The event handler now sets the flag *PingTimeout* and removes the record from the event list by clearing its entries. The event in record 2 has still not occurred yet so it is left unchanged. Thus, the event handler manages pending events for the protocol routines. As will be demonstrated in the protocols section, it is still the responsibility of the protocol or the main loop routine to periodically poll the flag.

2.3.2 Vehicle Manager

The vehicle manager runs on each block controller and facilitates vehicle control for the protocols. It keeps track of all vehicles currently in the block. The vehicle manager can also be used by the zone controller to create virtual vehicles within the blocks under the zone. This capability is a very useful mechanism for verifying the functionality of the fault tolerant control system. The vehicle manager supports several operations: adding a vehicle and updating vehicle positions, and removing vehicles. The following three sections describes these two operations.

2.3.2.1 Adding a Vehicle

The vehicle manager for a block keeps a list of all vehicles currently in the block's segment. For each active vehicle, the manager stores a record containing an identifier (*VID*), position (*VPOS*) and velocity (*VVEL*). A vehicle is added to this list as a result of a vehicle exchange protocol or a create virtual vehicle protocol which is initiated by the zone controller. The zone controller provides the VID, VPOS and VVEL for the vehicle. The vehicle manager creates a record for the new vehicle, storing these three pieces of information in that record. The other method of creating a vehicle in the block is in response to a vehicle moving from one block to the next. During the transition, the vehicle information is received from the previous block and a vehicle record is then instantiated for this vehicle in the new block. This protocol is discussed in Section 2.4.2.

2.3.2.2 Updating a Vehicle

Storing vehicle information in the vehicle manager list is not enough. Since the vehicles are presumed to be moving, their positions are going to change. For each vehicle in the list, an event is created for the next position change. The time till the position change is determined by VVEL and as such it differs for each vehicle. Since each vehicle has a pending event in the event list (time till next position change), the vehicle manager must keep a separate flag for each vehicle in the list. The flag is set by the event handler when the vehicle's position needs updated.

To update the vehicle position, the vehicle manager must update the VPOS field for the vehicle and then based on the velocity of the vehicle determine the event time until the next position update. An event is then created for this and the process repeats. A block's track has a limited number of positions associated with it. When a vehicle reaches the last position in the block, the vehicle manager removes it from the vehicle list. The vehicle exchange protocol insures that the vehicle is then added to the vehicle list of the next block.

2.3.2.3 Removing a Vehicle

Eventually, a vehicle will need to be remove from the vehicle manager as it leaves the block's segment of track. The vehicle manager takes a VID and scans its list for any vehicle with that VID. The entry is then cleared.

2.4 Protocols

Rudimentary yet robust protocols can be designed for the fault tolerant communication backbone. These protocols are not intended to be used in an actual transportation system. They are intended

to demonstrate the viability of the fault tolerant system. More complex and efficient protocols should be constructed. In addition, the control in these protocols runs between block controllers. In an actual system, the zone controller would coordinate much of the control. The protocols make the following assumptions about the vehicles and the tracks:

- A segment of track under a block controller can hold multiple vehicles.
- Vehicles always move from the left neighbor to the right neighbor.
- A block has one left neighbor and one right neighbor. There are no multiple merging points where multiple tracks meet.
- Block controllers internally divide their tracks into positions. A vehicle occupies one position.
- These positions are long enough that a vehicle can come to a complete stop within one position.

The following sections describe protocols which can be used to create virtual vehicles on the system, coordinate the exchange of a vehicle between blocks and a protocol to keep the zone controller notified of the current state of each block controller. A general overview is given for each protocol including a description of the messages used. Each protocol is then examined for robustness when used in conjunction with the communication backbone.

2.4.1 Virtual Vehicle Protocol

The virtual vehicle protocol is a two message protocol used to create virtual vehicles on block controllers. Creating virtual vehicles provides a means to simulate and test the fault tolerant controller system in a simulated vehicle environment without actually attaching real vehicles to the system.

The protocol is initiated by the zone controller which sends a virtual vehicle create (VVC) message to a block controller. The VVC message provides the VID, VPOS and VVEL for the virtual vehicle in the data field of the message. A VCC time-out is added to the zone controller's event list. Upon receiving a VVC message, the block controller creates a vehicle through the vehicle manager with the provided vehicle traits. The block then responds with a VVC acknowledgment. If a VCC-ACK is not received by the zone controller before the time-out, then the VCC message is resent. The protocol is simple but is also very robust.

2.4.1.1 Robustness

There are three possible outcomes for the VCC message. First, the VCC message arrives at the block controller, the vehicle is created and a VCC-ACK is sent back to the zone controller before the time-out. In this scenario, the protocol successfully completes.

The second outcome occurs when the VCC message never arrives at the block controller. Since a VCC-ACK is then never received by the zone controller, the VCC time-out must resend the original VCC message and the protocol starts at the beginning.

The third case is pictured in Figure 2-11. Notice, that the original VCC message is received and a virtual vehicle is created. However, the VCC-ACK is never received by the zone controller. According to the protocol, the VCC message is resent to the block. However, this results in another virtual vehicle being created. In order to prevent this, the block controller must examine the VIDs of all vehicles in the vehicle list. If an existing vehicle has the same VID as the one in the VCC message, then the message is ignored. This feature of the protocol provides at-most once semantics for virtual vehicle creation.

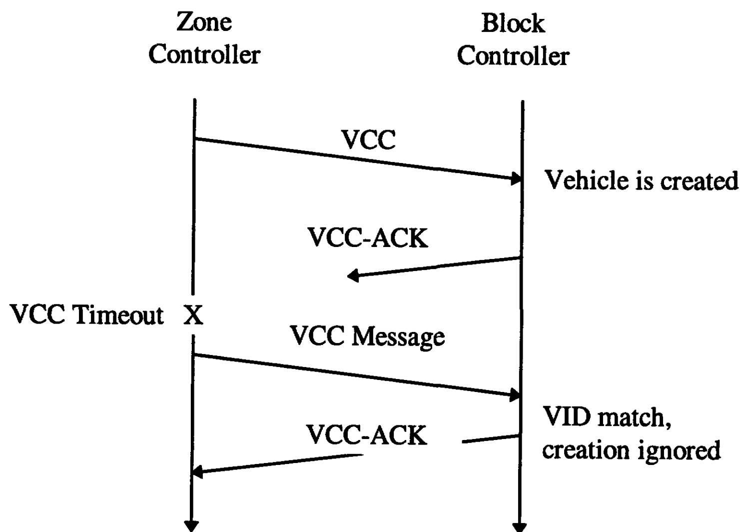


Figure 2-11. Virtual Vehicle Creation Protocol.

2.4.2 Vehicle Exchange

A critical feature of any distributed transportation controller is to be able exchange vehicles between blocks. In particular, it is used to coordinate the exchange as a vehicle moves from one block to the next. In a fault tolerant system, the protocol must be designed to account for messages which might be lost by the communication network.

A very simple vehicle exchange protocol acts in a similar fashion to the virtual vehicle protocol. When a vehicle reaches the end of a block, the controller initiates a vehicle exchange message with a neighboring block (the target). The vehicle exchange message contains an identifier unique to the vehicle in conjunction with the vehicle's velocity. A pending event is then created for a vehicle exchange time-out.

Upon receiving a vehicle exchange message, the target controller determines if the first track position in the block is free or not. If the track is clear, a vehicle exchange acknowledgment message is sent back to the originator and the vehicle information in the exchange message is used to create an entry in the vehicle manager. At this point, this block controller has responsibility of the vehicle and its vehicle manager handles updating the new positions for the vehicle. However, if the track is not clear, the controller does not return an acknowledgment nor does it add the vehicle to the vehicle manager list. By not returning an ACK, the target does not commit to the exchange.

Upon receipt of a vehicle exchange acknowledgment, the originator removes the vehicle from its vehicle manager and ends its responsibility for the vehicle. Or, if an ACK is not received by the vehicle exchange time out, the originator stops the vehicle in the last position of the block, and resends a vehicle exchange message. It could also notify the zone controller of the problem.

2.4.2.1 Robustness

The protocol, while simple is surprisingly robust. Possible failure points occur when messages are dropped by the network, disrupting the exchange.

Assume the original exchange message is lost by the network. After the time-out, the message is transmitted again and the protocol repeats. If the message is received by the target, but the acknowledgment is lost by the network, then the original vehicle exchange is retransmitted as in the previous case. However, the target has already received and possibly added the vehicle. Another vehicle should not be added to the vehicle manager of the target. Actual implementation of the protocol must insure that at most once semantics are enforced at the target end. One possibility involves scanning the vehicle manager's list of vehicles searching of the VID of the vehicle to be added. If the VID is already in the list, then one can conclude that the vehicle was already added.

This functionality can be built into the vehicle manager whenever it adds vehicles. By enforcing at most once semantics, multiple vehicles are not created from one original vehicle exchange message.

Thus, the protocol adequately handles dropped exchange and acknowledgment messages. However, it does create a large amount of transmission traffic if the track is not clear as the originator continually resends the exchange message. The protocol can be improved by adding blocking control where the target later informs the originator when the track becomes clear. This is a slightly more complicated protocol.

3. Implementation

A small scale prototype was implemented to illustrate the design concepts of the distributed fault tolerant transportation controller. The prototype also serves as a test bed for demonstrating the systems' ability to cope with simulated faults. It is hoped that the prototype can be used to develop more complicated control algorithms in conjunction with eventually being hooked up to a physical transportation system.

The overall topology consists of a single zone with two blocks. Circuit schematics were constructed for zone and block controller modules. PCB boards were manufactured based on the associated layouts for the two designs. Due to the symmetry of the chosen topology, the zone controller hardware is identical to the block controller hardware because the communication port constraints are the same. Thus, a board can be configured as either a zone or a block controller depending on its software configuration. For larger scale systems, this symmetry will no longer hold and modifications must be made to the zone controller layout. In particular, as the topology is expanded and more block controllers are added, the zone controller hardware will have to be modified to provide more communication ports. This poor scaling issue is because of the pint to point network topology chosen for block and zone controller communication.

Software was developed to implement the communication backbone. After the backbone was satisfactorily constructed and tested, simple control protocols were implemented to demonstrate the functionality of the fault tolerant system. The following sections discuss the hardware implementation of the controllers followed by the software for the communication backbone, the development tools and finally the control protocols.

3.1 Hardware

As previously mentioned, due to the symmetry of the chosen topology, the zone and block controllers share the same hardware. This section frequently refers to the block controller hardware but it actually applies to both. Refer to Appendix A for detailed schematics of the hardware components described here.

The block controller decomposes into the following modules:

- Main Processor
- Auxiliary Processor

- DUART
- DIP Switches
- LCD Display
- Power Disconnect Indicators

The main processor performs the bulk of the controller work including serving as the communication backbone and performing most of the routing protocols. The auxiliary processor acts as the glorified watchdog for the main processor. It also has the ability to disconnect the control module from any power electronics. A DUART is used to provide two extra communication channels to communicate with left and right neighbors. DIP switches are placed between critical signals, giving the ability to simulate faults by "breaking" a signal. The LCD is used to display pertinent information (e.g. error messages, vehicle simulations). Figure 3-1 illustrates the hardware components of the system. The shaded objects are off-board connections to other controllers. The DIP switch connections are not shown for clarity.

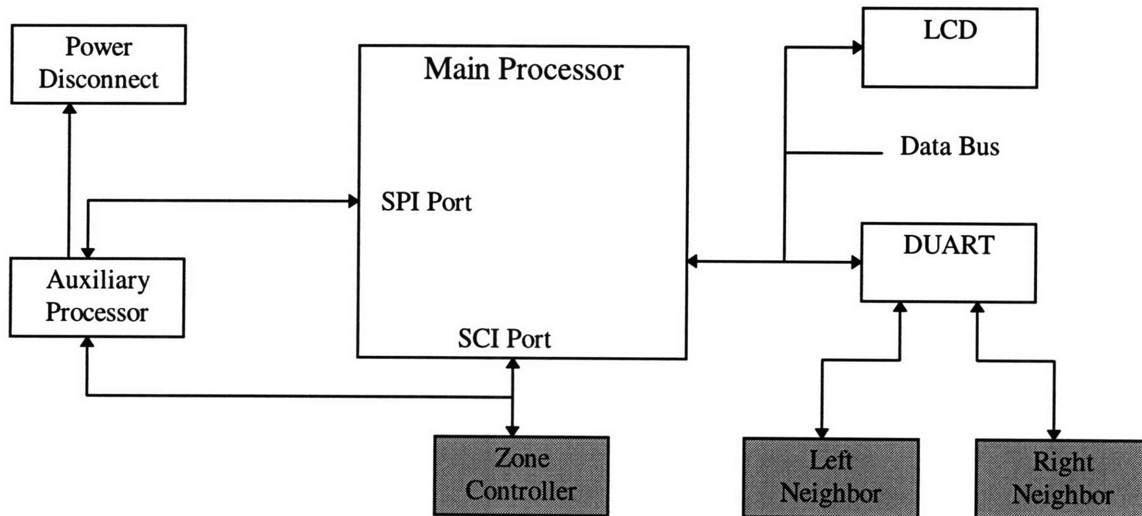


Figure 3-1. Controller Block Diagram

Each of these components will now be discussed in more detail.

3.1.1 Main Processor

The Motorola MC68HC912B32 (the HC12) is used as the main processor because of its combination of processing, memory and peripheral capabilities. The HC12 runs at 14.74MHz with 32Kbytes of FLASH EEPROM, 768 bytes of EEPROM and 1K of RAM. The 32K Flash EEPROM provides a large code space with the potential to later be dynamically updated while the

system runs in place. The 1K of RAM provides enough variable space for the communication backbone structures and protocol routine storage variables.

3.1.1.1 HC12 Communication Capabilities

The HC12 must be able to communicate with the auxiliary processor, the zone controller and the adjacent block controllers (referred to as the left and right neighbors). The HC12 has two hardware ports: an SPI (Serial Peripheral Interface) port and an SCI (Serial Communication Interface) port available for use. The SCI port is used to communicate with the zone controller. The SPI port is used to communicate with the auxiliary processor. Communication with the neighbors is handled via an external DUART.

The SCI port utilizes two signals: RxD (Receive Data) and TxD (Transmit Data) for receiving and transmitting data to and from the zone controller. Complications do arise however, as the auxiliary processor also uses the same two physical lines to receive and transmit messages with the zone controller. This problem is actually dealt with by the software and is discussed in more detail later.

The SPI port is used to communicate with the auxiliary processor. A hardware SPI port acts like a 16 bit register where 8 bits reside on each SPI port. During a transfer, data is rotated from one register to the next. SPI ports use four signals. Two signals are used for receiving and transmitting data. The third is used by one of the SPI ports to generate a clock signal (SCK) to coordinate the exchange. The fourth is a transfer enable pin. Refer to the Motorola documentation on the MC68HC912B32 for more information on how an SPI port works.

3.1.1.2 Background Mode

The HC12 has a nice feature in that it can be placed into a background mode. When in background mode, the HC12 can single step through code and display register and memory contents to a PC. In order to place the HC12 in BDM (background debug mode), a special BGND header was added. The BGND header allows the reset and BDM pins on the HC12 to be controlled by the PC via a cable which plugs into the header. The background mode is used only for development and debug purposes.

3.1.1.3 Programming Code

Most of the code resides inside the 32K Flash EEPROM. In order to program code into the FLASH, it is necessary to provide a 12V programming voltage to the Vfp pin. Through the use of a diode and a resistor this pin is normally +5V until a 12Volt supply is connected to Vfp. This

provides the programming voltage. Unfortunately the Flash has a limited lifetime of about 100 erases. The EEPROM has on the order of 10,000 erases until it goes bad. However, it is only 768 bytes. Thus, code could not be stored inside the EEPROM.

3.1.1.4 Resetting the HC12

There are actually two sources of reset for the HC12. The auxiliary processor has the ability to reset the HC12 since the AP is acting like a glorified watchdog. Furthermore, the PC can assert reset as part of the background debug mode. These two sources are combined through a PAL. If either signal is asserted then the PAL asserts the actual HC12 Reset line, resetting the processor.

3.1.2 Auxiliary Processor

A Motorola MC68HC705C8A (HC05) was chosen as the auxiliary processor for several reasons. First, it is a simple processor. Fortunately, the functionality requirements of the auxiliary processor are also quite simple. Moreover, an HC705C8A emulator board and software development tools were readily available. This was the largest factor. Many other processors (e.g. PICs) could have been used instead.

The HC05 serial peripheral interface port (SPI) is connected to the corresponding SPI port of the HC12, forming a communication link. In addition, the receive line of theHC05 SCI Port is connected to the receive line of the HC12 SCI Port. For a block controller, this line is the transmit line from the zone controller. Thus, if necessary, the HC05 can receive data from the zone controller. However, this feature is currently unimplemented, but the ability exists.

Unfortunately, the HC05 emulator cannot emulate the SPI Port. Without the SPI Port to communicate with the HC12, the auxiliary processor becomes a very poor watchdog. As such, the HC05 was not populated on the final prototype.

3.1.3 DUART

Since the main processor requires 4 communication links and it has only two hardware ports, an off-board DUART was added. A DUART contains two Universal Asynchronous Receiver Transmitters (UARTs). The DUART actually used was the PC16552D by National Semiconductor. It was chosen because it allows baud rates in excess of 560.8kbaud. This is well above the bandwidth needed in the current implementation. This particular DUART gives flexibility should the bandwidth needs increase.

Each UART acts as an independent receive and transmit channel. Each channel has its own interrupt line in conjunction with receive and transmit lines. The receive and transmit lines for channel one are connected to an RJ11 connector which holds a link to the left neighbor. Four-wire telephone cable is used to connect the two data lines plus two ground lines via the RJ11 connector to a corresponding RJ11 connector on the left neighbor. A pull up resistor was placed on the data input line leading into the UART. The pull-up resistor insured that garbage data is not received by the DUART when the connection is open. The data lines for channel 2 of the DUART were configured in a similar fashion except they led to an RJ11 connector linked to the right neighbor.

Both DUART channels interact with the HC12 via interrupts and the data bus. Each channel has its own interrupt line. These lines are connected via a PAL which ORs their values together producing a final signal which is connected to the HC12 interrupt request line (IRQ). Thus, it is the responsibility of the software to actually determine which channel generated the interrupt. The DUART data bus is connected to the HC12 data bus via the HC12's Port A. The HC12 controls the DUART by control lines tied to Port B in conjunction with a separate DUART enable pin which is not part of the data bus. This hardware configuration allows the software to write DUART commands to the high byte of the data bus, and read or write data via the low byte of the data bus.

3.1.4 LCD Display

A Seiko L4042 LCD Display is attached to the HC12 to provide information regarding the current state of the system. It is attached to the HC12 in the same fashion as the DUART. Port A on the HC12 is connected to the LCD's data bus. Command signals for the LCD are tied to Port B on the HC12. Finally, the LCD Enable pin is connected to a separate non-data bus pin on the HC12. With this configuration, the LCD can be controlled by writing the desired command into Port B, the data value (if any) to Port A and then toggling the LCD enable pin.

3.1.5 DIP Switches

Many of the critical signals on the block controller were routed through a set of DIP switches giving the ability to simulate faults by opening or closing connections. Table 3-1 lists all of the signals routed through DIP switches.

For example, if the DIP switch between VDD (+5V) and the power to HC12 is open then the HC12 no longer receives power. This effectively provides a processor failure. The overall system can then be analyzed to see how it handles this failure. Similar analysis applies to each of these signals. In order to simulate a communication fault between the current block controller and its right neighbor, open the DIP switch between the right neighbor input and the DUART SIN 2. Now the block controller no longer receives communication from the right neighbor.

Input to DIP Switch	Output from DIP Switch
+5V Power Supply	Power to the HC12
+5V Power Supply	Power to the HC05
+5V Power Supply	Power to the DUART
Left Neighbor Data Input	DUART SIN1 (serial input channel 1)
Right Neighbor Data Input	DUART SIN2 (serial input channel 2)
Zone Controller Data Input	RxD line to the HC12 and HC05
Data Output to the Zone Controller (TxD)	TxD line to Zone Controller
+5V Power supply	Low Power Detect

Table 3-1. DIP Switch Signals

3.2 Software Implementation of the Communication Backbone

The software implementation of the backbone controller consists of software construction of the router, the message handler, porthandlers, and the ping protocol. Each one of these modules will be discussed in detail with regards to their actual software implementation. Frequent references are made to Appendix B which contains the code developed for the communications backbone. The code was written to run on the HC12.

3.2.1 The Router

The router bridges the gap between destinations (e.g. the zone controller) and the four communication ports on the main processor. The routing table is a data structure four bytes long. Values corresponding to destination sources were designed to map directly as offsets from the first byte of the routing table. When this source is looked up in the routing table, the entry 1 byte off from the start of the table holds the communication link to be used for the left neighbor.

3.2.1.1 Initializing the Router

Initializing the router entails loading the default configuration for the routing table. The default configuration for the routing table is summarized in Figure 3-2. The default configuration matches the expected state of the communication ports and destinations when all the links are alive.

	Zone Controller	**Left Neighbor**	**Right Neighbor**	**Auxiliary Proc.**
Source	**SCI Port**	**UART 1 Port**	**UART 2 Port**	**SPI Port**

Figure 3-2. Default Routing Table

3.2.1.2 Routing a Message

When a protocol routine is ready to route a message, it calls the route message routine. Routing a message requires a destination source and the address of the first byte in the message to be transmitted. The router then uses the destination source as an offset into the table to obtain the destination port. Based on the destination port, the router makes a transmit message call to the appropriate port handler.

3.2.1.3 Updating the Routing Table

Assuming the ping protocol successfully detects communication link failures, the router must support a mechanism for updating its internal routing table. The update routing table routine takes as arguments, the destination source and the new port associated with that source. The updater then uses the destination source as an offset into the routing table, replacing the entry with the new port. This routine is called by either the ping protocol routine after it detects a communication failure or it may be called from the update routing table protocol.

The router also supports the ability to restore entries in the routing table. When a previously down communication link becomes available again, the ping protocol calls the restore entry routine. Restoring an entry takes the destination ID whose port should be restored. The port assigned to the destination is then modified to the default value. The default value is dependent on the actual topology.

3.2.2 Message Handler

The message handler was implemented using a single message queue for the prototype. The queue structure contains the queue, a pointer to the head, a pointer to the tail and the size of the queue. It

adheres to the first in, first out (FIFO) strategy. Messages are added to the tail of the queue and handled from the head of the queue.

The queue was implemented as a buffer whose size is a multiple of an end-message with a head and a tail pointer. The head and tail of the queue wrap around to the front when they reach the end of the buffer. Keeping the queue size a multiple of a complete message insures that messages will not be wrapped around the actual buffer and it reduces the overhead of adding/removing messages since the head and tail do not need to be checked for wrap around until after reading or writing entire messages. However, this decision makes variable message length difficult.

In order to initialize the queue, the head and tail pointers must be initialized to point to the first byte in the queue buffer. The underlying queue buffer is cleared as well. The message handler supports the following routines: adding a message, handling a message, removing a message and forwarding messages.

3.2.2.1 Adding a Message

In its simplest form, the addMessage routine is called by a port handler after it has successfully received a complete message. In an early version of the code, the port handler computed and verified the checksum for a received message. However, this lead to a lot of redundant code since all port handlers must verify the checksum on received messages. Hence, as an optimization, the checksum verification process was moved to the add message routine in the message handler.

Verifying the checksum is accomplished by stripping off the checksum appended to the message. (The checksum is the last two bytes of the message.) A checksum is then computed on the remaining message contents by summing the value of each byte. If the computed checksum does not match the checksum attached to the message then the message has been corrupted, otherwise the message is assumed to be intact. The addMessage routine discards any corrupted message. It does keep track of the number of bad checksums in a variable for statistical purposes. Every checksum failure does result in an appropriate error message being displayed to the LCD.

If the checksum verifies correctly then addMessage routine copies the message (sans checksum) into the message queue starting at the end of the queue. The tail is incremented as each byte of the message is copied. After copying the entire message, the tail is checked for wrap around.

3.2.2.2 Message Forwarding

Before a message is passed to an appropriate handler, the destination ID of the message is checked against the identification numbers used by the controller. Refer to the identifiers section for more discussion about which identifiers a controller responds to. If the destination ID does not match any of these identifiers, the message must be forwarded to the actual destination. The message handler calls the router passing the destination ID and the message as arguments. The router treats the message as an ordinary message in determining which port it should be sent to.

The only messages not subject to forwarding are ping protocol messages. Since the ping protocol is attempting to determine the connectivity of a communication link, they do not use real destination IDs in the message.

3.2.2.3 Handling a Message

Whenever the queue has at least one message, handle message is called by the main loop driver to actually process the message. In order to process a message, the handler must examine the first message in the queue which can be determined from the head of the queue pointer. The message type found in the message dictates the protocol routine which should handle the message.

Currently, this part of the handler routine is implemented as a set of compare and jump statements. Given a message code, the handler checks it against a message type, if they match it jumps to a pre-determined handler for that message type. The process repeats for all known message types until a match is found. If a match is not found, the message is ignored and removed from the queue.

Unfortunately, with this implementation every time a new message type is developed (e.g. by a user writing a new protocol on the system), the handle message code must be modified to include the new message type. A potentially better solution would involve generating a table in RAM at run time which has the mapping of message codes to appropriate protocol routines. The message handler could then provide an interface allowing dynamic generation of entries in the table. Unfortunately, this would use more RAM which is a more valuable resource than the 32K Flash ROM where the code resides. But it is an option that could be considered for handling messages.

Once the appropriate routine has been called to handle the message, the message must be removed from the queue.

3.2.2.4 Removing a Message

A message is removed from the queue by incrementing the head by the size of the message. Afterwards, the head of the queue is examined to check for wrap around. It is set to the first address in the queue buffer if wrap around is necessary.

3.2.3 SCI Port Handler

The Serial Communication Port (SCI) port handler is interrupt driven. The SCI Port itself is configured to trigger transmit empty and receive data interrupts on the HC12. The port is initialized with one stop bit, zero parity bits, transmit enabled, receive enabled, and receive interrupt enabled. The transmit interrupt enable is turned on when data is actually written into the transmit message buffer. The SCI Port is initialized with a baud rate of 19200bps.

The SCI Port handler can be broken down into two modules: interacting with the physical SCI Port and the management structures which sit on top of the port. Interacting with the Port involves writing control registers, and reading and writing the SCI data register.

The management routines are responsible for transmitting and receiving entire messages in conjunction with processing SCI interrupts. Upon receiving an SCI Port interrupt, the SCI port handler must determine if the interrupt cause was an empty transmit buffer or received data. Based on the interrupt source, the interrupt handler calls either the transmit or receive management layer.

The following sections describe both the management portion and the low level port details for transmitting and receiving message

3.2.3.1 Transmitting

When the router decides to transmit a message over the SCI Port, it calls the SCI handler transmit routine. In preparation of sending, the handler normally computes a checksum, appends it to the message, breaks up the high and low nibbles of each packet, and adds a header packet. As discussed previously, as an optimization the checksum is computed by the router since every port handler requires a checksum. The actual size of this new message with the data nibbles separated is now:

2*(EndMessage Size+Checksum Size)+1.

This formula is derived from the fact that the checksum (which is two bytes on the prototype) is appended to the end of the message generated from the protocol routines. Each byte in the resulting message is broken into two packets, doubling the size of the message. Finally, the SCI Port Handler adds a header byte to the front of the message to identify the start of a message.

After preparing the message for transmission, the message is added to the TMB by copying each byte in the message to the end of the TMB, pointed to by the TMB tail. The tail is advanced by the number of bytes in the prepared message. If the message was placed into an empty buffer then the transmit enable for the SCI Port is turned on to allow the message to be transmitted. Otherwise, the transmit enable would already be on since there were message bytes previously in the buffer.

The low level byte transmission process is interrupt driven. The current byte to be transmitted from the TMB is placed into the data register of the SCI Port. Once this byte begins being transmitted, the SCI Port sets a transmit buffer empty bit which in turn generates an SCI interrupt on the HC12. The SCI interrupt handler verifies that the transmit buffer empty flag has been set and then calls a routine to transmit the next byte. To transmit the next byte, the management layer of the port handler advances the TMB head pointer and checks for wrap around. If there are still bytes in the TMB, then the value pointed to by the head is then loaded into the SCI data register and the process repeats. On the other hand, if the TMB is empty, the transmit empty interrupt enable bit is turned off reduce unnecessary interrupts.

3.2.3.2 Receiving

Receiving messages is more complicated than transmitting. Received bytes must be assembled into messages and stored in the RMB. When a message is completely assembled, the port handler notifies the main event driver that it now holds a complete message.

Message bytes are received from the low level SCI Port via the data register. Reception of a message byte triggers an SCI receive interrupt. The SCI interrupt handler verifies that a SCI receive interrupt has occurred and calls the receive message portion of the SCI port handler. The received message byte is then read in from the SCI data register.

At this point, the management layer of port handler steps in and takes the received byte. The byte is inserted into the RMB. The RMB has three variables associated with it: head pointer, start pointer

and the tail pointer. The head pointer always points to the first valid byte in the buffer. The start pointer always points to the first byte of the message currently being received and the tail points to the next free spot in the buffer.

As the message bytes are received, the handler assembles them into messages, storing them in the RMB. First, the byte must be checked to see if it is a header byte. If it is in fact a header byte, the tail of the RMB is reset to the start of the current message. If a message is partially received and a header byte arrives then there was a transmission problem with the partial message. By resetting the tail, the contents of the partial message are discarded. Since the communications backbone does not guarantee message delivery, this is an acceptable solution. After resetting the tail, the header byte is discarded and the duties of the port handler are done for this byte.

If the received byte is not a header byte, it must be integrated into the stored message. Since the transmit portion of the SCI converts a byte of data into two bytes (high nibble, low nibble), the receive portion must reconstruct the original data value. A nibble flag is used to determine if the received byte contains the high or low nibble of a data byte. The nibble flag is set to denote a high nibble after a header byte is received. Thus, for the next received byte, the port handler treats it as the high nibble of the first data byte. The nibble flag is then toggled such that the next received byte becomes the low nibble of the data byte. When the high and low nibble for a data byte are received, they are combined and the resulting byte is placed into the RMB and the tail is advanced. The process repeats for the next data byte.

By keeping track of the number of completed data bytes received and comparing that number to the number of bytes in the message, the port handler can determine when a completed message has been received. Upon completion of the message, the port handler increments a variable which keeps track of the number of received messages in the SCI RMB. The main loop driver which is discussed in Section 3.5. polls this variable periodically. When the variable is non zero, the loop driver instructs the port handler to give the oldest message to the message handler.

3.2.4 DUART Port Handlers

DUART port handlers are responsible for interacting with the external DUART. Since the DUART has two channels for receiving and transmitting data, there are two port handlers associated with it. Both port handlers have exactly the same functionality with the exception that

one port handler interacts with channel 1 on the DUART and the other port handler interacts with channel 2 on the DUART. With regards to the overall topology, for a block controller, channel 1 is a direct link to the block's left neighbor and channel 2 is a direct link to the block's right neighbor. The following discussion about a DUART port handler applies to both port handlers.

The DUART port handlers must be able to send and receive data bytes to the DUART in conjunction with properly handling interrupt requests from the DUART chip.

3.2.4.1 Initializing UART 1 Port Handler

Initializing the DUART port handlers involves initializing the appropriate TMBs and RMBs and their associated pointers. Furthermore, each channel on the DUART must be initialized as well. A DUART channel is initialized with a baud rate of 19.2 kbaud in order to be compatible with the baud rate chosen for the HC12. As with the SCI Port, the DUART channels use 1 stop bit, eight data bits and no parity bit. Thus, a packet for the DUART is defined as a byte.

3.2.4.2 Handling DUART Interrupts

A single DUART interrupt handler is used for both DUART port handlers. Two DUART events cause a single interrupt on the HC12, informing the system of an event on channel 1 or an event on channel 2. The DUART port handlers are responsible for IRQ interrupts as these are generated by the DUART. The interrupt handler must first determine which DUART channel caused the interrupt. This determination is accomplished by reading the interrupt identification register (IIR) for each channel from the DUART. For a given channel, the IER lists any current interrupts. In particular, the interrupt handler checks to see if a transmit holding register or a received data available interrupt has occurred. If the IER for a channel shows that one of these interrupts has occurred, the interrupt handler calls the appropriate transmit or receive routine for the port handler in charge of the particular channel.

3.2.4.3 Transmitting

The management layer of the DUART port handler is similar to that of the SCI port handler. When the router has a message to transmit over a DUART channel, the appropriate DUART port handler is given the message. The port handler then breaks the high and low nibble of each data byte into independent bytes. Finally, the header byte is tacked on to the beginning of the message and the message is placed into the port handler's TMB. The TMB is 90 bytes long so it can hold two messages where each message waiting to be sent has a size determined by the formula in Section

3.2.3.1. Adding a message to a full TMB currently results in the message being dropped. Due to the large size of the TMB relative to the infrequency of transmitted messages it is an unlikely occurrence. Since this layer does not guarantee message delivery, a more complicated scheme does not seem worth the effort. If the TMB is initially empty, the transmit holding register empty (TRE) interrupt is enabled on the DUART for the particular channel. Thus a DUART byte transfer causes an interrupt which forces the DUART interrupt handler to call the transmit byte routine.

The low level transmission of bytes via the DUART is more complicated than the SCI Port. Assuming the transmit holding register for the particular channel is empty, the data must be sent over the data bus. This simple fact complicates usage of the data bus. Any other routine with code that uses the data bus must place within its code a critical section with interrupts disabled around the data transfer code. Otherwise, transmitting a byte over a DUART channel may actually interfere with the other routine's transfer since both routines use a shared data bus. Fortunately, the LCD is the only other module that uses the data bus.

In order to give the data byte to the DUART, the port handler loads the data byte into Port A of the HC12. Commands for writing the data byte to the data register for the DUART channel are loaded into PortB. The DUART chip select pin is then asserted. At this time, the DUART reads the commands and the data value is loaded into the transmit buffer. The TMB tail pointer is then incremented to the next data byte with an appropriate wrap around check. When the byte has been transmitted, the DUART will generate a transmit holding register empty interrupt which causes the process to repeat again. After transmitting a byte, the port handler turns off the TRE interrupt if the TMB becomes empty. This action prevents unnecessary interrupts from occurring on the HC12.

3.2.4.4 Receiving

The management layer of the receive portion of the DUART port handler is similar to that of the SCI port handler. Refer back to section 3.2.3.2 for details about how the management layer processes a received data byte.

Receiving a data byte begins when the DUART generates a received data available (RDA) interrupt. This causes an HC12 IRQ interrupt. The DUART interrupt handler determines which channel actually has the RDA and calls the appropriate port handler receive routine. The handler

generates a command byte in PortB. Toggling the chip select for the DUART results in the newly received data byte being placed onto Port A. The handler then reads in the new data byte and stores it in the RMB.

3.2.5 Ping Protocol

Implementing the ping protocol requires four separate protocol routines. The first one generates ping requests. The ping requests are processed by a ping request handler which responds with a ping response. In turn another routine handles ping responses by marking that they were received. Finally, the last component of the ping routine handles the ping time-out. These four routines are now discussed in more detail.

3.2.5.1 Ping Requests

Ping request messages are generated at periodic intervals and sent to each communication port. A ping request message currently contains no other message information besides the message code (PingReq), the ID of the message destination, the message source and the message size. The message data portion is currently left unfilled. The message destination is not the ID of the destination controller. Instead, since the protocol has knowledge of the actual communication links used, it contains an identifier corresponding to the port which lies at the other end of the communication link. In the same sense, the message source contains an identifier for the actual port the sender is giving the message to for transmission. Valid message source and destination identifiers include UART 1, UART 2, SCI Port, and SPI Port. Table 3-2 summarizes the possible message source and destination pairings based on the physical topology that has been established for the prototype.

Message Source	Message Destination
UART 1	UART 2
UART 2	UART 1
SCI Port	UART 1
SCI Port	UART 2
SPI Port	SPI Port

Table 3-2. Valid Ping Source and Destination Pairings

For a block controller, a ping request message is sent to each of the four port handlers with an appropriate message source and message destination for that pairing.

In order to keep track of which ports have responded to the ping and which ones have not, the ping request routine also prepares a variable called ALIVE. Each bit in ALIVE corresponds to a communication port. The idea is to record which ports have replied to the ping by setting that bit. So by starting a ping request, the ALIVE variable must be bit cleared, denoting each link as down until proven otherwise. As ping response are received, the appropriate bits in ALIVE are set.

Finally, the ping request routine creates an event for the ping time-out. Ping time-outs currently happen every .25 seconds. The flag for the ping time-out is actually bit seven of the ALIVE variable as this bit is currently unused. Thus, the address of the ALIVE variable is added to the event record in conjunction with a flag bit mask which only sets bit seven when the event occurs. The ping time-out event is then placed in the event handler.

3.2.5.2 Handling Ping Requests

Upon receiving a ping request message, the controller must acknowledge the ping by generating a ping response. A ping response message has a message code for the ping response. The message source and message destination fields are the opposite of what they were in the ping request. For example, if the destination on the ping request was UART 1 and the message source was UART 2, then for the ping response, the destination becomes UART 2 and the source becomes UART1. This information allows the recipient of the ping request to easily determine which communication port to send the response to. The data field of the ping request message is left empty. The ping request message is then sent back through the port it was received from.

3.2.5.3 Handling Ping Responses

Upon receiving a ping response message, the ping response handler checks the message destination. The corresponding bit in the ALIVE variable is then set, acknowledging that the communication link is alive. Returning to the example, if the initial ping request went to the UART 1 port, then the message destination field found in the ping response should be UART 1. This would result in the controller recognizing that the UART 1 port is alive.

3.2.5.4 Handling Ping Time-outs

When the event handler sets the ping time-out flag, the main loop driver eventually poles the flag which is now set and calls the ping time-out handler. At this time, if a ping response has not yet been received for a communication link, the link is considered down. This process is quickly performed by checking the bits in the ALIVE variable. If the bit for a particular port has not been

set then a ping response was not received. Currently the system prints an error message to the LCD stating that the communication link is down. Another protocol must then be started to actually update the routing table with a new communication link for the actual destination which was connected to the broken link. If the bit is set for a link then no error is reported and the routing table remains unchanged.

Pinging the communication links is a continual process. The last action of the ping time-out handler is to generate another ping request by calling the ping request routine.

3.3 Development Tools

3.3.1 Event Handler

The event handler has two primary responsibilities: it allows protocol routines to add events and when these pending events occur, the event handler sets an appropriate flag.

For each event, the event handler stores an event record. This event record is a five byte data structure where the first two bytes form an address for the flag associated with the event. The next two bytes form a time count value. The time count value is a 16-bit number which is used to determine when the event has occurred. It is expressed in terms of the number of delay interval counts. The last byte is actually used as a bit mask for the flag. The bit mask is used to determine which bits in the flag byte should be set when the event occurs.

Given an event record for each pending event, the event handler keeps track of all these records in an event buffer. The buffer size is a multiple of the event record size to insure that only complete records are stored in the buffer. The buffer currently holds ten event records so it is fifty bytes long. This size has proved to be adequate in the prototype. Based on the delay interval count, the event handler updates the time counts for each pending event. When an event record in the buffer gets a time count of zero, the event handler must set the appropriate flag bits.

The following sections describe the actions of the event handler including the determination of the delay interval count, adding an event, and the event handler interrupt.

3.3.1.1 The Delay Interval Count

An interrupt is generated at fixed intervals via a compare module on the HC12. The compare module holds a 16-bit value and continually compares the value to the running 16 bit system timer. When the values match, the interrupt is triggered. The timer is incremented every 134nsec. If an interrupt is desired every x seconds, the number of timer increments until the interrupt is x/134nsec. If this number is added to the current timer value and stored in the compare module, then an interrupt will occur in x seconds. For the event handler, the delay interval count can be changed by calculating a new value to be added to the timer module. For the prototype, the designed delay interval count is 500useconds. Hence, every 500usecs an interrupt is generated for the event handler. It is important to note that the value of the delay interval count dictates the number protocol routines load their time count values with.

3.3.1.2 Initializing

Initializing the event handler involves clearing all the records in the event buffer. Furthermore, the event handler interrupt must be initialized. This is done by setting the timer module 5 in output compare mode on the HC12. The interrupt generated by this timer module is initially left off since the buffer has no pending events after initialization. Adding an event to an empty event buffer turns on the interrupt.

3.3.1.3 Adding an Event

When a protocol routine wishes to add a pending event, it calls the add event routine. The arguments for adding an event are a two byte address for the event flag, a two byte time count and a one byte flag mask. These arguments must be pre-loaded into a temporary event record before calling add event.

In order to add the event, the event handler must parse through the event buffer to find an empty record position. Empty records have flag address fields of 00h. If an empty record slot is found, the record data is copied into the empty slot. Finally, if there were no elements previously in the buffer, the EH event delay interval is added to the current timer value and loaded into the compare module and the interrupt for the compare module is turned on.

If the event buffer happens to be full, then adding an event will result in an event buffer full (*EHBufferFull*) error code being returned. Interrupts are disabled while modifying the event buffer.

This action protects against the event handler interrupt occurring while modifying the event buffer which would lead to an inconsistent state of the buffer.

3.3.1.4 Event Handler Interrupt

Handling an event handler interrupt involves scanning the buffer for pending events. Any event whose address field is not 00h is a pending event. For each of these events, the time count is decremented. For each event whose time count reaches zero, the flag mask in the event record is combined with the address of the event flag to determine which bits in the flag should be set. The event handler then sets these bits and clears the record from the buffer by setting the record's flag address to 00h.

As an optimization, if there are currently no events in the buffer, the event handler interrupt routine turns the EH interrupt off. This prevents unnecessary interrupts from occurring in the system. Of course, this implies that add event always turns the interrupt back on.

3.3.2 Vehicle Manager

The vehicle manager keeps track of all the vehicles currently in the block by storing a vehicle record for each vehicle in a vehicle manager buffer. The vehicle manager takes responsibility for maintaining the current position of each vehicle within the block.

A vehicle record is a five byte data structure where the first byte contains a vehicle identification number (VID), the next byte is the vehicle's position (VPOS) relative to the current block and the following two bytes are used to store the vehicle's velocity (VVEL). The last byte was not initially planned on during the design phase. It is a variable used to denote a blocked vehicle (VBLOCK). If a vehicle is waiting for the next track position to become free, the vehicle manager marks the vehicle by setting the blocked flag in the record. These records are stored in the vehicle manager buffer. The buffer currently holds eight records. So a block can only hold eight vehicles at once on the prototype. The vehicle manager also reserves a bit flag for each record in the buffer. These flags are actually used to determine when a vehicle needs its position updated.

On the current prototype, the segment of track controlled by a block controller is divided into ten positions. The position field is simply a number which varies between 0 and the maximum number of positions in a block (10). The vehicle velocity is actually a special number tied to the prototype. Instead of storing an actual velocity value, it currently stores the number of event counts until the

position should change. The vehicle manager is responsible for using this number to create a vehicle update event. By directly correlating the value stored in VVEL with the value used by the event handler, it is possible to have higher level routines change the velocity of a vehicle and have this automatically change the rate at which the vehicle position is updated.

The vehicle manager supports the following routines: initialization, adding a vehicle, updating vehicle positions and removing vehicles. Initializing the vehicle manager is quite simple. The manager clears all vehicle records in the buffer by setting each VID field to 00h. The following two subsections will discuss adding, updating vehicles and removing vehicles.

3.3.2.1 Adding a Vehicle

When a protocol routines decides to add a vehicle to the block controller (for example after receiving a vehicle from a neighbor), it invokes the add vehicle routine. Add vehicle requires the VID, VPOS and VVEL for the new vehicle. These arguments are passed into add vehicle via a temporary vehicle record. If the vehicle is successfully added, a *VehicleAdded* flag is returned. Otherwise, a vehicle blocked or track full message may be returned.

In order to provide at most once semantics for the vehicle exchange protocol, the add vehicle first scans the list for a vehicle that shares the same VID as the one to be added. If there is a match, a VehicleAdded flag is returned but the vehicle is not re-added to the list. If the vehicle is not already in the list, the add routine checks to see if VPOS for the vehicle is already occupied. The position status variable has a bit set for each occupied position. If the bit for VPOS is set, then add vehicle returns a position occupied flag and the vehicle is not added.

Actually adding the vehicle involves scanning the vehicle manager list, searching for an empty record in the list. Again, empty records have VID tags of 00h which makes this process straightforward. The vehicle information from the temporary record is then copied into the first available free slot in the buffer. Interrupts are turned off while modifying the vehicle record buffer to maintain a consistent state, insuring that no two routines are modifying the buffer at the same time.

Once the vehicle has been added, the vehicle manager adds an event to the event handler where the event is a vehicle position update. The time count used is simply the VVEL value for the vehicle.

The flag set by the event handler is actually the bit flag the vehicle manager has reserved for this vehicle buffer slot.

3.3.2.2 Updating Vehicle Positions

One or more bits in the vehicle status register are set by the event handler when it is time to update a vehicle position. The bits set in vehicle status correspond to the entry in the vehicle list where the vehicle record to be updated exists. It is up to the main loop to pole the vehicle status flags.

Updating vehicle positions involves scanning the vehicle list, and finding any non-empty vehicle record whose corresponding vehicle status flag is set. Normally, the vehicles position is updated by one, and the position status variable is adjusted to reflect the new occupied position. A new event is created and added to the event handler for the next position update with the event time count proportional to the vehicle's velocity. The flag address is the vehicle status variable and the flag mask selects the entry position in the vehicle list. In other words, the nth entry in the list has the nth vehicle status bit set by the event handler.

However, there are two special cases to be considered: first, the vehicle is already in the last position of the block and second, the next position is already occupied. If the vehicle occupies the last position in the block, the vehicle manager initiates a vehicle exchange protocol. The vehicle is then left in the last position until it is removed by the protocol. In the second case, the vehicle position cannot be updated because the next position is already occupied. The vehicle record is marked as blocked an event for the next position update is not done.

Finally, after the update routine has updated the appropriate vehicle records, another pass is made through the list. This time, the manager only looks at vehicles which are blocked. If their desired update position is now free, the vehicle position is updated, the blocked flag is cleared, the position status variable is updated and an event is added to the event handler for the vehicle's next position update.

3.3.2.3 Removing Vehicles

The vehicle manager also supports a remove vehicle routine which is used by the vehicle exchange protocol. When it is time to remove a vehicle from the block, this routine is invoked with the VID for the vehicle as an argument. The list is scanned starting at the beginning checking each record

VID with the target VID. If a match is found, the position status bit for the vehicle's position is cleared and the VID field is set to zero, effectively removing the vehicle.

3.4 Control Protocols

Two control protocols were implemented to test the viability of the system as a whole. A virtual vehicle create protocol is used by the zone controller to place vehicles into the system. Vehicle exchange protocols are then used by the blocks to exchange vehicles.

3.4.1 Virtual Vehicle Create

The create virtual vehicle protocol (VVC) has five associated protocol routines. Two are used to generate VVC and VVC-ACK messages. Each message type in the protocol has its own handler. Finally a routine handles the VVC time-out event. Only one pending virtual vehicle create can be created at a time. The protocol routines share a one byte VV flag. Bit 7 of this flag is set by the event handler when a time out occurs. Bit 0 is used to store the reception of a VCC-ACK. These five routines are now discussed.

3.4.1.1 Virtual Vehicle Create Message

When initiating a virtual vehicle create message, the caller loads the temporary vehicle record with the desired vehicle attributes including VID, VPOS and VVEL. The VVC generator takes this information and produces a message as follows;

Vehicle Create Code	ID of Dest.	ID of Sender	MessSize	VID	VPOS	VVEL (2 bytes)

Figure 3-3. Virtual Vehicle Create Message

This created message is then passed on to the router for actual transmission to the destination. Finally, the routine also creates an event for a VCC time-out. Currently VCC time-outs happen .25 seconds after the VCC message is transmitted. This value is controlled by a VCC constant called VVMTimeCnt and is the time count value used in the event record. The virtual vehicle flag address and bit mask are also loaded into the event record such that bit 7 of the VV flag will be set when the time out occurs.

3.4.1.2 VCC Acknowledgment

The VCC handler takes the VCC message and creates a vehicle record out of the vehicle characteristics in the data portion of the message. An attempt is made to add the vehicle to the

vehicle manager. If the manager cannot add the vehicle because the position is blocked, a VCC-ACK is not returned. Otherwise, a call is made to generate a VCC-ACK.

When a vehicle is successfully added, a VCC acknowledgment message is sent to the sender. The data field of the VCC-ACK is left blank. The sender is determined by examining the original message. The source ID is taken from the destination field of the VCC message.

VCC-ACKs are handled by setting bit zero of the virtual vehicle flag. This bit is examined by the time-out handler in order to determine if the VCC message succeeded or not.

3.4.1.3 VCC Time Out

Processing a VCC time out event involves checking bit zero of the virtual vehicle flag. If the bit has been set, then a VCC-ACK has been received, the time-out clears the virtual vehicle flag and returns. Otherwise, an VVC failure message is displayed to the screen indicating that the vehicle was not created or that a reply was not received.

3.4.2 Vehicle Exchange

In its initial implementation, the vehicle exchange protocol is very similar to creating a virtual vehicle. It is introduced as a separate protocol with the understanding that it will be modified and expanded in the future. It is a two message protocol with five associated routines for generating and handling the two messages in conjunction with a time out handler. Only one vehicle exchange can be initiated at a time between two blocks.

3.4.2.1 Vehicle Exchange Message

When the vehicle manager updates the position of a vehicle already in the last track position for the block, it initiates the vehicle exchange (VE) protocol. The routine for generating a VE message takes in a pointer to a vehicle record which contains the vehicle's statistics. Since vehicles always flow from the right to left, the message destination is always the identifier for the block's right neighbor (LNID). The source, is then the identifier the block uses to identify itself to the left neighbor which is MYIDLN. The VID and VVEL for the vehicle to be exchanged are placed in the data field of the message. The VID and VVEL for the vehicle are also stored in two variables set aside by the vehicle exchange protocol. If the message needs resent later, these values are used to reconstruct the vehicle data.

The message is then passed to the router with the LNID as the destination. An event is then added to the event handler for a vehicle exchange time-out. The time-out duration is currently .25s and is controlled by a constant, VehExchCnt.

3.4.2.2 Handling a Vehicle Exchange Message

Upon receiving a vehicle exchange message, the VID and VVEL are extracted from the data field and placed into a temporary vehicle record. Since the vehicle is going to be entering the block, the position field is set to the first position of the block's track. Once the vehicle record is prepared, a call is made to the vehicle manager to add the vehicle. If the vehicle is added successfully a VE-ACK is generated, otherwise no response is sent.

3.4.2.3 Vehicle Exchange Acknowledgment

The vehicle exchange acknowledgment is constructed using information from the original VE message. The message code is a VE-ACK. The destination ID is the source ID in the VE message. The source ID for the block is the destination ID in the VE message. The VID of the added vehicle is then placed in the data field. The resulting message is then sent to the router for transmission.

3.4.2.4 Handling a Vehicle Exchange Acknowledgment

When a VE-ACK is received, the controller sets bit zero of the vehicle exchange flag. This bit is later examined by the time-out handler. Furthermore, the controller must now remove the vehicle from its vehicle manager as the vehicle is no longer in the block. The VID of the vehicle is removed from the VE-ACK data field and passed along as an argument to the vehicle manager's remove vehicle routine.

3.4.2.5 Vehicle Exchange Time-out

When processing a VE-time-out, bit zero of the vehicle flag is examined. If the bit is set, then an acknowledgment has been received and processed. In this case, the vehicle exchange flags are cleared and the routine terminates. Otherwise, an acknowledgment was not received and the protocol must be started again. The VID and VVEL of the vehicle to be exchanged are extracted from their storage variables (they were stored during the generate vehicle exchange routine) and placed into a temporary vehicle record. The time-out routine calls a generate vehicle exchange message which begins the protocol over again.

3.5 Main Loop

The main loop for both block and zone controllers is based on poling and interrupts. The main loop iterates through all the flags which require polling. If a flag is set, then a handler is called for that flag. When the end of the list is reached, the main loop driver starts over again. Table 3-3 summarizes all of the polled flags, the source of the flag and the routine called to handle the flag. The source of a flag is the routine responsible for setting the flag. The handling routine is responsible for actually acting on the flag.

Poled Flag	Source	Handling Routine	Description
RxSCICnt	SCI Port Handler	AddSCIMessage	SCI Port handler has at least one received message. Add it to the message queue.
RxUA1Cnt	UA1 Port Handler	AddUA1Message	DUART Channel 1 has at least one received message. Add it to the message queue.
RxUA2Cnt	UA2 Port Handler	AddUA2Message	DUART Channel 2 has at least one received message. Add it to the message queue.
Qsize	Handle Message	HandleMessage	The message queue has at least one message. Handle a message.
ALIVE.7	Event Handler	HanPingTimeout	A ping response time-out has occurred. Process it.
VVMFlag.7	Event Handler	HanVVTimeout	A virtual vehicle create time-out has occurred. Process it.
Vehicle Status	Event Handler	UpdateVehicles	At least one vehicle in the vehicle manager needs updating.
VehExch	Event Handler	HanVehExchTimeOut	A vehicle exchange time-out has occurred. Process it.
BannerFlag	Event Handler	UpdateBanner	Update the banner display.

Table 3-3. Summary of polled variables for loop driver

In addition to the poled loop flags, there are several interrupts which occur in the system Table 3-4 summarizes these interrupts and their respective handlers

Interrupt	Handler	Description
SCI Interrupt	SCI Port Handler	A receive data or transmit data empty interrupt has occurred.
IRQ	DUART Port Handler	Channel 1 or 2 has a receive data or transmit empty interrupt.
Timer Compare Module 5	Event Interrupt Handler	Time to update the event list.
Timer Compare Module 7	LCD Interrupt Handler	Print next character to LCD.

Table 3-4. Summary of interrupt sources

4. Further Study

There are many areas of further study in which this thesis could be extended. The following subsections touch upon several of these areas. They are by no means inclusive.

4.1 Expanding the Controller Algorithms

The scope of the thesis quickly converged to designing and constructing a fault tolerant communication backbone upon which high level control algorithms and protocols could be written. The control protocol actually designed and implemented was quite simple and was used primarily to demonstrate the correctness of the communication network. Much more research can be done on developing more complicated protocols for managing vehicles including merging and vehicle re-routing.

In particular, more control should be placed at the zone controller level. Since there are fewer zone controllers (by an order of magnitude) it makes sense to place the control at their level. The zone controllers can then be designed with high reliability. The block controllers would then have act as monitors, relaying vehicle status back to the zone controller and responding to commands from the zone controller. There are many block controls. By keeping their responsibilities light, they can be designed more cheaply without an emphasis on high availability.

4.2 Zone Controller Voting

The zone controller is a critical component of the system. Block controllers rely on the zone controller for guidance when they detect failures. If the blocks lose contact with their zone controller, they can always have a prepared emergency back up such as stopping all vehicles. But this is not very graceful. Hence, it is important to ensure that the zone controller behaves properly and has high availability.

In order to make the zone controller more reliable, a level of redundancy must be added. A common technique involving passive redundancy is to use multiple controllers which simultaneously control the system in a cooperative fashion [1]. For example, the zone controller could be implemented with three processors. A voting scheme would then be utilized to determine decisions sent to the block controllers. For voting, all three zone processors receive information (e.g. a track processor has faulted). If everything worked perfectly, all three processors would agree on the same course of action. However, what if one of the zone processors faults, and does

not respond or gives a bad decision. With the voting scheme, as long as the other two processors agree on a course of action, a majority vote will hold and the correct course will still be taken. As a result of voting, a zone controller can withstand the loss of one of its processors without bringing the entire zone controller down. The following figure illustrates what the zone controller would look like.

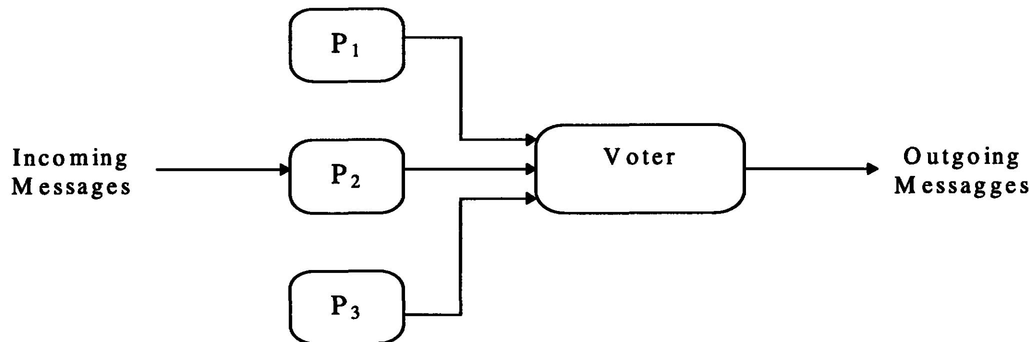


Figure 4-1. Voting on the Zone Controller.

Using a voting system in conjunction with multiple processors is a technique known as Byzantine architecture [5]. The problem with a Byzantine architecture involves synchronizing all three processors to produce outgoing messages at the same time. The implementation of this type of voting mechanism would be very difficult and was beyond the scope of this thesis but does warrant more research.

Another interesting approach for voting on the zone controller is to allow the block controllers to actually act as the voter. As in the previous example, the zone controller could consist of three processors. However, now each processor sends what it thinks the proper action is back to the block controller. The block controller examines the three responses and chooses to act on the majority if the responses differ. This type of voting mechanism has a huge advantage when applied to the system developed in this thesis because it requires no hardware changes other than additional communication links. Three zone controllers could act as the three processing units. A block controller sends messages to all three zone controllers and waits for responses from all three. If one of the zone controller units faults or goes down, the block will still get good information from two units and this forms a majority.

4.3 Adding a Central Controller

There is actually another layer which sits on top of the zone controllers. A single central controller could be designed which had a communication link with each zone controller. The rest of the network would remain unchanged. Figure 4-2 illustrates what the new network would look like with the addition of a central controller.

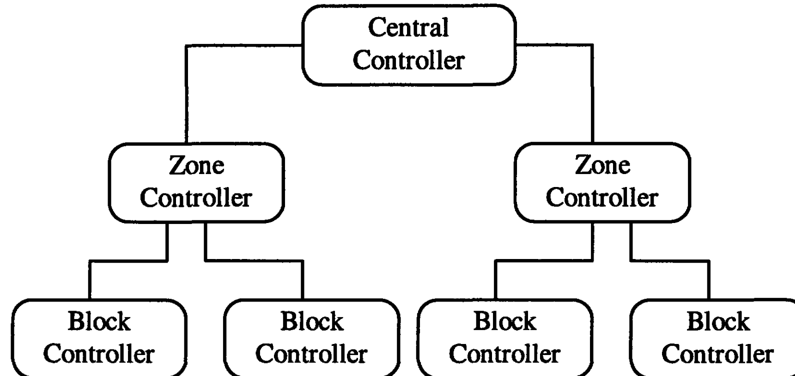


Figure 4-2. System hierarchy with a Central Controller.

The central controller now has knowledge of the entire transportation system. With this information, it could construct a global model of the transportation system. This global model has several good uses. First of all, any error or fault messages can be propagated up to the central controller. Human monitors could be at the central controller and take appropriate action based on the type of error or fault (e.g. sending a repairman to a Block controller and replacing it). Furthermore, a software model could be constructed at the central controller end which modeled the physical transportation. The model could display vehicles which correspond to real vehicles on the track along with their movements. This software model would facilitate the analysis of traffic and merging algorithms.

4.4 Statistics

Despite the construction of a fault tolerant controller, an in depth performance analysis is essential to measuring the effectiveness of the system as a whole. Moreover, extended analysis and testing of the system would allow us to determine valuable fault tolerant statistics such as the mean time to failure (MTTF) and the mean time between failures (MTBF).

4.5 Dynamically Download New Code

The ability for block controllers to download and receive code updates while the system is in operation would be a nice but not an essential feature of the system. It would allow the system to update the code without losing service.

4.6 Adding the Watchdog Circuit

Due to the problem with the HC05 hardware emulator, a watchdog circuit was not created for each block and zone controller. It is still an important feature of the system. The watchdog provides another source for detecting processor failures. In addition, the watchdog could be constructed to receive messages from the zone controller by the same communication lines as the main processor. In the case of a main processor failure, the watchdog can be instructed to reboot the main processor.

5. Conclusion

In conclusion, a distributed fault tolerant transportation controller was designed and a small scale prototype was implemented. Emphasis was placed on a high reliability communication backbone. The actual control protocols were intended to demonstrate the functionality of the backbone. Together, they produced a system which could quickly detect and handle processor and communication errors.

The system needed to be able to quickly detect processor and communication failures. A ping protocol was developed for this purpose. With all the elements in the network running the ping protocol, controller failures are immediately noticed by at least two neighbors and the zone controller because the failed processor no longer responds to pings. In the case of a communication line failure, using the ping protocol, modules at both ends of the link quickly detect the communication failure and can notify the zone controller.

Noise and disturbances on the communication lines are dealt with by port handlers using checksums and header packets. Effective use of acknowledgments and time-outs in the message protocol design stage handles the case where messages are not delivered.

Once the errors have been detected, the system had to handle them. Dynamic routing tables and message forwarding by the message handlers were introduced to cope with handling communication failures. By updating the routing table, messages can be sent over different communication lines and forwarded by other network elements until the message reaches its final destination. When the communication link is repaired, the routing table is updated, restoring communication on the link.

Adding the zone controller and keeping it up to date with a global view of all the block allows processor failures to be dealt with. By placing control at the zone controller level, if a block fails, the zone controller can determine an appropriate course of action and instruct neighboring block controllers appropriately.

It is hoped that this research will become part of a much larger system with more complicate control algorithms at the zone controller level in addition to being attached to an actual

transportation system. The thesis has shown that a distributed fault tolerant controller can be constructed to meet the needs of transportation systems.

6. References

[1] Cho, Young and Bien, Zeungnam. "Reliable control via an additive redundant controller." International Journal of Control, vol. 50, July 1989. p 385-398.

[2] Dooling, Dave. "Technology 1996: transportation". IEEE Spectrum, vol. 33, January 1996. p. 82-86.

[3] Gray, Jim. "High-Availability Computer Systems."*Computer*. September 1991.

[4] Metcalfe, Robert M and Boggs, David. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*. July 1976.

[5] Nanya, Takashi and Goosen, Hendrik. "The Byzantine Hardware Fault Model." IEEE Transactions on Computer-Aided Design. vol. 8, November 1989. p. 1226-1231.

[6] Piuri, Vincenzo. "Design of Fault-Tolerant Distributed Control Systems." IEEE Transactions on Instrumentation and Measurement, vol. 43, April, 1994. p 257-264.

7. Acknowledgments

This thesis would not have been possible without the guidance and support of many people. I would like to take this time to thank some of the people and groups involved in helping me.

Prof. Richard D. Thornton, Professor of Electrical Engineering at MIT
Prof. Thornton was my advisor and mentor. I also had the privilege of serving as a teaching assistant for him in the Fall of 1996. I cannot thank him enough for giving me the opportunity to work on this project. It has been a very rewarding and enjoyable experience.

Dr. Brian Perreault, Doctoral Student at MIT
Brian gave me irreplaceable guidance and support. Brian, thank you for all the advice and resources you gave me. And thank you for putting up with me while trying to get your Ph.D. done at the same time. Good luck with MagneMotion.

Dr. Tracy Clark and the people at MagneMotion
Tracy and the rest of the people at MagneMotion Inc. gave me the financial resources to manufacture the PCB boards in conjunction with helping me generate the goals for this project. I wish MagneMotion the best of luck. And Tracy, thank you for taking the time to read and correct my thesis.

Joel L. Dawson, Doctoral Student at Stanford University
Joel, you have been a great friend through out this entire process. I appreciated your many late night visits in lab when I was tired and needed a break. I also appreciated your fresh perspective. Thanks for allowing me to bounce ideas off you. And thanks for the advice on the analog circuitry as well. Good luck at Stanford!

Mom
Thank you for letting me call you late at night in lab when I needed someone to talk to. You do not know how much your confidence and faith helped me get through this process. Thank you.

People at the Laboratory for Electromagnetic and Electronic Systems
Working at LEES has been a wonderful experience. I never thought working on hardware could be so much fun. I have really enjoyed getting to know some of the people in lab.

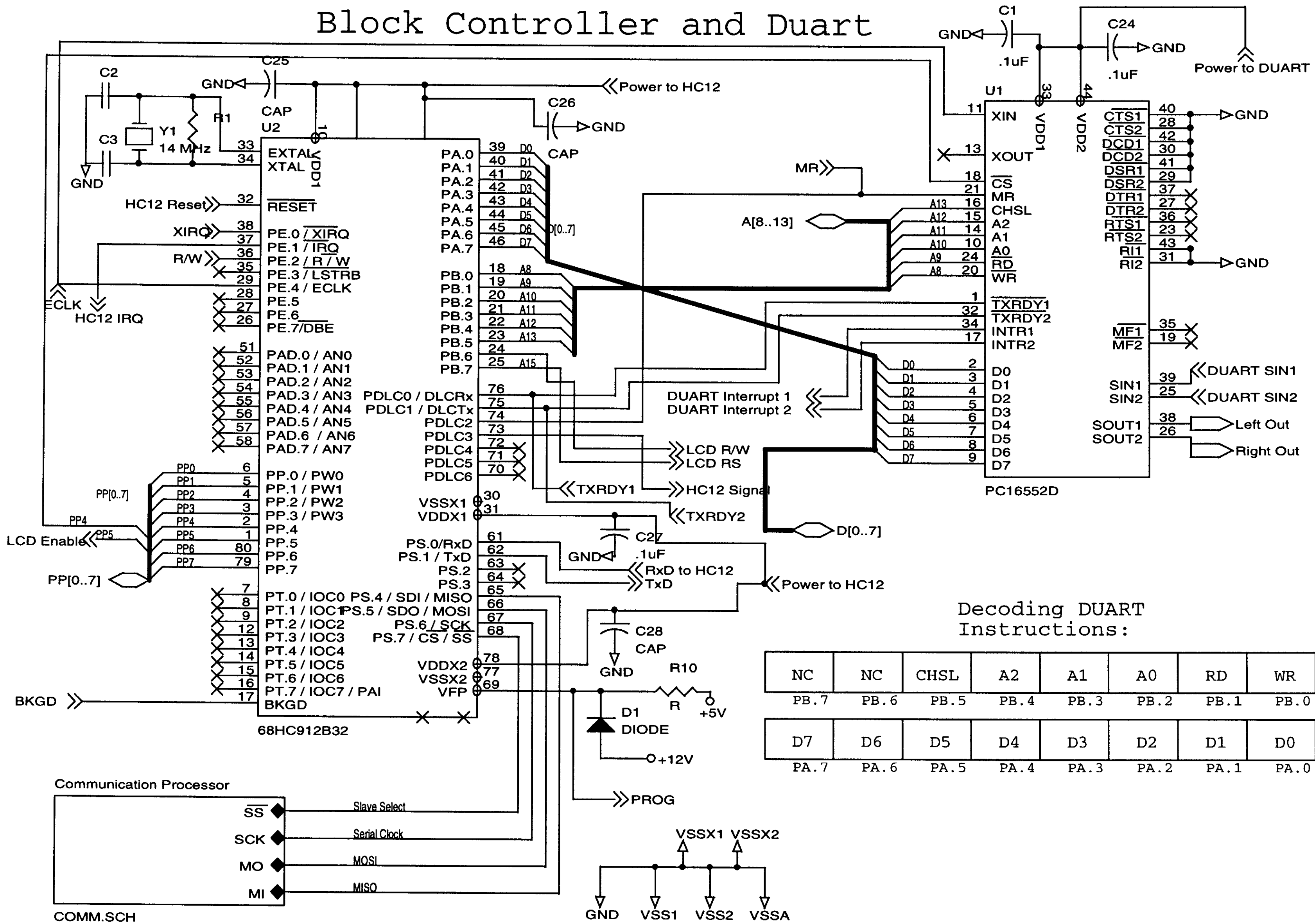
P&E MicroComputer Systems Inc.
P&E Microcomputer systems was kind enough to let me use an Alpha software release of their HC12 development system. The code developed for this thesis would not have been possible without their support.

Motorola University Support
The HC12 is a new microcontroller which has just recently become widely available to the public. Motorola University Support was kind enough to donate enough samples to me for constructing the prototype.

8. Appendix A: Prototype Schematics

A brief description of the hardware used for the prototype block and zone controllers is described in Chapter 3. This appendix contains a complete listing of the schematics used to construct the prototypes.

Block Controller and Duart

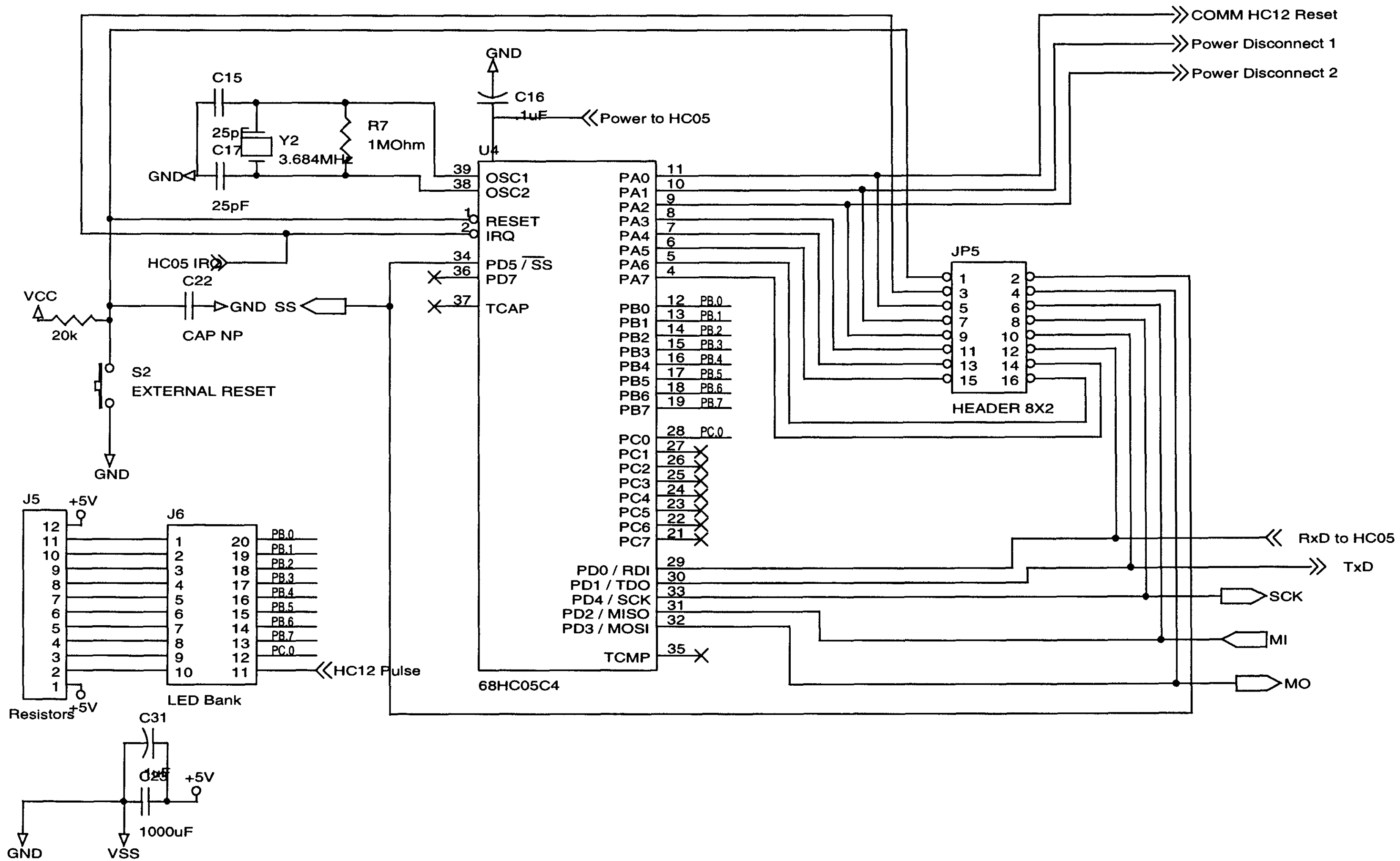


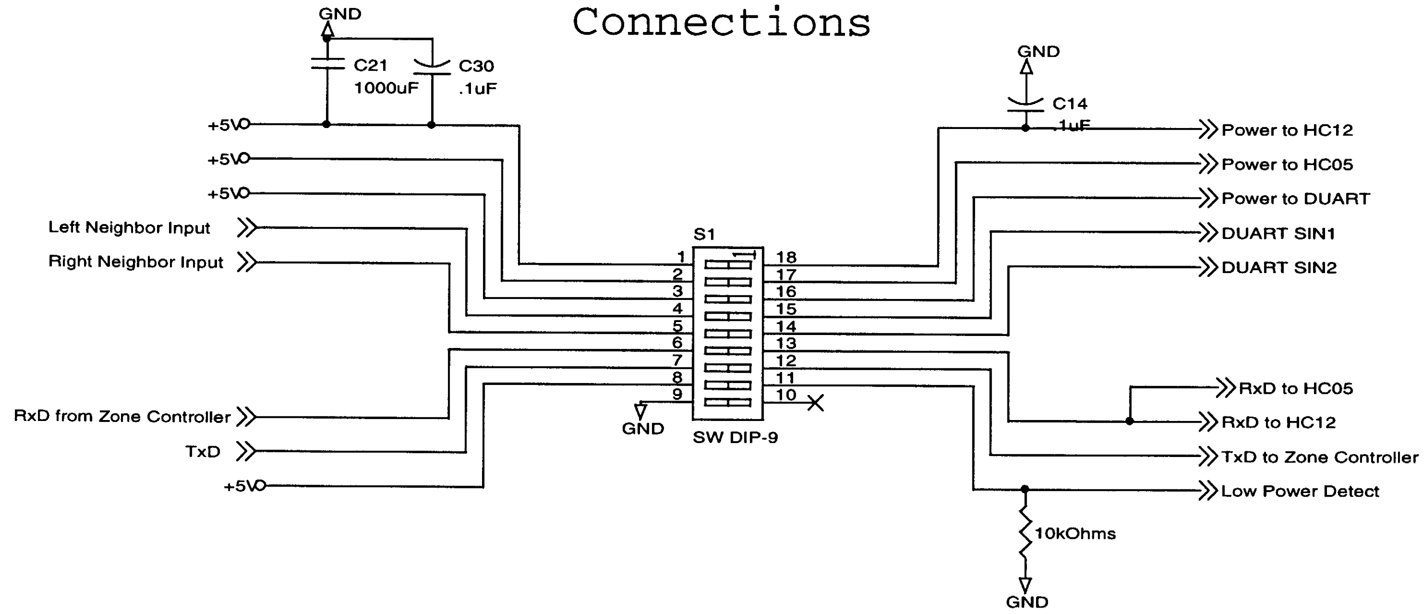
Decoding DUART Instructions:

NC	NC	CHSL	A2	A1	A0	RD	WR
PB.7	PB.6	PB.5	PB.4	PB.3	PB.2	PB.1	PB.0

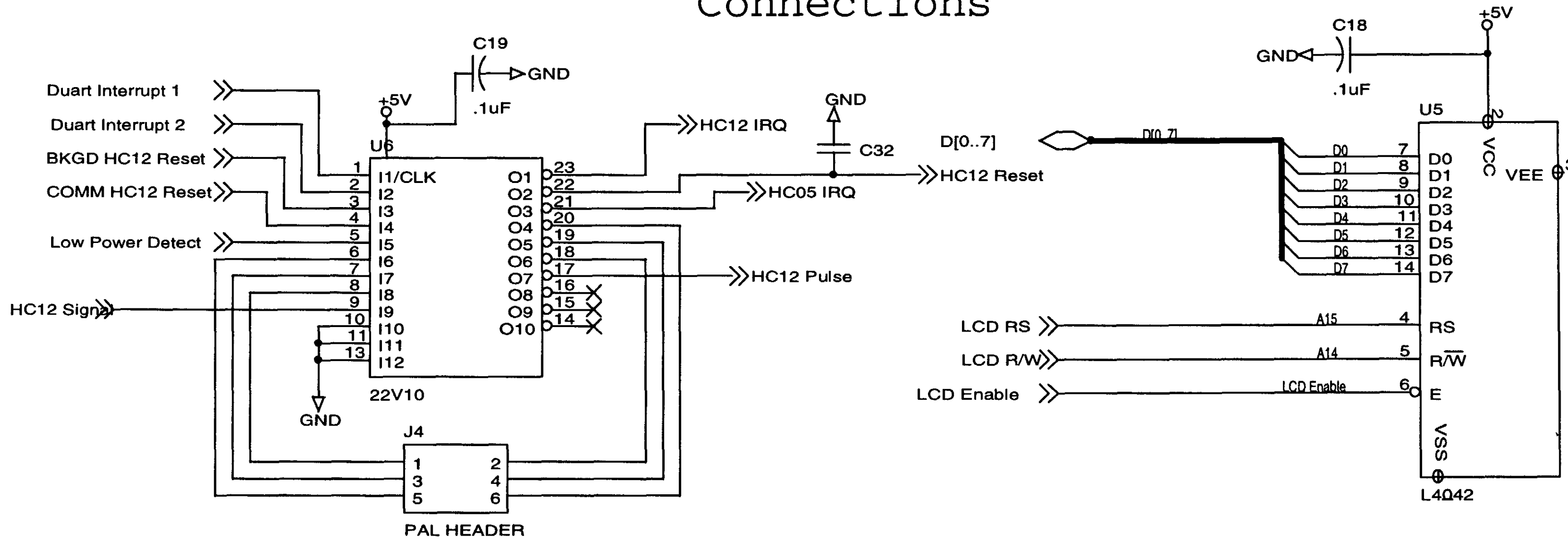
D7	D6	D5	D4	D3	D2	D1	D0
PA.7	PA.6	PA.5	PA.4	PA.3	PA.2	PA.1	PA.0

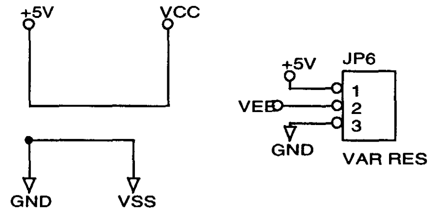
Communication Processor for Block Controller




DIP Switch
Connections
GND
C21
1000uF
C30
.1uF
+5V
+5V
+5V
Left Neighbor Input
Right Neighbor Input
RxD from Zone Controller
TxD
+5V
S1
1
2
3
4
5
6
7
8
9
GND
SW DIP-9
18
17
16
15
14
13
12
11
10
GND
C14
.1uF
Power to HC12
Power to HC05
Power to DUART
DUART SIN1
DUART SIN2
RxD to HC05
RxD to HC12
TxD to Zone Controller
Low Power Detect
10kOhms
GND

PAL and LCD Connections





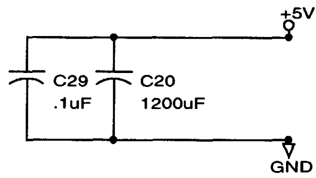
Decoding LCD Instructions:

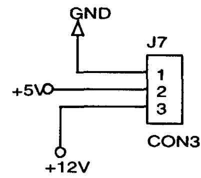
RS	R/W	NC	NC	NC	NC	NC	NC
PB.7	PB.6	PB.5	PB.4	PB.3	PB.2	PB.1	PB.0

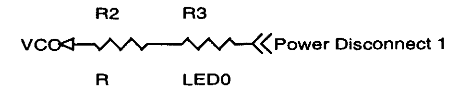
D7	D6	D5	D4	D3	D2	D1	D0
PA.7	PA.6	PA.5	PA.4	PA.3	PA.2	PA.1	PA.0

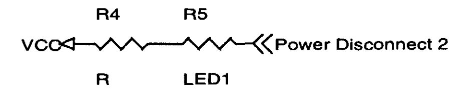
Communication Connections

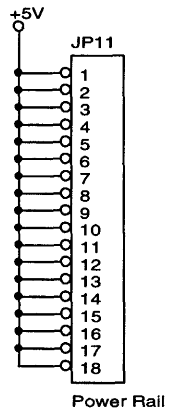
Power Electronics

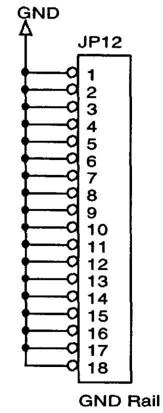




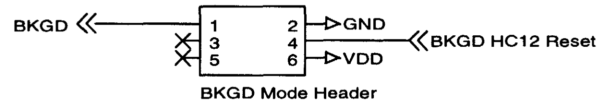


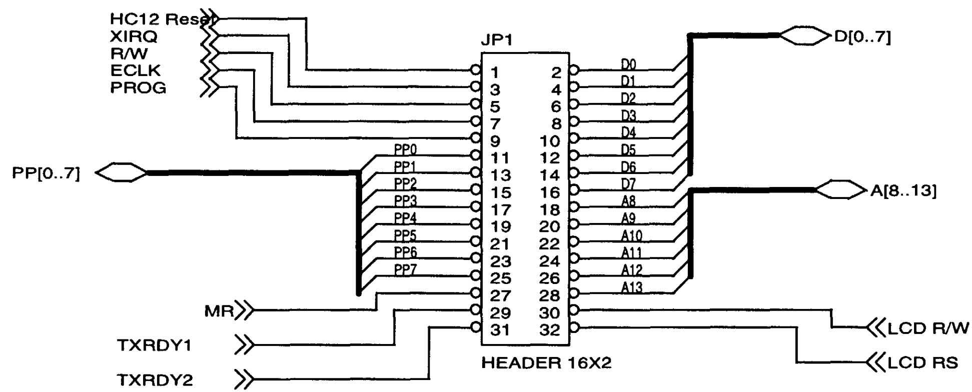






Header Connections





9. Appendix B: Code Listing

A brief description of the code written to implement the block and zone controllers is included in the implementation discussion in Chapter 3. All code is written for the HC12 microcontroller. This appendix contains a full, commented listing of the following code segments:

Constants and Variables:
HC12Equ.H, Messages.H, EventHan.H, Vehicles.H, Ident.H, Virtual.H, Vars12.H

Inits and Main Loop:
Shell12.ASM, Init.ASM

Communication Backbone
MessHand.ASM, Router.ASM, SCI12.ASM, UART1.ASM, UART2.ASM

Protocol Tools
EventHan.ASM, Vehicle.ASM

Protocols
Ping.ASM, Virtual.ASM, VehExch.ASM

Miscellaneous
Display.ASM, CheckSum.ASM, LCD.ASM, Banner.ASM, Misc.ASM

Constants and Variables:

```
;-------------------------------------------------------------------
; HC12EQU.h
; Constant Equates Used by the modules.
;-------------------------------------------------------------------

MESSAGE_SIZE     EQU $16          ; including checksum 22 bytes
ChkSumSize       EQU $02
QueueSize        EQU 8*$14        ; holds 8 messages

PingDelay        EQU  $7D0        ; ping every .5seconds (1000 * 500useconds)


; Timer increments every 135nsec. Value of $FA0 implies that event
; handler is called every 500 useconds

EHDelayInterval EQU  $FA0

; Routing Table Constants
DestField        EQU  $00
PortField        EQU  $01
EntrySize        EQU  $02    ; every 2 bytes

$IF ZONE_CONTROLLER
DefaultTableSize EQU   $06
$ELSEIF
DefaultTableSize EQU   $04
$ENDIF

ZoneController      EQU  $00
CentralController   EQU  $05
LeftNeighbor        EQU  $01
LeftBlock           EQU  $02
RightNeighbor       EQU  $02
RightBlock          EQU  $01
CoProcessor         EQU  $03    ; Either Main Processor or Auxiliary Proc.
RouteTableEntries   EQU  $06
RouteTableSize      EQU  $0C    ; # bytes in the routing table (6 entries)

TrackChar           EQU  $02
VehicleChar         EQU  $00

; Routing Table --> Entry constants (port handlers)
SCIPort          EQU  $00
SPIPort          EQU  $01
DUART1           EQU  $02    ; only valid for Main Processor
DUART2           EQU  $03    ; only valid for Main Processor

RAM              EQU $0800   ; first RAM address
ENDRAM           EQU $0BFF
EEPROM           EQU $0D00   ; first EEPROM address (ends at $0FFF)
FLASH            EQU $8000

HeaderByte       EQU $3F     ; first byte in a message

; LCD Buffer Information
LCDBufferSize  EQU  $40     ; 64 character buffer
ClearDispCode  EQU  $FF     ; code used in buffer entry to clear display
ChgAddrCode    EQU  $FE     ; used in buffer entry to signify change addr

; DUART2 Port Handler
UA2MessageSize  EQU  2*Message_size+1
UA2RMBSize      EQU  2*Message_size     ; RMB can hold 2 messages
UA2TMBSize      EQU  $5A                ; TMB holds 2 messages

; DUART1 Port Handler
UA1MessageSize  EQU 2*Message_size+1
UA1RMBSize      EQU 2*Message_size
;UA1TMBSize     EQU *(2*Message_Size+1)
UA1TMBSize      EQU $5A

;SCI Port Handler Related
SCIMessageSize  EQU 2*MESSAGE_SIZE+1
SCITMBSize      EQU $5A
SCIRMBSize      EQU 2*MESSAGE_SIZE

;SPI Port Handler Related Equates
SPIMessageSize  EQU    2*MESSAGE_SIZE+1    ; convert 2 ASCII + 1 header byte
SPITMBSize      EQU    MESSAGE_SIZE
SPIRMBSize      EQU    2*MESSAGE_SIZE
```

```
; Banner Constants
WindowSize      EQU   $14
BannerBufSize   EQU   $28                 ; 40 character buffer
BannerInterval  EQU   $320             ; .1 seconds (200 * 500usec)

;---------------------------------------------------------------------
; Messages.h
; Block / Zone Controller --> HC12 --> Message Code listing
;
; Gives the message codes used to identify message types.
;---------------------------------------------------------------------

; Current supported message types
Print_Data      EQU    $01
Ping_Request    EQU    $02
Ping_Response   EQU    $03

VirtualVehGen   EQU    $04
VVACKMess       EQU    $05

VehicleExchange EQU    $06
VehicleExchACK  EQU    $07
VehicleExchBLK  EQU    $08
VehicleExchCont EQU    $09


; Current offsets from start of the message to fields
MessCode        EQU     $00
MessDest        EQU     $01
MessSrc         EQU     $02
MessSize        EQU     $03
MessData        EQU     $04        ; message has 16 bytes of data
MessChkSum      EQU     $14        ; so checksum falls as last 2 bytes

; Timeout Delay Values for protocols
VVMTimeCnt      EQU       $1388     ; .25s timeout period
VehExchCnt      EQU       $1388     ; .25s

;---------------------------------------------------------------------
; EventHan.h
; Block & Zone Controller --> HC12 --> Event Handler Definitions
;
; Constants used by the event handler.
;---------------------------------------------------------------------

; Each event record is five bytes. The first two bytes specify the
; event time. The next two bytes specify the address of the byte flag
; associated with the particular event. The 5th byte is a flag mask. The
; mask is used to determine which bits in the flag byte should be
; set when the event time has expired.

EventRecSize      EQU      $05     ; each event record is five bytes
EHFlagMask        EQU      $04
EHTime            EQU      $02
EHFlagAddr        EQU      $00
NumEvents         EQU      $0A     ; number of recs EH can hold
EHBufferSize      EQU      EventRecSize*$0A
EHBufferFull      EQU      $01     ; error code for buffer full

;---------------------------------------------------------------------
; Vehicle.h
; Block Controller --> HC12 --> Vehicle Handler --> Definitions file
;
;---------------------------------------------------------------------

; Each vehicle record is 5 bytes long. The first byte is the VID,
; the 2nd byte is VPOS and the last two bytes are VVEL. The last byte is
; a flag reserved for use by the vehicle manager and is set when a
; vehicle is blocked from moving forward.

; Error message returned by AddVehicle:
PositionOccupied        EQU       $00
VehicleAdded            EQU       $02
TrackFull               EQU       $03
Blocked                 EQU       $01
Free                    EQU       $00

VehicleRecSize          EQU       $05       ; each record is 4 bytes
VID                     EQU       $00
VPOS                    EQU       $01
VVEL                    EQU       $02
```

```
VBLOCKED                EQU      $04

LastPosition            EQU      $80
NumPositions            EQU      $80   ; when pos. == $80, in last slot
MaxNumVehicles          EQU      $08   ; at most, 8 vehicles in a block
;VehicleBufferSize      EQU      MaxNumVehicles*VehicleRecSize
VehicleBufferSize       EQU      $28
VehicleVel              EQU      $1388 ; corresponds to a .25sec


; Character codes for printing vehicles and track
VehicleBack           EQU        $00
VehicleFront          EQU        $01
TrackBack             EQU        $02
TrackFront            EQU        $03

;-------------------------------------------------------------------
; Ident.h
; Destination and Source Identification Numbers used by the system.
; Values depend on whether the code is compiled for LB, RB or ZC
;-------------------------------------------------------------------

$IF  RIGHT_BLOCK
MYIDRN  EQU    $04    ; i am 04 to my right neighbor
MYIDLN  EQU    $05    ; i am 05 to my left neighbor
ZCID    EQU    $01
LNID    EQU    $02
RNID    EQU    $03
$ENDIF

$IF   LEFT_BLOCK
MYIDLN  EQU   $03
MYIDRN EQU    $02
ZCID   EQU    $01
LNID   EQU    $04
RNID   EQU    $05
$ENDIF

$IF  ZONE_CONTROLLER
MYIDLN  EQU  $01
MYIDRN  EQU  $01

; I NEED TO KNOW THE ALIASES USED BY RB AND LB TO PROPERLY FORWARD MESSAGES
LB_ID1  EQU  $02
LB_ID2  EQU  $03
RB_ID1  EQU  $04
RB_ID2  EQU  $05
LNID    EQU  $02
RNID    EQU  $04
$ENDIF

;-------------------------------------------------------------------
; Vars12.h
;
; Block /ZoneController --> HC12 --> Variable definition file
;
; Variable definions for the HC12 shell code and related routines
;-------------------------------------------------------------------

       ORG RAM
; LCDBuffer Related Variables
LCDBuffer     DS      LCDBufferSize
LCDHead       DS      2
LCDTail       DS      2
VVMFlag       DS      1
ALIVE         DS      1
ALIVEOld      DS      1
MessBuffer    DS     Message_Size
TimerFlag     DS     1
TimerFlag2    DS     1


; Vehicle Handler related variables
PositionStatus DS     1
VehicleBuffer  DS     VehicleBufferSize
VehicleCnt     DS     1               ; # vehicles in the block
VehicleStatus  DS     1
VehicleRec     DS     VehicleRecSize

; Variables used by Vehicle Exchange Protocol
VehExchFlag    DS     1           ; bit 0 used for vehexch timeout
```

```
ExchVID         DS     1
ExchVVEL        DS     2
PendingVID      DS     1

; Event Handler related variables
EventHanCnt     DS     1                ; # pending events in the EH buffer
EHRecord        DS     EventRecSize
EHBuffer        DS     EHBufferSize     ; the buffer for pending events

; Routing Variables
RouteTable      DS      RouteTableSize
RouteMessPort   DS      1

; Message Queue Related Variables
QUEUE           DS      QueueSize
QueueHead       DS      2
QueueTail       DS      2
QSize           DS      1

ChkSumHi        DS      1
ChkSumLo        DS      1
ChkSumFailures DS     1

; UART1 Related Variables
UA1TMB        DS       UA1TMBSize
UA1RMB        DS       UA1RMBSize
RxUA1Head     DS       2
RxUA1Tail     DS       2
RxUA1Start    DS       2
TxUA1Head     DS       2
TxUA1Tail     DS       2
RxUA1HiLo     DS       1
RxUA1Cnt      DS       1
TxUA1Cnt      DS       2

; UART2 Related Variables
UA2TMB       DS        UA1TMBSize
UA2RMB       DS        UA1RMBSize
RxUA2Head    DS        2
RxUA2Tail    DS        2
RxUA2Start   DS        2
TxUA2Head    DS        2
TxUA2Tail    DS        2
RxUA2HiLo    DS        1
RxUA2Cnt     DS        1
TxUA2Cnt     DS        2

; Banner Related Variables
BannerFlag    DS        1
BannerBuffer DS         BannerBufSize
WindowHead    DS        2
BannerTail    DS        2
BannerChars   DS        1

; SCI Port Related Variables
SCITMB        DS        SCITMBSize    ; buffer for transmitting messages
SCIRMB        DS        SCIRMBSize    ; buffer for received messages
TxSCIHead     DS        2
TxSCITail     DS        2
TxSCICnt      DS        2
RxSCIHead     DS        2
RxSCIStart    DS        2      ; first addr. of the current message
RxSCITail     DS        2
RxSCICnt      DS        1
RxSCIHiLo     DS        1    ; 1 if high nibble received, 0 if low nibble
```

Inits and Main Loop:

```
;---------------------------------------------------------------------------
; Shell12.ASM
; Block / Zone Controller --> HC12 --> Shell
;
; Shell for the HC12 code.
;---------------------------------------------------------------------------

$SET  INCLUDE_FLASH

;$SET  ZONE_CONTROLLER
$SETNOT ZONE_CONTROLLER

;$SET    RIGHT_BLOCK
$SETNOT RIGHT_BLOCK

$SET     LEFT_BLOCK
;$SETNOT   LEFT_BLOCK

$include "hc12\HC12EQU.h"
SELFID  EQU   ZoneController
;SELFID   EQU  LeftBlock
;SELFID   EQU RightBlock
$include "hc12\ident.h"
$include "hc12\hc12reg.h"
$include "hc12\eventhand.h"
$include "hc12\vehicles.h"

; now include variable files
$include "hc12\vars12.h"

         org  FLASH
$include "hc12\init.asm"
$include "hc12\lcd.asm"
$include "hc12\misc.asm"
$include "hc12\checksum.asm"
$include "hc12\sci12.asm"
$include "hc12\router.asm"
$include "hc12\messhand.asm"
$include "hc12\display.asm"
$include "hc12\actions.asm"
$include "hc12\ping.asm"
$include "hc12\uart1.asm"
$include "hc12\uart2.asm"
$include "hc12\eventhan.asm"
;$include "hc12\spi12.asm"
$include  "hc12\virtual.asm"
$include "hc12\banner.asm"
$include "hc12\vehicle.asm"
$include "hc12\vehexch.asm"
         org EEPROM

INIT
   SEI      ; disable interrupts
   ; initialize the stack pointer
   LDS      #ENDRAM      ; stack grows down from end of RAM
   JSR      InitFLASH      ; common routines initiaized in the FLASH now.

   CLI

MAIN
   JSR       InitPingProtocol
   JSR       InitBanner
   JSR       InitVehExch
   JSR       Delay5ms
   JSR       DisplayTrack
   JSR       GenPingReq

$IF  ZONE_CONTROLLER
  LDY     #VehicleRec
  LDD     #$1200
  STD     VVEL,Y
  LDAA    #$02
  STAA    VID,Y
  LDAA    #$01
  STAA    VPOS,Y
  LDAA    #LB_ID1
  JSR      GenVVMess
$ENDIF
```

```
  LBRA    WaitLoop     ; iterate through the message loops

;------------------------------------------------------------------------------
; Interrupt Handlers
;------------------------------------------------------------------------------

SCIIntHandler

  LDAA SC0SR1
  BRSET SC0SR1,$20,SCIIntHandler_Rx
  JSR   TxNextSCI
  RTI

SCIIntHandler_Rx
  JSR  RxMessageSCI      ; otherwise, it was a receive interrupt
  RTI

SPIIntHandler
;    JSR  RxMessageSPI
    RTI

TC5IntHandler
   JSR  HandleEHInt
   RTI

TC6IntHandler    ; compare module used by the Ping Protocol
;  JSR HandlePingInt
  RTI

TC7IntHandler    ; compare/capture module 7 interrupt
  JSR HandleLCD
  RTI

ResetHandler
  LBRA  INIT

;------------------------------------------------------------------------------
; Active Interrupt and Reset Vectors (in EEPROM space)
;------------------------------------------------------------------------------
   org  RESET
   LBRA  ResetHandler

   ORG   IRQ_INT
   LBRA  HandleDUARTInt

   ORG  SPI_INT
   LBRA  SPIIntHandler

   ORG  SCI_INT
   LBRA  SCIIntHandler

   ORG  TC5_INT
   LBRA TC5IntHandler

   ORG  TC6_INT
   LBRA  TC6IntHandler

   ORG  TC7_INT
   LBRA  TC7IntHandler

;------------------------------------------------------------------------------
; Interrupt and Reset Vector table (in FLASH PROM SPACE)
;------------------------------------------------------------------------------
$IF INCLUDE_FLASH
   org _RESET
   FDB RESET

   ORG _IRQ_INT
   FDB IRQ_INT

   ORG  _SPI_INT
   FDB  SPI_INT

   ORG  _SCI_INT
   FDB  SCI_INT

   ORG  _TC5_INT
   FDB  TC5_INT

   ORG  _TC6_INT
```

```
   FDB  TC6_INT

   ORG  _TC7_INT
   FDB  TC7_INT
$ENDIF

;-----------------------------------------------------------------------
; Init.ASM
; Block / Zone Controller → HC12 → Calls to Init routines + main loop
; Basic Initalization routines which are stored on FLASH.
;-----------------------------------------------------------------------

$IFNOT INCLUDE_FLASH
InitFlash
InitContinue
VersionString
WaitLoop
$ELSEIF

PriorityLoop    ;call this routine to perform a check of the port handlers
WaitMess0
  LDAA     RxUA2Cnt
  BEQ      WaitMess1
  JSR      AddUA2Mess
WaitMess1
  LDAA     RxUA1Cnt
  BEQ      WaitMess2
  JSR      AddUA1Mess
WaitMess2
  LDAA     RxSCICnt
  BEQ      WaitMess3
  JSR      AddSCIMess
WaitMess3
  RTS

WaitLoop
  JSR     PriorityLoop
  LDAA    QSize
  BEQ     WaitMess5
  JSR     HandleMessage

WaitMess5
 BRCLR     ALIVE,%10000000,WaitMessA
 JSR       HanPingTimeout
 JSR       PriorityLoop
 JSR       GenPingReq
 JSR       PriorityLoop

WaitMessA
  BRCLR    VVMFlag,%01000000,WaitMessB    ; vvm ping pending
  BRCLR    VVMFlag,%10000000,WaitMessB    ; virtual vehicle timeout?
  JSR      HanVVTimeout
  JSR      PriorityLoop

WaitMessB
  LDAA     VehicleStatus
  BEQ      WaitMessD                     ; vehicle manager update?
  JSR       UpdateVehicles
  JSR       PriorityLoop

WaitMessD
   BRCLR     VehExchFlag,$80,WaitMessE
   JSR       HanVehExchTO    ; vehicle exchange timeout
   JSR       PriorityLoop
WaitMessE
   BRCLR     VehExchFlag,$40,WaitMessC
   JSR       HanVehExchContTO
WaitMessC
  BRCLR      BannerFlag,%00000001,WaitMess6
  JSR        UpdateBanner
WaitMess6
  BRA        WaitLoop


InitFlash
  MOVB    #$FF,DDRA
  MOVB    #$FF,DDRB
  MOVB    #$FF,DDRP
  MOVB    #$00,DLCSCR
  MOVB    #%01111100,DDRDLC
  MOVB    #$FF,TimerFlag
```

```
        MOVB    #%00010000,PORTP

        JSR     InitLCD
        JSR     InitSCIHandler
        JSR     InitRouter
        JSR     InitMessHandler
        JSR     InitUA1Handler
        JSR     InitUA2Handler
        JSR     InitEventHandler
        JSR     InitVehicleList

        RTS

$ENDIF
```

Communication Backbone:

```
;---------------------------------------------------------------------------
; MessHand.ASM
; Block / Zone Controller --> HC12 --> Message Handler
;
; The message handler is in charge of the message queue. The message type
; of messages in the queue are examined here and sent to the appropriate
; action routine based on the message type.
;
; Public Methods: InitMessHandler
; Private Methods:
;---------------------------------------------------------------------------

$include "hc12\messages.h"

$IFNOT INCLUDE_FLASH
InitMessHandler
RemoveMessage
HandleMessage
AddMessage
$ELSEIF
;---------------------------------------------------------------------------
; InitMessHandler: Intializes the message handler including the message
;                  queues. Assumes first byte in queue space is addr. QUEUE
;                  Also uses constant: Queue size
;
; Modifies:        ACCA, X, Head, Tail
;---------------------------------------------------------------------------
InitMessHandler:
    ; empty out the queue
    LDX  #Queue
    LDY  #QueueSize
    JSR  ClearBuffer

    CLR  ChkSumFailures
    LDX  #Queue
    STX  QueueHead      ; initialize head and tail addresses to top of queue
    STX  QueueTail
    CLR  QSize          ; variable for # messages in the queue
    RTS
;---------------------End of InitMessHandler-----------------------------

;-----------------------------------------------------------------------
; AddMessage - Adds a message to the queue. If the queue is full, calls
;              HandleMessage then adds the message to the Qeueue.
;              Typically called by a Port Handler when it has received
;              an entire message
;
;              AddMessage now performs a checksum check on each message
;              being added to the Queue. If the checksums do NOT match,
;              the message will not be added to the queue and the bad
;              message count will be updated.
;
; Arguments:    X contains address of first byte.
; Assumes:      Messages are a fixed size (Message Size). Always copies
;               MESSAGE_SIZE bytes.
; Modifies:     ACCA,X,QUEUE,TAIL,AddMessage_CNT,AddMessage_SRC
;-----------------------------------------------------------------------
AddMessage
  ; First, compute a checksum on the message to be added and verify
  ; result. Assumes the last 2 bytes in the message hold the original
  ; checksum.
  TFR  X,Y      ; copy source in Y
  LDAA  #Message_Size-$02
;  LDAA  #MessageSize-#ChkSumSize
  ; X already has the address of the first byte
  JSR  CompChkSum


  ; Y has index into current source
  TFR Y,X      ; now X has address of first message byte
  LDAA  MessChkSum,X
  CMPA  ChkSumHi            ; check the high byte of the checksum
  BNE   ChkSumFailure
  INX
  LDAA  MessChkSum,X
  CMPA  ChkSumLo            ; check the low byte of the checksum
  BNE   ChkSumFailure       ; we failed a checksum
  BRA   AddMessage_0         ; checksums passed so continue processing

ChkSumFailure
```

```
  INC  ChkSumFailures
  JSR  UpdateChkFail
  RTS  ; return, do not add the message to the queue

; Y still holds address of first source byte
AddMessage_0
; Assumes Tail is currently pointing to the next available free slot
;  LDAB  #Message_Size-#ChkSumSize       ; we have copied 0 bytes so far
  LDAB  #Message_Size-2

AddMessage_1
  LDAA  ,Y          ; load a byte
  LDX  QueueTail
  STAA  ,X          ; store the byte in the queue
  INX
;  CPX #QUEUE+#QUEUE_SIZE
  CPX  #QUEUE+$A0
  BNE AddMessage_2
  LDX #QUEUE             ; wrap the tail around to the front of the queue

AddMessage_2
  STX QueueTail              ; store the new tail
  INY                        ; increment source
  DECB                       ; dec # bytes copied so far
  BNE AddMessage_1           ; do we have more bytes to copy?
  INC QSize
  RTS
;------------------------------End of AddMessage------------------------

;-----------------------------------------------------------------------
; HandleMessage: Assumes there is at least one message in the queue.
;               Examines the message type and calls an appropriate action
;               routine.
;
; Modifies:  ACCA, X, Head
;
; Additions:    If the message Code is not a Ping request or Ping ID,
;               the destination field is examined. If it does not match
;               the current IDs of the controllers, the message is
;               forwarded to the appropriate destination. Forwarding
;               occcurs by calling the RouteMessage routine with the
;               message destination ID and the address of the Queue head
;               as arguments.
;
; Currently supports the following routines: PingReq, PingRes,VirtualVehGen,
;            VVACKMess, VehicleExchange,VehicleExchangeBlk,VehicleExchangeCont,
;            and VehicleExchACK.
;-----------------------------------------------------------------------
HandleMessage
  LDX  QueueHead         ; X points to the first byte in the message
  LDAA  MessCode,X       ; load the message code   (one byte)

HandleMessage_0
  CMPA #Ping_Request
  BNE HandleMessage_1
  JSR  HandlePingReq
  BRA  HanMessDone

HandleMessage_1
  CMPA  #Ping_Response
  BNE  HandleMessage_forw
  JSR  HandlePingRes
  BRA  HanMessDone

; We must now determine if the message needs forwarding.
HandleMessage_forw
  LDAA  MessDest,X      ; load the destination ID
  CMPA  #MYIDLN         ; compare to the ID I use with my left neighbor
  BEQ   HandleMessage_2 ; message for us?
  CMPA  #MYIDRN         ; compare to the ID I use with my right neighbor
  BEQ   HandleMessage_2 ; message for us?

  ; if we got this far, the message is not for us. We should
  ; forward the message to the appropriate destination
  TFR  A,X             ; load destination ID
  LDY  QueueHead       ; make sure Y points to the start of the message
  JSR  RouteMessage    ; Route the message
  BRA  HanMessDone

; if the message is intended for us, we can now handle it
HandleMessage_2
```

```
  LDAA  MessCode,X
  CMPA  #VirtualVehGen
  BNE   HandleMessage_3
  JSR   HandleVVGen
  BRA   HanMessDone

HandleMessage_3
   CMPA  #VVACKMess
   BNE   HandleMessage_4
   JSR   HandleVVAck
   BRA   HanMessDone

HandleMessage_4
   CMPA  #VehicleExchange
   BNE   HandleMessage_5
   JSR   HanVehExch
   BRA   HanMessDone

HandleMessage_5
   CMPA  #VehicleExchACK
   BNE   HandleMessage_6
   JSR   HanVehExchACK
   BRA   HanMessDone

HandleMessage_6
   CMPA  #VehicleExchBLK
   BNE   HandleMessage_7
   JSR   HanVehExchBLK
   BRA   HanMessDone

HandleMessage_7
 CMPA  #VehicleExchCont
 BNE   HandleMessage_8
 JSR   HanVehExchCont
 BRA   HanMessDone
HandleMessage_8
HanMessDone
    JSR RemoveMessage
HanMessDone_0
   RTS

;----------------------------------------------------------------------
; RemoveMessage - Advances the Head of the Queue. Typically called after
;                 a message has been acted upon.
; Modifies:       HEAD,ACCA,X
;----------------------------------------------------------------------
RemoveMessage
  ; Assumes QUEUESIZE is always a multiple of the message size
  LDD   QueueHead
  ADDD  #MESSAGE_SIZE-$02
;  ADDD  #MESSAGE_SIZE-ChkSumSize
  TFR   D,X
  CPD   #QUEUE+$A0
;  CPD    (#QUEUE+#QueueSize)
  BNE  RemoveMessage_0
  LDX  #QUEUE            ; wrap the head of the queue around to the front

RemoveMessage_0
  STX  QueueHead
  DEC  Qsize
  RTS
;----------------------End of RemoveMessage----------------------------
$ENDIF

;-------------------------------------------------------------------------
; Router.ASM
;
; Block / Zone Controller --> HC12 --> Router
;
; The Router is responsible for keeping track of the relationships between
; ports and destinations. It keeps this information in a routing table and
; provides tools to update this routing table if necessary.
; The router also takes a message and a destination code and routes it to
; the appropriate PortHandler to be transmitted.
;-------------------------------------------------------------------------

$IFNOT INCLUDE_FLASH
InitRouter
RouteMessage
UpdateEntry
RestoreEntry
```

```
$ELSEIF

;-----------------------------------------------------------------------
; RestoreEntry:   A holds an identifier in the table. This routine
;                 restore the default port associated with this destination.
;                 This routine should be called after a the ping protocol
;                 has detected that a link has been restored.
; Arguments:      A holds the ID of the destination. The destination is
;                 in the routing table.
; Modifies:       RouteTable
;-----------------------------------------------------------------------
RestoreEntry
  LDX  #RouteTable
  LDAB #EntrySize
  LDY  #$00

  ; make a linear search of the routing table, searching for the entry
RestoreEntry_next
  CMPA  DestField,X
  BEQ    RestoreEntry_match
  INY                  ; advance index into default table
  ABX
  CPY  #DefaultTableSize
  BNE  RestoreEntry_next
  BRA  RestoreEntry_fin   ; Destination not in table!

RestoreEntry_match     ; we found the match
  LDAB  DefaultTable,Y ; load default value
  STAB  PortField,X    ; update the entry with the default value
RestoreEntry_fin
  RTS                  ; we are done

;-----------------------------------------------------------------------
; UpdateEntry:  Given ID1 in B and ID2 in A, changes the destination port
;               of the entry for ID1 to that of ID2. Call this routine
;               when communication should be re-routed from one destination
;               to another. ID2 must be prepared to forward the message.
; Arguments:    B holds destination ID to be updated. A holds ID of destination
;               to use.
; Modifies:       RouteTable
;-----------------------------------------------------------------------
UpdateEntry
  ; First, scan through the list of entries, finding ID2 (in A) and
  ; then save its port.
  PSHB  ; Save ID2
  LDAB  #EntrySize
  LDY   #RouteTableEntries
  LDX   #RouteTable

UpdateEntry_0
  CMPA  DestField,X
  BEQ   UpdateEntry_match      ; we found the entry
  ABX
  DBNE  Y,UpdateEntry_0
  ; otherwise, we have reached end of list...destination not listed!!!
  LDX   #BadDestStr
  JSR   PrintStrToBanner
  RTS

UpdateEntry_match            ; we found the entry, get its destination port
  LDAA  PortField,X
  ; now we need to scan the table AGAIN, this time looking for ID1
  ; when we find it, we want to store the new PortField
  PULB          ; restore ID2
  EXG  A,B
  PSHB          ; save port field

  LDY   #RouteTableEntries
  LDX   #RouteTable
  LDAB  #EntrySize

UpdateEntry_next
  CMPA  DestField,X
  BEQ   UpdateEntry_2       ; we found the entry
  ABX
  DBNE  Y,UpdateEntry_next
  ; otherwise, we have reached end of list...destination not listed!!!
  LDX  #BadDestStr
  JSR  PrintStrToBanner
  RTS
```

```
UpdateEntry_2                  ; we found the entry to update
  PULB                            ; restore port field
  STAB  PortField,X               ; store the value
  RTS

;-----------------------------------------------------------------------
; InitRouter: Initialize the entries in the table. Default entries are
;             currently initialized at the code level and not via messages.
;
; Modifies:   RoutingTable, ACCA,X
;-----------------------------------------------------------------------
InitRouter

    ; router is a buffer of 12 characters. Each entry pair takes two
    ; bytes. the first byte is the Destination ID and the second byte
    ; is a constant the corresponds to the port to use.
    ; if the entry has a destination ID == 0, then it is empty.

   LDX  #RouteTable
   LDY  #RouteTableSize
   JSR  ClearBuffer

   LDY  #$00
   LDAB #EntrySize
   LDX  #RouteTable

$IF ZONE_CONTROLLER
   LDAA  #RB_Id1
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #RB_Id2
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #LB_Id1
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #LB_ID2
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #CoProcessor
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #CentralController
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField
$ELSEIF    ; then we are filling in a block controller (has 4 entries)
   LDAA  #ZCID
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #LNID
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY
```

```
   LDAA  #RNID
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
   ABX
   INY

   LDAA  #CoProcessor
   STAA  DestField,X
   LDAA  DefaultTable,Y
   STAA  PortField,X
$ENDIF     ; last 2 entries are left blank for block controllers

    RTS
;----------------------End of InitRouter---------------------------------


;-------------------------------------------------------------------------
; RouteMessage: Takes a destination code in X and the starting address
;               of the message in Y. Determines appropirate PortHandler
;               and initiates a transmission of the message with the
;               PortHandler.
;
;               RouteMessage also tacks a checksum on to the end of the
;               message. The last 2 bytes of the message buffer passed in
;               must be empty and available as storage for the checksum.
;
; Modifies:     ACCA,X,Y
;-------------------------------------------------------------------------
RouteMessage
  PSHY
  TFR   X,A             ; destination field is now in A.
  LDY   #RouteTableEntries
  LDAB  #EntrySize
  LDX   #RouteTable

RouteMessage_0
  CMPA  DestField,X
  BEQ   RouteMessage_1
  ABX
  DBNE  Y,RouteMessage_0
  ; otherwise, we have reached end of list...destination not listed!!!
  LDX   #BadDestStr
  JSR   PrintStrToBanner
  PULY
  RTS

RouteMessage_1       ; we found the entry
  LDAA  PortField,X
  STAA  RouteMessPort
  PULY
  JSR   SendMessage
  RTS   ; we are done

BadDestStr   FCB  'Bad Dest. ',0

;-------------------------------------------------------------------------
; SendMessage:    Takes the message pointed to by Y and the actual
;                 port in RouteMessPort variable. Appends a checksum
;                 and routes the message to appropriate port.]
; Arguments       RouteMessPort, Y has address of first message byte
;                 and the message buffer has room for the checksum at
;                 the end.
;-------------------------------------------------------------------------
SendMessage
  ; First, compute a checksum and append it to the end of the message
;  LDAA  #MESSAGE_SIZE-ChkSumSize    ; # bytes to be computed in ChkSum
  LDAA #MESSAGE_SIZE-2
  TFR  Y,X              ; prepare X argument
  JSR  CompChkSum       ; compute checksum

  TFR  Y,X             ; X now points to starting address of message
  LDAA  ChkSumHi       ; load high byte of calculated checksum
  STAA  MessChkSum,X   ; store high byte of the checksum

  INX
  LDAA  ChkSumLo
  STAA  MessChkSum,X             ; store low byte of the checksum

  ; Now route and transmit message at the appropriate port
  TFR   Y,X             ; load X with first address byte
```

```
  LDAA  RouteMessPort
  ; Determine which port will handle the message
  CMPA  #SCIPort        ; use SCI Port?
  BEQ   RtMessSCI
  CMPA  #DUART1         ; use DUART1?
  BEQ   RtMessDUART1
  CMPA  #DUART2         ; use DUART2?
  BEQ   RtMessDUART2
  CMPA  #SPIPort        ; use SPIPort
  BEQ   RtMessSPI
  BRA  RtMessSCI       ; default transmission port.

RtMessSCI                ; SCI Port Handler
    JSR   TxMessageSCI   ; transmit message
    RTS

RtMessDUART1
    JSR   TxMessageUA1
    RTS

RtMessDUART2
    JSR   TxMessageUA2
    RTS

RtMessSPI
;    JSR   TxMessageSPI   ; SPI Port Handler
    RTS

;-----------------------End of SendMessage--------------------------------

;-------------------------------------------------------------------------
; RouteToPort:    Takes a message address in Y and a destination in X.
;                 Bypasses current state of the routing table, using
;                 the original links between the elements. This routine
;                 will need to be changed if the network topology changes
;                 as well.
;-------------------------------------------------------------------------
RouteToPort
  LDAA DefaultTable,X        ; load desired destination port
  STAA RouteMessPort
  JSR  SendMessage
  RTS

$IF ZONE_CONTROLLER
; columns are:         RB1ID, RB2ID, LB1ID, LB2ID, AP, CC
DefaultTable     FCB   DUART2,DUART2,DUART1,DUART1,SPIPort,SCIPort

$ELSEIF
; columns are:         ZC ID,  LN ID, RN ID, AP
DefaultTable     FCB   SCIPort,DUART1,DUART2,SPIPort
$ENDIF

$ENDIF

;--------------------------------------------------------------------
; SCI12.ASM
; Block / Zone Controller --> HC12 --> SCI Port Handler Code
;
; Routines for the Serial Communications Interface (SCI) used to communicate
; with the zone controller.
;
; Public Routines:  InitSCIHandler, TXMessageSCI, RXMessageSCI, AddSCIMess
;                   TxNextSCI
;-----------------------------------------------------------------------

$IFNOT INCLUDE_FLASH
RxMessageSCI
TxMessageSCI
AddSCIMess
InitSCIHandler
TxNextSCI
$ELSEIF

;---------------------------------------------------------------------
; RxMessageSCI: Call after receiving a RDRF interrrupt on the SCI port.
;               Copies packet into a message buffer. If last packet in the
;               message, call PrintString. EVENTUALLY: place the completed
;               message in a message handler queue
; Application Note: This routine is called as an interrupt handler!
;                   It is based on the RDRF Flag
; 05/06/97      Assumes message is in ASCII format and has a header byte
```

```
;---------------------------------------------------------------------------
RxMessageSCI:
  LDAA  SC0SR1                  ; clear receive flag
  LDAA  SC0DRL                  ; load in the received byte
  LDX   RxSCITail             ; address of where to put the new packet

  ; Examine if received byte was a HeaderByte or not.
  CMPA  #HeaderByte
  BNE   RxMessSCI_0           ; not a header byte so process normally

  ; otherwise we have received a header byte in the middle of
  ; receiving another message. ADD ERROR HANDLER CODE HERE AT LATER DATE
  LDX   RxSCIStart            ; start at beginning of current message
  STX   RxSCITail             ; add new bytes at this location
  MOVB  #$FF,RxSCIHiLo        ; reset hi/lo nibble flag for next message

  ; since we do NOT want to actually store the header byte,
  ; we are done at this point as the system is ready to receive the new
  ; message.
  BRA  finish_sci

  ; Process the received byte
RxMessSCI_0
  ; Test RxSCIHiLo to determine if the received packet is a high or low
  ; nibble. Add the nibble appropriately to the Tail.
  BRCLR  RxSCIHiLo,$01,RxMessSCI_1    ; branch if it is the low nibble
  LSLA
  LSLA
  LSLA
  LSLA                  ; rotate the nibble into the high nibble
  STAA   ,X             ; store the high nibble in the buffer
  BCLR   RxSCIHiLo,%00000001   ; toggle flag for next byte
  BRA    finish_sci   ; we are done

RxMessSCI_1      ; we have just received a low nibble, incorporate it
  ANDA   #$0F     ; make sure the high nibble is zeroed out
  ORAA   ,X       ; combine high nibble in B with low nibble in A
  STAA   ,X       ; store the complete packet back into the buffer
  BSET   RxSCIHiLo,%00000001     ; toggle flag for next byte

  ; Now we need to load up and increment the Tail
  LDX    RxSCITail
  INX
  STX    RxSCITail             ; inc tail of the rmb
  LDD    RxSCITail
  SUBD   RxSCIStart
  CPD   #Message_Size        ; is it a complete message?
  BNE    finish_sci           ; if not, our job is done

  ; otherwise we need to make sure the Tail is not at the end of the
  ; SCIRMB (if so, we need to wrap it around) AND we need to alert
  ; the event driver that we have a message to add to the queue
  INC   RxSCICnt                ; we have received a complete message
  MOVB  #$FF,RxSCIHiLo          ; set Hi/lo flag for next incoming message

  LDX   RxSCITail
  STX   RxSCIStart                ; record new start addr of the next message
;  CPX   #SCIRMB+SCIRMB_SIZE
  CPX   #SCIRMB+$2C                ; end of buffer?
  BNE  finish_sci

  LDX   #SCIRMB
  STX   RxSCITail                ; wrap the tail around
  STX   RxSCIStart               ; wrap around start addr of next message

finish_sci
  RTS
;-------------------------End of RxMessageSCI----------------------------


;---------------------------------------------------------------------------
; InitSCI: Initialization routine for the SCI Port
;          Baud Rate = 19200bps. RIE, TE, RE interrupts are all turned on
;          Transmit and Receive lines are also enabled.
;          SCI Port packet length is one byte.
;---------------------------------------------------------------------------
InitSCI
       MOVB  #$00,SC0BDH
       MOVB  #$18,SC0BDL         ; sets baud rate of 19200bps
       MOVB  #%00000000,SC0CR1  ; 8 data bits, 1 start, 1 stop, NO PARTIY
       ; enable appropriate interupts (TIE,TCIE,RIE,TE,RE)
```

```
       MOVB  #%00101100,SC0CR2  ; right now, just turn on RIE,TE,RE
       RTS          ; Initialization is Complete

;-----------------------------------------------------------------------
; InitSCIHandler: Initializes the SCI Data Port Handler
;                 Clears out SCI receive and transmit buffers.
;                 Modifies A and X,SCIRMB,SCITMB,RxSCIHead,RxSCITail,RxSCIStart
; 05/06/97        Current Implementation uses ClearBuffer
;-----------------------------------------------------------------------
InitSCIHandler
  JSR  InitSCI   ; initialize the port
  LDY  #SCITMBSize
  LDX  #SCITMB
  JSR  ClearBuffer

  LDY  #SCIRMBSize
  LDX  #SCIRMB
  JSR  ClearBuffer

  CLR   RxSCICnt        ; clear number of unprocessed receive messages
  MOVB  #$FF,RxSCIHiLo ; clear flag
  LDX   #SCIRMB
  STX   RxSCIHead       ; always points to first message in the buffer
  STX   RxSCITail       ; always points to the next available buffer entry
  STX   RxSCIStart      ; addr into buffer of first byte of current message

  LDX   #SCITMB
  STX   TxSCIHead
  STX   TxSCITail
  LDD   #$00
  STD   TxSCICnt

  RTS
;------------------------End of InitSCIHandler------------------------------

;-----------------------------------------------------------------------
; TxMessageSCI - Transmit message to the SCI port. Assumes the address of the
;                first message byte is in X and that the message length
;                is MESSAGE_SIZE. It will copy the message into its own
;                buffer. SCITMB must be at least 2*MessageSize+1.
; Modifies:      SCITMB,Y,X,A,
; Current Implementation: Procedure blocks! Waits until entire message has
;                         been transmited before it returns to caller.
;                         In the future this will be interrupt driven.
; 04/26/97                Now transmits data in ASCII format.
;                         So SCITMB must be twice as long as MESSAGE_SIZE
; 04/29/97                Added Header byte to transmissions
;-----------------------------------------------------------------------
TxMessageSCI
  LDY  TxSCITail
  ; place address of destination in X
  JSR  PrepareMessage
  ; We need to make sure the tail does not need wrapped around.
  LDD  TxSCITail
  ADDD  #SCIMessageSize
  CPD   #SCITMB+$5A
  BNE   TxMessSCI_1
  LDD   #SCITMB
TxMessSCI_1
  SEI
  STD   TxSCITail
  ; increment # bytes to be transmitted
  LDD  TxSCICnt
  ADDD #SCIMessageSize
  STD  TxSCICnt

  ; now we need to make sure transmit empty holding register is on
  BSET  SC0CR2,%10000000
  CLI

  RTS   ; we are all done
;-------------------------End of TxMessageSCI----------------------------

;-----------------------------------------------------------------------
; TxNextSCI:   Called after a TDRE interrupt on the SCI Port. Takes the
;              next byte in the TMB and loads it into the shift register.
;              If the TMB is empty, turns of the transmit interrupt flag.
; Modfies      TxSCIHead, SCITMB, TXSCICnt
;-----------------------------------------------------------------------
TxNextSCI
  LDD  TxSCICnt
```

```
  BEQ  TxNextSCI_1     ; if we have no bytes to transmit

  ; otherwise we have at least one byte to load into the shift register
  LDY  TxSCIHead
  LDAA  ,Y
  STAA  SC0DRL    ; write value to be transmitted into SC Data Register
  LDX   TxSCICnt
  DEX
  STX   TxSCICnt
  INY
  STY   TxSCIHead
  CPY   #SCITMB+$5A
  BNE   TxNextSCI_fin    ; wrap around test
  LDY   #SCITMB
  STY   TxSCIHead
  BRA   TxNextSCI_fin

TxNextSCI_1          ; no bytes to transmit, so turn off interrupts
  BCLR  SC0CR2,%10000000

TxNextSCI_fin
  RTS


;-----------------------------------------------------------------------
; AddSCIMess:  Called by the event handler to actually take a message in the
;              scirmb (receive message buffer) and place it in the message
;              queue (by calling AddMessage)
;
; Modifies:    A,X,RxSCIHead
;-----------------------------------------------------------------------
AddSCIMess
  ; First, copy the message contents into the queue
  LDX   RxSCIHead             ; address of first byte in message
  JSR   AddMessage            ; add the message to the queue

  ; now advance RxSCIHead to next part of the SCIRMB
  LDD   RxSCIHead
  ADDD  #MESSAGE_SIZE         ; increment by the length of the message
;  CPD   #SCIRMB+SCIRMB_SIZE ; are we at the end of the scirmb?
  CPD   #SCIRMB+$2C
  BNE   AddSCIMess_0
  LDD   #SCIRMB               ; wrap the head around

AddSCIMess_0
  STD   RxSCIHead             ; store new head of the queue
  DEC   RxSCICnt              ; we just removed an SCI message..
  RTS
;-------------------------End of AddSCIMess------------------------------

$ENDIF

;-----------------------------------------------------------------------
; UART1.ASM
; Block / Zone Controller --> HC12 --> UART 1 Port Handler Code
;
; Routines for controlling the first UART on the DUART chip.
;
; Public Routines:  InitDU1Handler, TXMessageDU1, RXMessageDU1, AddDU1Mess
; Private Routines: InitDU1, TXDU1, RXDU1
;-----------------------------------------------------------------------

; Details:
; For transmitting, we have a buffer UA1TMB which MUST be a multiple of
; 2*MESSAGE_SIZE+1. The buffer has a head and a tail. Both wrap around.
; Messages are added to the buffer by starting at the tail location.
; The next byte to transmit is chosen by the head pointer.

; For receiving, we have a buffer UA1RMB which MUST be an multiple of
; MESSAGE_SIZE. The buffer has a head

$IFNOT INCLUDE_FLASH
InitUA1Handler
TxMessageUA1
AddUA1Mess
InitDUART1
RxMessageUA1
TxNextUA1
HandleDUARTInt
$ELSEIF
```

```
;---------------------------------------------------------------------------
; InitUA1Handler: Initializes the DU1 Data Port Handler
;                 Clears out DU1 receive and transmit buffers.
;                 Modifies A and X
;
; Notes:   MR --> PDLC2, CS --> PP4, CHSL = PB.5, A2..A0 = PB4..2
;          RD --> PB1    WR --> PB0
;---------------------------------------------------------------------------
InitUA1Handler
  JSR   InitDUART1

  ; Now that UART1 has been initialized, clear out our transmit and
  ; receive buffers.
  LDY   #UA1TMBSize
  LDX   #UA1TMB
  JSR   ClearBuffer

  LDY   #UA1RMBSize
  LDX    #UA1RMB
  JSR    ClearBuffer

  CLR   RxUA1Cnt          ; clear number of unprocessed receive messages
  ;NOTE: FOR RxUA1HiLo, bit 7 is used to record the presence of a header
  ; packet for receiving.  bit 0 is used to determine if the next nibble
  ; is going to be a hi or lo nibble
  MOVB  #$01,RxUA1HiLo  ; clear flag
  LDX   #UA1RMB
  STX   RxUA1Head         ; always points to first message in the buffer
  STX   RxUA1Tail         ; always points to the next available buffer entry
  STX   RxUA1Start        ; addr into buffer of first byte of curren message

  LDX   #UA1TMB
  STX   TxUA1Head
  STX   TxUA1Tail
  LDX   #$00
  STX   TxUA1Cnt          ; 2 byte number for # of packets in transmit buffer

  RTS

;---------------------------------------------------------------------------
; TxMessageUA1 - Adds a message to the offboard UART1 transmission queue.
;                Actual tranmission occurs by TxNextUA1. UA1TMB must be a
;                multiple of 2*MessageSize+1. Also enables ETHREI for UART1
;                as we now have data to transmit. Interrupts are disabled
;                for this action.
;
; Arguments:     X holds the address of the first byte of the message.
;                Message must have length Message_Size
; Modifies:      UA1TMB,Y,X,A,TxUA1Cnt,TxUA1Tail
;---------------------------------------------------------------------------
TxMessageUA1
  ;first, make sure buffer has space for the new message
  LDD   #UA1MessageSize

  ; First, copy the message contents into UA1TMB
  LDY  TxUA1Tail       ; place address of destination in Y
  JSR  PrepareMessage  ; add header byte, convert to ASCII, copy into UA1TMB

  ; since UA1TMB is a multiple of 2*Message_size+1, after copy each message,
  ; we need to check for the tail needing wrapped around.

  ; we need to  make sure the tail does not need to be wrapped around
  LDD   TxUA1Tail
  ADDD  #UA1MessageSize
;  CPD    #UA1TMB+UA1TMBSize
  CPD   #UA1TMB+$5A
  BNE   TxMessUA1_1
  LDD  #UA1TMB               ; wrap to beginning
  STD   TxUA1Tail

TxMessUA1_1
  SEI                        ; turn off interrupts (one cycle delay)
  STD  TxUA1Tail

  ; increment # bytes to be transmitted
  LDD    TxUA1Cnt
  ADDD   #UA1MessageSize
  STD    TxUA1Cnt

  ; Now we want to make sure the ETHREI is set because we have
```

```
  ; data to transmit.

  MOVB  #$00,DDRA
  MOVB  #$FF,DDRB
  MOVB  #%00100101,PORTB     ; want to read in IER
  BCLR  PORTP,%00010000
  LDAA  PORTA                ; load IER values into A
  BSET  PORTP,%00010000      ; turn DUART back off
  ORAA  #%00000010           ; set ETHREI, leave rest unchanged
  STAA  PORTA
  MOVB  #$FF,DDRA
  MOVB  #%00100110,PORTB     ; want to write to IER
  BCLR  PORTP,%00010000
  BSET  PORTP,%00010000
  CLI                        ; turn interrupts back on

  RTS                        ; we are all done

;----------------------------------------------------------------------
; AddUA1Mess:  Called by the event handler to actually take a message in the
;              ua1rmb (receive message buffer) and place it in the message
;              queue (by calling AddMessage)
;
; Modifies:    A,X,RxUA1Head
;----------------------------------------------------------------------

AddUA1Mess
  ; First, copy the message contents into the queue
  LDX   RxUA1Head            ; address of first byte in message
  JSR   AddMessage           ; add the message to the queue

  ; now advance RxUA1Head to next part of the UA1RMB
  LDD   RxUA1Head
  ADDD  #MESSAGE_SIZE        ; increment by the length of the message
;  CPD   #UA1RMB+UA1RMBSize  ; are we at the end of the ua1rmb?
  CPD   #UA1RMB+$2C
  BNE   AddUA1Mess_0
  LDD   #UA1RMB              ; wrap the head around

AddUA1Mess_0
  STD   RxUA1Head            ; store new head of the queue
  DEC   RxUA1Cnt             ; we just removed an UA1 message..
  RTS

;----------------------------------------------------------------------
; InitDUART1: Initialization routine for the UART 1.
;             Baud Rate = 19200bps. Receive and Transmit Com. Int. Enabled
;             8 bit data, 1 stop bit, no parity
;----------------------------------------------------------------------
InitDUART1
   LDAA  #%00010000
   ORAA  DDRP
   STAA  DDRP                 ; make PortP.4 an output (/CS for DUART)
   BSET  PORTP,%00010000      ; make sure chip select intially turned off

   BSET  PORTDLC,%00000100    ; assert Master Reset
   NOP    ; MR must be asserted for at least 500ns.
   NOP
   NOP
   NOP
   NOP
   NOP
   BCLR  PORTDLC,%00000100
   NOP
   NOP
   NOP
   NOP
   NOP

   MOVB  #$FF,DDRA            ; configure ports A and B as outputs
   MOVB  #$FF,DDRB

   ; set up the Line Control Register
   MOVB  #%10000011,PORTA     ; 8 data bits, 1 stop bit, no parity, DLAB = 1
   MOVB  #%00101110,PORTB     ; A2..A0 = 011 for Line Control, CHSL=1,WR=0
   BCLR  PORTP,%00010000      ; make the command active
   BSET  PORTP,%00010000

   ; now that DLAB is high, access the baud rate divisors
   MOVB  #$18,PORTA           ; low byte of divisor for baud rate generator
   MOVB  #%00100010,PORTB     ; access Divisor Latch (low byte)
```

```
    BCLR   PORTP,%00010000
    BSET   PORTP,%00010000

    MOVB   #00,PORTA              ; high byte of divisor for baud rate generator
    MOVB   #%00100110,PORTB
    BCLR   PORTP,%00010000
    BSET   PORTP,%00010000

    MOVB   #%00000011,PORTA        ; we need to clear the DLAB bit now
    MOVB   #%00101110,PORTB       ; access line control register
    BCLR   PORTP,%00010000
    BSET   PORTP,%00010000

    ; Now, setup the interrupts we wish to have enabled
    MOVB   #%00000001,PORTA       ; enable Received Data interrupt
    MOVB   #%00100110,PORTB       ; write to the IER
    BCLR   PORTP,%00010000
    BSET   PORTP,%00010000
    RTS

;----------------------------------------------------------------------
; RxMessageUA1:  Call to handle RDAI interrupt from UART1. Copies the
;                received packet into a message buffer. If last packet in the
;                message, increments RxUA1Cnt to notify event handler that
;                a completed message has been received.
;
; Modifies       UA1RMB,RxUA1Start,RxUA1Tail,X
;----------------------------------------------------------------------
RxMessageUA1
   ; load in the received byte
   MOVB  #$00,DDRA
   MOVB  #$FF,DDRB
   MOVB  #%00100001,PORTB     ; want to read Receive Buffer Register
   BCLR  PORTP,%00010000
   LDAA  PORTA                ; load in received byte
   BSET  PORTP,%00010000

   LDX   RxUA1Tail            ; address of where to put the new packet

   ; Examine if received byte was a HeaderByte or not.
   CMPA  #HeaderByte
   BNE   RxMessUA1_0          ; not a header byte so process normally

   ; otherwise we have received a header byte in the middle of
   ; receiving another message. ADD ERROR HANDLER CODE HERE AT LATER DATE
   LDX   RxUA1Start           ; start at beginning of current message
   STX   RxUA1Tail            ; add new bytes at this location
   MOVB  #%10000001,RxUA1HiLo ; bit 7 acks header byte, bit 0 resets hi/lo

   ; since we do NOT want to actually store the header byte,
   ; we are done at this point as the system is ready to receive the new
   ; message.
   BRA  RxMessUA1_fin

   ; Process the received byte
RxMessUA1_0
   ; if we have not received a header packet, ignore this packet
   BRCLR  RxUA1HiLo,$80,RxMessUA1_fin

   ; Test RxUA1HiLo to determine if the received packet is a high or low
   ; nibble. Add the nibble appropriately to the Tail.
   BRCLR  RxUA1HiLo,$01,RxMessUA1_1    ; branch if it is the low nibble
   LSLA
   LSLA
   LSLA
   LSLA                    ; rotate the nibble into the high nibble
   STAA   ,X               ; store the high nibble in the buffer
   BCLR   RxUA1HiLo,$01
   BRA    RxMessUA1_fin   ; we are done

RxMessUA1_1       ; we have just received a low nibble, incorporate it
   ANDA   #$0F     ; make sure the high nibble is zeroed out
   ORAA   ,X       ; combine high nibble in B with low nibble in A
   STAA   ,X       ; store the complete packet back into the buffer
   BSET   RxUA1HiLo,$01

   ; Now we need to load up and increment the Tail
   LDX    RxUA1Tail
   INX
   STX    RxUA1Tail            ; inc tail of the rmb
   LDX    #RxUA1Start
```

```
  LDD    ,X                 ; load addr of first byte in current message
  ADDD   #MESSAGE_SIZE
  CPD    RxUA1Tail          ; is it a complete message?
  BNE    RxMessUA1_fin      ; if not, our job is done

  ; otherwise we need to make sure the Tail is not at the end of the
  ; UA1RMB (if so, we need to wrap it around) AND we need to alert
  ; the event driver that we have a message to add to the queue
  INC    RxUA1Cnt             ; we have received a complete message
  MOVB   #$01,RxUA1HiLo       ; clear header flag and set hi/lo nibble flag

  LDX    RxUA1Tail
  STX    RxUA1Start            ; record new start addr of the next message
;  CPX    #UA1RMB+UA1RMBSize
  CPX    #UA1RMB+$2C             ; end of buffer?
  BNE    RxMessUA1_fin

  LDX    #UA1RMB
  STX    RxUA1Tail             ; wrap the tail around
  STX    RxUA1Start            ; wrap around start addr of next message

RxMessUA1_fin
  RTS

;-----------------------------------------------------------------------
; TxNextUA1      Sends the next byte to be transmitted to the the offboard
;                UART1. This routine should be called to handle a UART1
;                transmitter holding register empty interrupt. If there are no
;                more bytes to be transferred, it turns off ETHREI from
;                the DUART. Assumes any other off chip devices using the
;                data bus are inactive (i.e. they are critical sections)
;
; Modifies:      UA1TMB, TxUA1Head, A,Y, TxUA1Cnt
;------------------------------------------------------------------------
TxNextUA1
   LDD  TxUA1Cnt
   CPD  #00        ; do we have any bytes to transfer?
   BEQ   TxNextUA1_0

   ; otherwise we have at least one byte we can load into UART1
   MOVB  #$FF,DDRA
   MOVB  #$FF,DDRB
   LDY   TxUA1Head
   LDAA  ,Y
   STAA  PORTA
   MOVB  #%00100010,PORTB
   BCLR  PORTP,%00010000      ; execute the command
   BSET  PORTP,%00010000
   LDX   TxUA1Cnt
   DEX
   STX   TxUA1Cnt             ; record that we have transmitted a byte
   INY
   STY   TxUA1Head
;   CPY   #UA1TMB+UA1TMBSize  ; have we gone past the buffer?
   CPY   #UA1TMB+$5A
   BNE   TxNextUA1_finish
;   BNE   TxNextUA1_0
   LDY   #UA1TMB              ; wrap around
   STY   TxUA1Head
   BRA   TxNextUA1_finish


TxNextUA1_0       ; No more bytes to transmit so disable UART1 interrupt
   MOVB  #$00,DDRA
   MOVB  #$FF,DDRB
   MOVB  #%00100101,PORTB   ; want to read in IER
   BCLR  PORTP,%00010000
   LDAA  PORTA              ; load IER values into A
   BSET  PORTP,%00010000    ; turn DUART back off
   MOVB  #$FF,DDRA
   ANDA  #%11111101         ; clear ETHREI, leave rest unchanged
   STAA  PORTA
   MOVB  #%00100110,PORTB     ; want to write to IER
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000

TxNextUA1_finish
   RTS

;------------------------------------------------------------------------
; HandleDUARTInt:   Assuming an interrupt was received from the IRQ line,
```

```
;                 process it. The int is either an empty transmit buffer
;                 or a receive character interrupt.
;
; Modifies:
;---------------------------------------------------------------------------
HandleDUARTInt
  ; Check Channel 1 for source of interrupt, then Channel 2
  MOVB  #$00,DDRA
  MOVB  #%00101001,PORTB      ; Want to read IER
  BCLR  PORTP,%00010000       ; execute the command
  LDAA  PORTA
  BSET  PORTP,%00010000
  MOVB  #$FF,DDRA

  CMPA   #04                   ; test for receive interrupt
  BNE    HandleDUARTInt_0      ; it wasn't a Channel 1 received data interrupt
  JSR    RxMessageUA1          ; it was, so retrieve new data
  RTI
HandleDUARTInt_0
   CMPA  #02
   BNE   HandleDUARTInt_1
   JSR   TxNextUA1
   RTI
HandleDUARTInt_1               ; Check Channel 2 interrupts
  MOVB  #$00,DDRA
  MOVB  #%00001001,PORTB      ; Want to read IER for channel 2
  BCLR  PORTP,%00010000       ; execute the command
  LDAA  PORTA
  BSET  PORTP,%00010000
  MOVB  #$FF,DDRA

  CMPA   #04                  ; test for receive interrupt
  BNE    HandleDUARTInt_2     ; it wasn't a Channel 1 received data interrupt
  JSR    RxMessageUA2         ; it was, so retrieve new data
  RTI

HandleDUARTInt_2
   CMPA  #02                  ; empty transmit buffer message?
   BNE   HandleDUARTInt_fin
   JSR   TxNextUA2
HandleDUARTInt_fin
   RTI
$ENDIF

;---------------------------------------------------------------------------
; UART2.ASM
;
; Block / Zone Controller --> HC12 --> UART 2 Port Handler Code
;
; Routines for controlling the second UART on the DUART chip.
;
; Public Routines:  InitDU2Handler, TxMessageUA2,RxMessageUA2 AddUA2Mess,
;                   TxNextUA2
; Private Routines: InitDUART2
;---------------------------------------------------------------------------

; Details:
; For transmitting, we have a buffer UA2TMB which MUST be a multiple of
; 2*MESSAGE_SIZE+1. The buffer has a head and a tail. Both wrap around.
; Messages are added to the buffer by starting at the tail location.
; The next byte to transmit is chosen by the head pointer.

; For receiving, we have a buffer UA2RMB which MUST be an multiple of
; MESSAGE_SIZE. The buffer has a head

$IFNOT INCLUDE_FLASH
InitUA2Handler
TxMessageUA2
AddUA2Mess
InitUART2
RxMessageUA2
TxNextUA2
$ELSEIF

;---------------------------------------------------------------------------
; InitUA2Handler: Initializes the DU1 Data Port Handler
;                 Clears out DU1 receive and transmit buffers.
;                 Modifies A and X
;
; Notes:   MR --> PDLC2, CS --> PP4, CHSL = PB.5, A2..A0 = PB4..2
;          RD --> PB1    WR --> PB0
```

```
;-----------------------------------------------------------------------
InitUA2Handler
  JSR   InitDUART2

  ; Now that UART2 has been initialized, clear out our transmit and
  ; receive buffers.
  LDY   #UA2TMBSize
  LDX   #UA2TMB
  JSR   ClearBuffer

  LDY   #UA2RMBSize
  LDX    #UA2RMB
  JSR    ClearBuffer

  CLR   RxUA2Cnt         ; clear number of unprocessed receive messages
  ;NOTE: FOR RxUA2HiLo, bit 7 is used to record the presence of a header
  ; packet for receiving.  bit 0 is used to determine if the next nibble
  ; is going to be a hi or lo nibble
  MOVB  #$01,RxUA2HiLo  ; clear flag
  LDX   #UA2RMB
  STX   RxUA2Head        ; always points to first message in the buffer
  STX   RxUA2Tail        ; always points to the next available buffer entry
  STX   RxUA2Start       ; addr into buffer of first byte of curren message

  LDX   #UA2TMB
  STX   TxUA2Head
  STX   TxUA2Tail
  LDX   #$00
  STX   TxUA2Cnt         ; 2 byte number for # of packets in transmit buffer

  RTS
;------------------------End of InitUA2Handler------------------------------

;-----------------------------------------------------------------------
; TxMessageUA2 - Adds a message to the offboard UART1 transmission queue.
;                Actual tranmission occurs by TxNextUA2. UA2TMB must be a
;                multiple of 2*MessageSize+1. Also enables ETHREI for UART1
;                as we now have data to transmit. Interrupts are disabled
;                for this action.
;
; Arguments:     X holds the address of the first byte of the message.
;                Message must have length Message_Size
; Modifies:      UA2TMB,Y,X,A,TxUA2Cnt,TxUA2Tail
;-----------------------------------------------------------------------
TxMessageUA2

  ; First, copy the message contents into UA2TMB
  LDY  TxUA2Tail        ; place address of destination in Y
  JSR  PrepareMessage   ; add header byte, convert to ASCII, copy into UA2TMB

  ; since UA2TMB is a multiple of 2*Message_size+1, after copy each message,
  ; we need to check for the tail needing wrapped around.

  ; we need to  make sure the tail does not need to be wrapped around
  LDD   TxUA2Tail
  ADDD  #UA2MessageSize
;  CPD   #UA2TMB+UA2TMBSize
  CPD   #UA2TMB+$5A
  BNE   TxMessUA2_1
  LDD  #UA2TMB              ; wrap to beginning
  STD   TxUA2Tail

TxMessUA2_1
  SEI                       ; turn off interrupts (one cycle delay)
  STD  TxUA2Tail

  ; increment # bytes to be transmitted
  LDD     TxUA2Cnt
  ADDD    #UA2MessageSize
  STD     TxUA2Cnt

  ; Now we want to make sure the ETHREI is set because we have
  ; data to transmit.

  MOVB  #$00,DDRA
  MOVB  #$FF,DDRB
  MOVB  #%00000101,PORTB     ; want to read in IER
  BCLR  PORTP,%00010000
  LDAA  PORTA                ; load IER values into A
  BSET  PORTP,%00010000      ; turn DUART back off
  ORAA  #%00000010           ; set ETHREI, leave rest unchanged
```

```
  STAA  PORTA
  MOVB  #$FF,DDRA
  MOVB  #%00000110,PORTB     ; want to write to IER
  BCLR  PORTP,%00010000
  BSET  PORTP,%00010000
  CLI                        ; turn interrupts back on

  RTS                        ; we are all done

;-------------------------------------------------------------------------
; AddUA2Mess:  Called by the event handler to actually take a message in the
;              ua1rmb (receive message buffer) and place it in the message
;              queue (by calling AddMessage)
;
; Modifies:    A,X,RxUA2Head
;-------------------------------------------------------------------------

AddUA2Mess
  ; First, copy the message contents into the queue
  LDX   RxUA2Head            ; address of first byte in message
  JSR   AddMessage           ; add the message to the queue

  ; now advance RxUA2Head to next part of the UA2RMB
  LDD   RxUA2Head
  ADDD  #MESSAGE_SIZE        ; increment by the length of the message
;  CPD   #UA2RMB+UA2RMBSize  ; are we at the end of the ua1rmb?
  CPD   #UA2RMB+$2C
  BNE   AddUA2Mess_0
  LDD   #UA2RMB              ; wrap the head around

AddUA2Mess_0
  STD   RxUA2Head            ; store new head of the queue
  DEC   RxUA2Cnt             ; we just removed an UA2 message..
  RTS

;-------------------------------------------------------------------------
; InitDUART1: Initialization routine for the UART 1.
;             Baud Rate = 9200bps. Receive and Transmit Com. Int. Enabled
;             8 bit data, 1 stop bit, no parity
;-------------------------------------------------------------------------
InitDUART2
   LDAA  #%00010000
   ORAA  DDRP
   STAA  DDRP                 ; make PortP.4 an output (/CS for DUART)
   BSET  PORTP,%00010000      ; make sure chip select intially turned off
   BCLR  PORTDLC,%00000100    ; make sure MR is off

   MOVB  #$FF,DDRA            ; configure ports A and B as outputs
   MOVB  #$FF,DDRB

   ; set up the Line Control Register
   MOVB  #%10000011,PORTA     ; 1 stop bit, no parity, DLAB = 1
   MOVB  #%00001110,PORTB     ; A2..A0 = 011 for Line Control, CHSL=1,WR=0
   BCLR  PORTP,%00010000      ; make the command active
   BSET  PORTP,%00010000

   ; now that DLAB is high, access the baud rate divisors
   MOVB  #$18,PORTA            ; low byte of divisor for baud rate generator
   MOVB  #%00000010,PORTB     ; access Divisor Latch (low byte)
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000

   MOVB  #00,PORTA            ; high byte of divisor for baud rate generator
   MOVB  #%00000110,PORTB
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000

   MOVB  #%00000011,PORTA      ; we need to clear the DLAB bit now
   MOVB  #%00001110,PORTB     ; access line control register
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000

   ; Now, setup the interrupts we wish to have enabled
   MOVB  #%00000001,PORTA     ; enable Received Data interrupt
   MOVB  #%00000110,PORTB     ; write to the IER
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000
   RTS

;-------------------------------------------------------------------------
; RxMessageUA2:  Call to handle RDAI interrupt from UART1. Copies the
```

```
;                received packet into a message buffer. If last packet in the
;                message, increments RxUA2Cnt to notify event handler that
;                a completed message has been received.
;
; Modifies       UA2RMB,RxUA2Start,RxUA2Tail,X
;----------------------------------------------------------------------
RxMessageUA2:
  ; load in the received byte
  MOVB  #$00,DDRA
  MOVB  #$FF,DDRB
  MOVB  #%00000001,PORTB     ; want to read Receive Buffer Register
  BCLR  PORTP,%00010000
  LDAA  PORTA                ; load in received byte
  BSET  PORTP,%00010000

  LDX   RxUA2Tail            ; address of where to put the new packet

  ; Examine if received byte was a HeaderByte or not.
  CMPA  #HeaderByte
  BNE   RxMessUA2_0          ; not a header byte so process normally

  ; otherwise we have received a header byte in the middle of
  ; receiving another message. ADD ERROR HANDLER CODE HERE AT LATER DATE
  LDX   RxUA2Start           ; start at beginning of current message
  STX   RxUA2Tail            ; add new bytes at this location
  MOVB  #%10000001,RxUA2HiLo ; bit 7 acks header byte, bit 0 resets hi/lo

  ; since we do NOT want to actually store the header byte,
  ; we are done at this point as the system is ready to receive the new
  ; message.
  BRA  RxMessUA2_fin

  ; Process the received byte
RxMessUA2_0
  ; if we have not received a header packet, ignore this bit
  BRCLR  RxUA2HiLo,$80,RxMessUA2_fin

  ; Test RxUA2HiLo to determine if the received packet is a high or low
  ; nibble. Add the nibble appropriately to the Tail.
  BRCLR  RxUA2HiLo,$01,RxMessUA2_1    ; branch if it is the low nibble
  LSLA
  LSLA
  LSLA
  LSLA                ; rotate the nibble into the high nibble
  STAA   ,X           ; store the high nibble in the buffer
  BCLR   RxUA2HiLo,$01
  BRA    RxMessUA2_fin   ; we are done

RxMessUA2_1       ; we have just received a low nibble, incorporate it
  ANDA   #$0F     ; make sure the high nibble is zeroed out
  ORAA   ,X       ; combine high nibble in B with low nibble in A
  STAA   ,X       ; store the complete packet back into the buffer
  BSET   RxUA2HiLo,$01

  ; Now we need to load up and increment the Tail
  LDX    RxUA2Tail
  INX
  STX    RxUA2Tail            ; inc tail of the rmb
  LDX    #RxUA2Start
  LDD    ,X                   ; load addr of first byte in current message
  ADDD   #MESSAGE_SIZE
  CPD    RxUA2Tail            ; is it a complete message?
  BNE    RxMessUA2_fin        ; if not, our job is done

  ; otherwise we need to make sure the Tail is not at the end of the
  ; UA2RMB (if so, we need to wrap it around) AND we need to alert
  ; the event driver that we have a message to add to the queue
  INC    RxUA2Cnt               ; we have received a complete message
  MOVB   #$01,RxUA2HiLo         ; clear header flag and set hi/lo nibble flag

  LDX    RxUA2Tail
  STX    RxUA2Start              ; record new start addr of the next message
;  CPX    #UA2RMB+UA2RMBSize
  CPX    #UA2RMB+$2C               ; end of buffer?
  BNE    RxMessUA2_fin

  LDX    #UA2RMB
  STX    RxUA2Tail               ; wrap the tail around
  STX    RxUA2Start              ; wrap around start addr of next message

RxMessUA2_fin
```

```
    RTS

;-----------------------------------------------------------------------
; TxNextUA2      Sends the next byte to be transmitted to the the offboard
;                UART1. This routine should be called to handle a UART1
;                transmitter holding register empty interrupt. If there are no
;                more bytes to be transferred, it turns off ETHREI from
;                the DUART. Assumes any other off chip devices using the
;                data bus are inactive (i.e. they are critical sections)
;
; Modifies:      UA2TMB, TxUA2Head, A,Y, TxUA2Cnt
;-----------------------------------------------------------------------
TxNextUA2
   LDD  TxUA2Cnt
   CPD  #00         ; do we have any bytes to transfer?
   BEQ   TxNextUA2_0

   ; otherwise we have at least one byte we can load into UART1
   MOVB  #$FF,DDRA
   MOVB  #$FF,DDRB
   LDY   TxUA2Head
   LDAA  ,Y
   STAA  PORTA
   MOVB  #%00000010,PORTB
   BCLR  PORTP,%00010000       ; execute the command
   BSET  PORTP,%00010000
   LDX   TxUA2Cnt
   DEX
   STX   TxUA2Cnt              ; record that we have transmitted a byte
   INY
   STY   TxUA2Head
;   CPY   #UA2TMB+UA2TMBSize  ; have we gone past the buffer?
   CPY   #UA2TMB+$5A
   BNE   TxNextUA2_finish
   LDY   #UA2TMB               ; wrap around
   STY   TxUA2Head
   BRA   TxNextUA2_finish

TxNextUA2_0        ; No more bytes to transmit so disable UART1 interrupt
   MOVB  #$00,DDRA
   MOVB  #$FF,DDRB
   MOVB  #%00000101,PORTB   ; want to read in IER
   BCLR  PORTP,%00010000
   LDAA  PORTA              ; load IER values into A
   BSET  PORTP,%00010000    ; turn DUART back off
   MOVB  #$FF,DDRA
   ANDA  #%11111101         ; clear ETHREI, leave rest unchanged
   STAA  PORTA
   MOVB  #%00000110,PORTB     ; want to write to IER
   BCLR  PORTP,%00010000
   BSET  PORTP,%00010000

TxNextUA2_finish
   RTS
$ENDIF
```

Protocol Tools:

```
;--------------------------------------------------------------------------
; EventHan.ASM
;
; Block / Zone Controller --> HC12 --> Event Handler Code
;
; Supported Routines: InitEventHandler, HandleEHInt, AddEvent
;
; Code to support the event handler. Events are added to the EH buffer. Each
; event has an associated time value in terms of the smallest time increment.
; At every minimum time increment, the EH (Event Handler) decrements the time
; field for each record. When a time field for a record turns to zero, it is
; time for the event to occur. The EH then sets the appropriate bits for
; specified flag.

; An Event Record in the buffer is 5 bytes. The first 2 bytes specify the
; address of a one byte flag. The next 2 bytes hold the time count till the
; event. The last byte is used as a mask for the flag. When the time count
; goes to zero, the EH uses the mask to set the appropriate bits on the
; particular event flag.
;
; The flag mask works as follows: every bit that is set in the mask will cause
; the corresponding bit to be set when the time count reaches zero. Any bit
; in the mask that is zero implies that that bit in the flag will be untouched.
;
; Note: If the ADDRESS for the flag is zero then the associated record is
; considered empty. We can only have NumEvent events pending.


$IFNOT   INCLUDE_FLASH
InitEventHandler
AddEvent
HandleEHInt
$ELSEIF

;--------------------------------------------------------------------------
; InitEventHandler -- Sets up the EH buffer, clearing all the records and
;                     setting up the compare module to be used for the timer.
; Note: TC5 is used as a compare module for the event handler
;--------------------------------------------------------------------------
InitEventHandler
  ; initialize the variables
  LDX   #EHBuffer
  LDY   #EHBufferSize
  JSR   ClearBuffer          ; clear the EH buffer
  CLR   EventHanCnt          ; clear number of pending events


  ; now initialize TC5 which is going to be used ONLY by the event handler
  BCLR   TCTL1,%00001100     ; disconnect Channel 5 from its output pin
  BCLR   TMSK1,%00100000     ; Turn OFF the interrupt enable for now
  BSET   TIOS,%00100000      ; Channel 5 is now in compare mode

  RTS

;--------------------------------------------------------------------------
; AddEvent -- Finds an available slot in the EHBuffer and places the event in
;             the slot. Does NOT add the event if the EHBuffer is full, we
;             will eventually want to add an error handling routine here.
;             Interupts are briefly turned off while the buffer is modified.
;
; Args:       The event information must be stored in an event record
;             (EHRecord) before calling AddEvent. As such, you can NEVER
;             load EHRecord while inside an interrupt!!! Otherwise, you
;             could be trashing a value used by another caller.
;             EHRecord must be filled as follows:
;             First two bytes are the address for the flag to be set,
;             next two bytes are time count for the event expressed in number
;             of EHMinimumDelay counts. Last byte is the flag mask
;
; Returns:     ACCAA has a value of 1 if the event record could not be added
;              because the buffer was full. It returns 0 otherwise
;--------------------------------------------------------------------------
AddEvent
  LDAA   EventHanCnt
  CMPA   #NumEvents
  BEQ    AddEvent_full         ; no more space in the event buffer

  ; otherwise, search for first free record slot in the buffer by starting
  ; at the start of the buffer (offsetting by 2 so we can look at the tag/
  ; flag address field
```

```
  LDX    #EHBuffer
AddEvent_0
  LDY    ,X                    ; load in flag address value
  BEQ    AddEvent_1
  LDAB   #EventRecSize
  ABX                          ; increment x to the next record
  BRA    AddEvent_0

  ; we have found the empty record so insert event data stored on the stack.
AddEvent_1
  SEI                          ; disable interrupts while we modify EHBuffer
  LDY   #EHRecord
  LDD   ,Y
  STD    ,X                    ; store flag address

  LDD    2,Y                   ; extract time count for the event
  STD    2,X                   ; store time count for event

  LDAA   4,Y                   ; extract flag mask
  STAA   4,X                   ; store flag mask

  LDAA  EventHanCnt
  BNE   AddEvent_2             ; test if there are already pending events
  LDD   TCNT
  ADDD  #EHDelayInterval
  STD   TC5
  BSET  TMSK1,%00100000        ; enable EHHandler interrupt

AddEvent_2
  INC    EventHanCnt
  CLI
  LDAA   #$00                  ; no errors
  BRA    AddEvent_fin

AddEvent_full
  LDAA   #EHBufferFull

AddEvent_fin
  RTS

;-------------------------------------------------------------------------
; HandleEHInt:     Handle event handler interrupt by decrementing the count
;                  field for each pending event in the queue. If the count
;                  field is zero after the decrement then set the appropriate
;                  flag bits and clear the event record.
;-------------------------------------------------------------------------
HandleEHInt
  LDAA   EventHanCnt
  BEQ    HandleEHInt_5     ; any pending events?

  ; otherwise, parse through buffer, examining each record and adjusting the
  ; count
; LDX    #EHBuffer-#EventRecSize
  LDX    #EHBuffer-$05

HandleEHInt_0
  LDAB    #EventRecSize
  ABX
;  CMPX    #EHBuffer+EHBufferSize
  CPX    #EHBuffer+$32
  BEQ     HandleEHInt_fin
  LDD    ,X
  BEQ    HandleEHInt_0     ; if record is not empty, process it

  ; if it is not empty, advance to time count, decrement it and test if it
  ; has reached zero or not. X holds address of flag and is at the start
  ; of the record.
  LDY   2,X                ; load time count value
  DEY
  STY   2,X                ; store it
  BNE  HandleEHInt_0       ; if it was not zero, move on to next event

  LDAA  4,X                ; load the flag mask
  LDY   ,X                 ; handle extra level of indirection
  ORAA  ,Y                 ; apply the mask to the flag
  STAA  ,Y                 ; store updated flag
  LDD   #$00
  STD   ,X                 ; clear out the flag address to clear record
  DEC   EventHanCnt        ; we just removed a pending event...
  BRA   HandleEHInt_0
```

```
HandleEHInt_5         ; no active records in the event handler so turn int. off
  BCLR   TMSK1,%00100000   ; turn off the interrupt

HandleEHInt_fin
  ; reload timer
  LDD  TCNT
  ADDD  #EHDelayInterval    ; add time till next interrupt
  STD  TC5                  ; store next interrupt time
  RTS
$ENDIF

;--------------------------------------------------------------------------
; Vehicle.ASM
;
; Block / Zone Controller --> HC12 --> Vehicle Creation / Display Code
;
; Routines: InitVehicleList, AddVehicle, UpdateVehicles
;--------------------------------------------------------------------------

; Overview:
; A vehicle is represented as a vehicle record. In a vehicle record, the first
; byte is a vehicle ID (VID), second byte is vehicle postion (VPOS) and the last
; two bytes are for the vehicle velocity (VVEL). VVEL is expressed in terms of
; the number of 50usec counts until it moves to the next position. VPOS is
; expressed in terms of the position on the track segment for the current
; block.

; The vehicle manager has been augmented to manage vehicles and their
; positions on the track. The vehicle manager uses a PostionLocked flag
; to determine if a particular position is locked or not. Each bit in the
; flag corresponds to a track position. Bit 0 is position zero and bit 7 is
; position 7. When updating a vehicle position, if the next position is
; locked (occupied) by another vehicle, the manager sets a blocked flag
; or the blocked vehicle. If a vehicle is blocked, its velocity value
; is no longer used to form an event. The vehicle is considered stopped!!
; Every time a position is unlocked, the vehicle list is scanned for blocked
; vehicles and their position is updated, with an event being set
; for in the next position update.

; NOTE: VID = 0 is used to designate an empty vehicle record

$IFNOT INCLUDE_FLASH
InitVehicleList
AddVehicle
RemoveVehicle
UpdateVehicles
UpdateBlkVehicles
$ELSEIF
;--------------------------------------------------------------------------
; InitVehicleList:  Clears vehicle record list. Clears VehicleCnt and
;                   VehicleStatus.
; Arguments         None
; Modifies:         VehicleList, VehicleCnt,
;--------------------------------------------------------------------------
InitVehicleList
  LDX   #VehicleBuffer
  LDY   #VehicleBufferSize
  JSR   ClearBuffer

  CLR   VehicleCnt
  CLR   VehicleStatus
  CLR   PositionStatus   ; 0 = free, 1 = occupied. Each bit = a position
  RTS

VPosString  FCB  'VPOS: ',0

;--------------------------------------------------------------------------
; RemoveVehicle -- Takes a VID in A and removes it from the vehicle list
;                  also updates position status. VPOS is stored in B.
;                  Remove vehicle if VID and VPOS match arguments
; Arguments:       VID in A, VPOS in B
; Modifies:        VehicleBuffer, PositionStatus
;--------------------------------------------------------------------------
RemoveVehicle
  LDX  #VehicleBuffer
  LDY #$08    ; eight entries to be examined
RemoveVeh_0
  CMPA VID,X
  BNE  RemoveVeh_cont
```

```
  CMPB  VPOS,X
  BNE   RemoveVeh_cont

RemoveVeh_fou
  ; otherwise we have found a match
  DEC  VehicleCnt
  LDAA VPOS,X
  EORA PositionStatus
  STAA PositionStatus
  LDAA #$00
  STAA VID,X
  LDAA  VPOS,X
  JSR  ClearVehicle

  JSR  UpdateBlkVehicles     ; update any vehicles waiting on us to move
  BRA  RemoveVeh_fin

RemoveVeh_cont
  PSHD
  LDAB #VehicleRecSize
  ABX
  PULD

  DEY
  BNE  RemoveVeh_0
RemoveVeh_fin
  RTS

;----------------------------------------------------------------------------
; UpdateVehicles: Called to process one or more of the vehicle status bits
;                 being set. Updates the vehicles position. The vehicle record
;                 is cleared from the list if the new positioin is at the end
;                 of the block (i.e. >= NumPositions).
; Note:           The vehicle is not advanced if the next position is occupied.
;                 Instead, the vehicle is blocked.
; Modifies:       VehicleStatus, VehicleBuffer, VehicleCnt
;----------------------------------------------------------------------------
UpdateVehicles
  LDAA   VehicleCnt
  LBEQ    UpdateVehicles_fin

  ; otherwise, parse through buffer, examining each record and its associated
  ; status bit. If the status bit for a record is high, we need to update
  ; its position.

  ; A is going to keep track of the current status bit. We will shift these
  ; out to the right such that bit 0 always holds the status of the current
  ; record being examined.

  ; B is going to keep track of the current flag position which can be
  ; used as a flag mask later

  SEI
  LDAB   #$01
  LDAA   VehicleStatus      ; load status bits
  CLR   VehicleStatus
  CLI

; LDX #VehicleBuffer-#VehicleRecSize
  LDX    #VehicleBuffer

UpdateVehicles_0
  LSRA           ; rotate record status bit into carry bit
  BCC  UpdateVehicles_cont

  ; otherwise we have a vehicle we can update
  PSHD

  ; make sure the record is not empty (vehicle removed before event)
  LDAA  VID,X
  BNE   UpdateVehicles_A
  PULD
  BRA   UpdateVehicles_cont

UpdateVehicles_A
  LDAA  VPOS,X
  CMPA  #LastPosition
  BEQ  UpdateVehicles_1    ; for now, just erase vehicle when it
                           ; reaches the end

  ; first, make sure position is free for the new position
```

```
  LDAA  VPOS,X
  LSLA        ; shift over one position
  ANDA PositionStatus
  BEQ UpdateVeh_free    ; position is free

  ; otherwise, we must mark ourselves as blocked and remark the position as
  ; locked
;  STAB  PositionStatus   ; restore position status
  LDAA  #Blocked
  STAA  VBLOCKED,X
  PULD
  BRA   UpdateVehicles_cont

UpdateVeh_free
  LDAA  VPOS,X
  PSHX
  JSR   ClearVehicle
  PULX

  ; need to clear out current position
  LDAA  VPOS,X
  EORA  PositionStatus      ; any 1-1 pair becomes a 0, rest unchanged
  STAA  PositionStatus

  LDAA   VPOS,X
  LSLA   ; advance over one spot
  STAA   VPOS,X
  ORAA   PositionStatus
  STAA   PositionStatus

  LDAA   VPOS,X
  PSHX
  JSR  DisplayVehicle
  PULX

  ; lastly, create a new event record
  PULD
  LDY   #EHRecord
  STAB  EHFlagMask,Y   ; store our flag mask
  PSHD
  PSHX
  LDD   VVEL,X     ; load the velocity
  STD   EHTime,Y
  LDD  #VehicleStatus
  STD  EHFlagAddr,Y
  JSR   AddEvent
  PULX
  PULD
  BRA   UpdateVehicles_cont

UpdateVehicles_1   ; vehicle has reached end so initiate a vehicle exchange
  PSHX
  JSR   GenVehExch
  PULX
  PULD
UpdateVehicles_cont   ; advance to next entry
   PSHB
   LDAB  #VehicleRecSize
   ABX   ; update X to next record
   PULB
   LSLB  ; update mask to next record
   TBNE  A,UpdateVehicles_0
   JSR   UpdateBlkVehicles

UpdateVehicles_fin
  JSR  UpdateFirstPos
  RTS
;-----------------------------------------------------------------------------

;-----------------------------------------------------------------------------
; UpdateFirstPos:  After updating vehicles, checks if first position is free
;                  or locked. If it is free and Pending VID exists, calls
;                  routine to generate a vehicle exchange continue
;-----------------------------------------------------------------------------
UpdateFirstPos
  BRSET   PositionStatus,$01,UpdateFirstPos_fin
  LDAA    PendingVID
  BEQ     UpdateFirstPos_fin   ; abort if no pending VID
  ; otherwise, call GenVehExchCont
  JSR     genvehExchCont
```

```
UpdateFirstPos_fin
  RTS

;---------------------------------------------------------------------------
; UpdateBlkVehicles: Called after a vehicle position has been updated.
;                    Runs through the list and frees any vehicle that
;                    was blocked but whose next position is no longer
;                    blocked.
; Modifies:          PositionStatus, VehicleBuffer
;---------------------------------------------------------------------------
UpdateBlkVehicles
  LDAA  VehicleCnt
  BEQ   UpdateBlkVeh_fin
  LDAB   #$01           ; b is our index into the buffer, when it reaches
                        ; zero, we have examined all vehicles
  LDX   #VehicleBuffer

UpdateBlkVeh_0
  LDAA VID,X
  BEQ  UpdateBlkVeh_next   ; if VID == 0, then empty slot
  LDAA  VBlocked,X
  BEQ   UpdateBlkVeh_next  ; if vehicle is not blocked, ignore it

  ; we have found a blocked vehicle
  LDAA  VPOS,X
  LSLA
  ANDA PositionStatus
  BNE  UpdateBlkVeh_next  ; if not zero, position is still occupied
  LDAA  VPOS,X

  PSHB
  PSHX
  JSR   ClearVehicle    ; clear current position
  PULX
  LDAA  VPOS,X
  EORA PositionStatus   ; free our position
  STAA PositionStatus

 LDAA  VPOS,X
 LSLA
 STAA VPOS,X          ; store new position
 ORAA PositionStatus
 STAA PositionStatus

 LDAA  VPOS,X
 PSHX
 JSR  DisplayVehicle
 PULX
 LDAA  #Free          ; vehicle is NOT blocked anymore
 STAA  VBlocked,X

; now create an event for it
  LDY   #EHRecord
  STAB  EHFlagMask,Y   ; store our flag mask
  PSHD
  PSHX
  LDD   VVEL,X     ; load the velocity
  STD   EHTime,Y
  LDD  #VehicleStatus
  STD  EHFlagAddr,Y
  JSR   AddEvent
  PULX
  PULD

  PULB

UpdateBlkVeh_next
   PSHB
   LDAB   #VehicleRecSize
   ABX
   PULB
   LSLB
   TBNE   B,UpdateBlkVeh_0
UpdateBlkVeh_fin
   JSR    UPdateFirstPos
   RTS

;---------------------------------------------------------------------------
; AddVehicle:    Takes vehicle data from VehicleRecord, finds an empty entry
;                in the vehicle buffer and inserts the new vehicle. It then
;                inserts the new vehicle in the empty slot. If we have reached
```

```
;                 MaxVehicleCnt then the vehicle is NOT added.
;                 If VID of new vehicle is already in vehicle list, then
;                 we do not add the vehicle but DO return VehileAdded message.
;                 This is used to enforce at most once semantics on vehicle
;                 additions.
; Arguments:      new vehicle data stored in VehicleRec
; Modifies:       VehicleCnt, VehicleBuffer
; 05/25/97        If VPOS is locked by another vehicle, add vehicle fails and
;                 returns a PositionOccupied error message. Returns
;                 VehicleAdded if vehicle was in fact added
;-----------------------------------------------------------------------
AddVehicle
  LDAA   VehicleCnt
  CMPA   #MaxNumVehicles
  BEQ    AddVehicle_full

  LDX   #VehicleRec

  LDY   #VehicleBuffer
  LDAB   #$08
AddVehicleDuplicate
  LDAA  VID,Y
  CMPA  VID,X
  BEQ   AddVeh_dupFound
  PSHB
  LDAB  #VehicleRecSize
  ABY
  PULB
  DBNE  B,AddVehicleDuplicate
  BRA   AddVehicleNoDup

AddVeh_dupFound
  LDAA  #VehicleAdded
  RTS

AddVehicleNoDup
  ; make sure the position for the new vehicle is not locked
  LDAA  VPOS,X
  ANDA  PositionStatus
  ; now, since only one bit in VPOS is set, the exclusive or will return
  ; zero if the bit set in VPOS matches the one set in PositionStatus. It
  ; returns 1 otherwise.
  BEQ  AddVehicle_A
  ; otherwise we cannot add the vehicle
  LDAA  #PositionOccupied
  RTS

AddVehicle_A      ; we are ready to add the vehicle
; First, mark the position as being taken
  LDAA  VPOS,X
  ORAA  PositionStatus
  STAA  PositionStatus

  ; ACCB is going to keep track of the available slot we are going to insert
  ; the new vehicle into. i.e. bit n means inserting into record n
  LDAB   #$01              ; B is going to keep track of the entry we insert into

  ; search for first free record slot in the list
  LDX    #VehicleBuffer
AddVehicle_0
  LDAA   VID,X       ; examine VID for current record
  BEQ    AddVehicle_1
  LDAA   #VehicleRecSize    ; advance to next record
  EXG    A,B
  ABX
  EXG    A,B
  LSLB                      ; adjust bit flag
  BRA     AddVehicle_0

AddVehicle_1          ; we have found an empty record
  SEI                 ; turn interrupts off while we modify the buffer
  ; copy the vehicle information into the available slot (pointed to by X)
  LDY     #VehicleRec
  LDAA    VID,Y
  STAA    VID,X
  LDAA    VPOS,Y
  STAA    VPOS,X
  PSHD
  LDD     VVEL,Y
  STD     VVEL,X
  LDAA    #$00
```

```
  STAA    VBlocked,X      ; currently vehicle is not blocked
  PULD

  INC     VehicleCnt
  CLI

  ; we also need to add the vehicle position change to the event handler
  ; we only do this if the velocity is non-zero!!! otherwise the vehicle
  ; does not move
  LDY     VVEL,X
  BEQ     AddVehicle_cont

  LDY     #EHRecord
  STAB    EHFlagMask,Y   ; store our flag mask
  LDD     VVEL,X     ; load the velocity
  STD     EHTime,Y
  LDD     #VehicleStatus
  STD     EHFlagAddr,Y
  PSHX
  JSR     AddEvent
  PULX

AddVehicle_cont
  ; for now, display new vehicle position to the screen
  LDAA    VPOS,X
  JSR     DisplayVehicle
  LDAA    #VehicleAdded       ; set return code
  RTS

AddVehicle_full
  LDAA   #TrackFull
  RTS
$ENDIF
```

Protocols:

```
;---------------------------------------------------------------------------
; Ping.ASM
;
; Block / ZoneController --> HC12 --> Ping Protocol Routines
;
; These routines are used to implement the Ping Protocol. The ping protocol
; is a SPECIAL low level protocol which periodically checks the connection
; for each Port Handler. If the ping generator, does not receive a response
; from the host connected at the other end of a port handler, then the
; connection is reported as being down and the routing table is updated
; with a new connection. Subsequent initiations of the Ping Protocol will
; attempt to re-establish contact with the port.
;
; Flags: ALIVE    bit 7:  Ping Timeout
;                 bit 6:  Ping Pending
;                 bit 0:  SCI Port Response
;                 bit 1:  DUART 1 Port Response
;                 bit 2:  DUART 2 Port Response
;---------------------------------------------------------------------------
$IFNOT INCLUDE_FLASH
HandlePingReq
ClrRTStr
ZCDownStr
HandlePingInt
HandlePingRes
GenPingReq
InitPingProtocol
HanPingTimeout
$ELSEIF

;--------------------------------------------------------------------------
; GenPingReq:         Sends out ping requests to all known sources.
;                     Also clears the flags in the ALIVE variable. ALIVE
;                     should be examined after appropriate time has been
;                     given for message replies.
; Modifies:           ACCA,X,MessBuff
;--------------------------------------------------------------------------
GenPingReq
  LDAA  #%00001000
  EORA  PortDLC             ; HACK!!! Toggle HC12 bit here
  STAA  PortDLC
  LDX   #MessBuffer
  LDY   #Message_Size
  JSR   ClearBuffer

  LDX   #MessBuffer
  LDAA  #Ping_Request
  STAA  MessCode,X                 ; store message code

  ; gen ping request is different for each element in the system
$IF ZONE_CONTROLLER
  LDAA  #ZoneController
  STAA  MessSrc,X
  ; send to original comm link used between ZC and Left Block
  LDAA  #LeftBlock
  STAA  MessDest,X
  LDX   #LeftBlock
  LDY   #MessBuffer
  JSR   RouteToPort         ; special router call, routes to default link

  LDX   #MessBuffer
  LDAA  #RightBlock
  STAA  MessDest,X
  LDX   #RightBlock
  LDY   #MessBuffer
  JSR   RouteToPort

  LDX   #MessBuffer
  LDAA  #CentralController
  STAA  MessDest,X
  LDX   #CentralController
  LDY   #MessBuffer
  JSR   RouteToPort
$ELSEIF
  ; block controllers have right and left neighbors, ping them
  LDAA  #RightNeighbor
  STAA  MessSrc,X
  LDAA  #LeftNeighbor       ; ping link to left neighbor
```

```
  STAA  MessDest,X
  LDX   #LeftNeighbor
  LDY   #MessBuffer
  JSR   RouteToPort          ; special router call, routes to default link

  LDX   #MessBuffer
  LDAA  #LeftNeighbor
  STAA  MessSrc,X
  LDAA  #RightNeighbor        ; ping link to right neighbor
  STAA  MessDest,X
  LDX   #RightNeighbor
  LDY   #MessBuffer
  JSR   RouteToPort          ; special router call, routes to default link

  ; the only catch is that the LB and RB must be distinguished when
  ; pinging their respective lines to the zone controller. So their
  ; block position comes into play
  LDX  #MessBuffer
  LDAA #ZoneController
  STAA MessDest,X
  LDAA #SelfID           ; are we a right or left block?
  STAA  MessSrc,X
  LDX   #ZoneController
  LDY   #MessBuffer
  JSR   RouteToPort
$ENDIF

 ; We want to place a timeout into the Event Handler. Our flag will
 ; be bit 7 of ALIVE
   LDX  #EHRecord
   LDAA #%10000000
   STAA EHFlagMask,X            ; store flag mask (high bit in this case)
   LDD  #PingDelay
   STD  EHTime,X            ; store delay count
   LDD  #ALIVE
   STD  EHFlagAddr,X              ; store address of flag variable
   JSR  AddEvent
   MOVB #%01000000,ALIVE ; clear our flags, but set bit 6 (ping pending)
   RTS
;-----------------------End of GenPingReq---------------------------------

;-----------------------------------------------------------------------
; HandlePingReq:      Called to handle reciept of a PingRequest.
;                     A PingRequest message has the sender's represenation
;                     with respect to us in the 2nd byte. We simply send
;                     a Ping Response to the specified source.
;                     Assumes HEAD points to the first byte in the message
;                     which is of size Message_Size
;
; Modifes:            ACCA,X,MessBuff
;-----------------------------------------------------------------------
HandlePingReq
  LDX   #MessBuffer
  LDY   #Message_Size
  JSR   ClearBuffer       ; get rid of any residue in the MessBuffer
  LDX   #MessBuffer
  LDAA  #Ping_Response    ; message code for the response
  STAA  MessCode,X        ; store the message code

  ; we want to send the ping response back to the sender
  LDY   QueueHead
  LDAA  MessSrc,Y
  STAA  MessDest,X

  ; now we need to fill in the source. What is our relationship
  ; with the sender? (ZC, LB, RB, LN, RN, CC)
  TFR   A,X
  LDAB  ConvertTable,X    ; determine what we are with respect to the source

  LDY   #MessBuffer
  STAB  MessSrc,Y         ; store what we are

  ; reload destination into X
  TFR A,X               ; A holds destination
  LDY   #MessBuffer
  JSR   RouteToPort       ; transmit the message
  RTS

; use message source to look into this table to determine what we
; represent from the sender. i.e if source is our right neighbor,
; the we are the sources left neighbor.
```

```
$IFNOT ZONE_CONTROLLER
ConvertTable   FCB   SelfID        ; if src = ZC, we are RB
               FCB   RightNeighbor     ; if src was a LN, we are the RN
               FCB   LeftNeighbor      ; if src was a RN, we are the LN
$ELSEIF
ConvertTable   FCB   ZoneController
               FCB   ZoneController      ; if src was LB, we are ZC
               FCB   ZoneController
$ENDIF

;-------------------------------------------------------------------------
; HandlePingRes:  Examines source of the ping response and sets the
;                 appropriate bit in the ALIVE flag, acknowledging that
;                 the source is alive.
;
; Modifes:              ACCA,X,MessBuffer,ALIVE
;-------------------------------------------------------------------------
HandlePingRes
  LDX   QueueHead
  LDAA   MessSrc,X       ; load in the source of the message

; If we are the ZC, sources are CC (set bit 0), LB (set bit 1) and RB
; (set bit 2)
$IF ZONE_CONTROLLER
  ; based on this value, set appropriate bits in ALIVE register.
  CMPA   #CentralController
  BNE   HandlePingRes_0
  BSET  ALIVE,%00000001
HandlePingRes_0              ; Did our left neighbor report in?
   CMPA   #LeftBlock
   BNE   HandlePingRes_1
   BSET  ALIVE,%00000010
HandlePingRes_1
   CMPA   #RightBlock
   BNE   HandlePingRes_2
   BSET  ALIVE,%00000100
$ELSEIF
  ; based on this value, set appropriate bits in ALIVE register.
  CMPA   #ZoneController
  BNE   HandlePingRes_0
  BSET  ALIVE,%00000001
HandlePingRes_0              ; Did our left neighbor report in?
   CMPA   #LeftNeighbor
   BNE   HandlePingRes_1
   BSET  ALIVE,%00000010
HandlePingRes_1
   CMPA   #RightNeighbor
   BNE   HandlePingRes_2
   BSET  ALIVE,%00000100
$ENDIF

HandlePingRes_2
   RTS            ; unknown type

;-------------------------------------------------------------------------
; InitPingProtocol -   Right now, we only have to initialize ALIVE variables.
; Note:                Bit 7 of ALIVE is set whenever TC6 interrupt is
;                      generated. Bit 6 is set whenever a GenPingReq
;                      is issued. The remaining bits signify response
;                      status from ports. GenPingReq actually turns on the
;                      interrupt.
;-------------------------------------------------------------------------
InitPingProtocol
  MOVB   #%00111111,ALIVE     ; clear bit flags.
  MOVB   #%00111111,ALIVEOld

  ; we also want to report all links as down to start off with all links up
  LDAA  #$9B
  JSR  SetLCDAddr
$IFNOT ZONE_CONTROLLER
  LDX  #ClrStrLN
  JSR  PrintString
  LDX  #ClrStrRN
  JSR  PrintString
$ELSEIF
  LDX  #ClrStrLB
  JSR  PrintString
  LDX  #ClrStrRB
  JSR  PrintString
$ENDIF
  LDAA #$DB
```

```
  JSR   SetLCDAddr
$IFNOT ZONE_CONTROLLER
  LDX   #ClrStrZC
  JSR   PrintString
$ELSEIF
  LDX   #ClrStrCC
  JSR   PrintString
$ENDIF
  LDX   #APDownStr
  JSR   PrintString
  RTS

ClrStrZC  FCB 'ZC: Up ',0
ZCDownStr FCB 'ZC: Dn ',0
ClrStrLN  FCB 'LN: Up ',0
LNDownStr FCB 'LN: Dn ',0
ClrStrRN  FCB 'RN: Up',0
RNDownStr FCB 'RN: Dn',0
APDownStr FCB 'AP: Dn',0
ClrStrAP  FCB 'AP: Up',0

; zone controller strings
ClrStrLB  FCB 'LB: Up ',0
LBDownStr FCB 'LB: Dn ',0
ClrStrRB  FCB 'RB: Up',0
RBDownStr FCB 'RB: Dn',0
ClrStrCC  FCB 'CC: Up ',0
CCDownStr FCB 'CC: Dn ',0

Isolated FCB 'Isolated!',0


;---------------------------------------------------------------------------
; HanPingTimeOut:  Called to handle a timer interrupt where it is time to
;                  examine responses from a ping request.
;
;---------------------------------------------------------------------------
HanPingTimeOut
  BRCLR  ALIVE,%01000000,HPTOut_fin1    ; make sure a  ping request was issued
  BCLR   ALIVE,%11000000
  BRA    HanPingTimeoutCont

HPTOut_fin1
  MOVB  ALIVE,ALIVEOLD
  BCLR  TMSK1,%01000000     ; turn off interrupt enable
  CLR   ALIVE

HanPingTimeoutCont
  ; exclusive OR the old value and the new value...if 0 then no change so
  ; don't PRINT anything!!
  LDAA   ALIVEOLD
  EORA   ALIVE
  STAA   ALIVEOLD

HPTOut_ZC
  ; First examine the zone controller
;  BRCLR  ALIVEOLD,%00000001,HPTOut_LN   ; determine if there has been a change
  ; if there has been, then we need to act on it
  LDAA   #$DB
  JSR    SetLCDAddr
  BRSET  ALIVE,%00000001,HPTOut_ZCup       ; Test zone controller (SCI Port)
  ; if it is not set, we need to print appropriate information
$IFNOT ZONE_CONTROLLER
  ; if zone controller is down, try routing through one of the
  ; neighbors.
  ; try right neighbor first
;  BRCLR  ALIVE,%00000100,ZCDN_RBDN      ; test if RB is up
;  LDAA    #RNID
;  LDAB    #ZCID             ; update zone controller entry
;  JSR     UpdateEntry       ; map messages through RB
;  BRA     ZCDown
;ZCDN_RBDN
  ; since right and zone controllers are down, try re-routing through
  ; left neighbor
;  BRCLR  ALIVE,%00000010,ZCDown  ; test if RB is up
;  LDAA    #LNID
;  LDAB    #ZCID                  ; update zone controller entry
;  JSR     UpdateEntry            ; map messages through RB
;  BRA     ZCDown
;ZCDown
  LDX  #ZCDownStr          ; print the down string
```

```
$ELSEIF
  LDX  #CCDownStr
$ENDIF
  JSR  PrintString
  BRA  HPTOut_LN

HPTOut_ZCUp
$IFNOT ZONE_CONTROLLER
;  LDAA #ZCID
;  JSR  RestoreEntry
  LDX #ClrStrZC          ; print out updating routing table string
$ELSEIF
  LDX #ClrStrCC
$ENDIF
  JSR  PrintString

  ; test left neighbor or left block link
HPTOut_LN                    ; Test if Left Neighbor Out
;  BRCLR ALIVEOLD,%00000010,HPTOut_RN  ; determine if there has been a change
  LDAA  #$9B
  JSR   SetLCDAddr
  BRSET ALIVE,%00000010,HPTOut_LNUp

$IFNOT ZONE_CONTROLLER
  ; if the left neighbor is currently down, try to route through the
  ; right neighbor, if the right neighbor is down, try though the
  ; zone controller...otherwise we are isolated
  BRCLR   ALIVE,%00000100,LNDN_RNDN
  LDAA    #RNID
  LDAB    #LNID
  JSR     UpdateEntry
  BRA     LNDN_RNDN_ZCDN

LNDN_RNDN
  BRCLR   ALIVE,%00000001,LNDN_RNDN_ZCDN
  ; otherwise, we will forward through zone controller
  LDAA  #ZCID
  LDAB  #LNID
  JSR   UpdateEntry     ; change entry to route through right neighbor

LNDN_RNDN_ZCDN          ; otherwise we are isolated
  LDX  #LNDownStr
$ELSEIF
  ; if the left neighbor is currently down, try to route through the
  ; RB. First, make sure RB is alive
  BRCLR   ALIVE,%00000100,LBDN_RBDN      ; test if RB is up
  LDAA    #RNID
  LDAB    #LB_ID1   ; update Left neighbor entry
  JSR     UpdateEntry                    ; map messages through RB
  LDAA    #RNID
  LDAB    #LB_ID2
  JSR     UpdateEntry
LBDN_RBDN
  LDX  #LBDownStr
$ENDIF

  JSR  PrintString
  BRA  HPTOut_RN

HPTOut_LNUp
$IFNOT ZONE_CONTROLLER
  ; we need to restore the routing table values for the left block
  LDAA #LNID
  JSR  RestoreEntry
  LDX  #ClrStrLN
$ELSEIF
  ; we need to restore the routing table values for the left block
  LDAA #LB_ID1
  JSR  RestoreEntry
  LDAA #LB_ID2
  JSR  RestoreEntry
  LDX  #ClrStrLB
$ENDIF
  JSR  PrintString

HPTOut_RN
;  BRCLR ALIVEOLD,%00000100,HPTOut_fin
  LDAA  #$A2
  JSR   SetLCDAddr
  BRSET ALIVE,%00000100,HPTOut_RNUp
  ; if bit is not set, we need to print down information
```

```
$IFNOT ZONE_CONTROLLER
  ; if right neighbor is down, try routing through the left neighbor
  BRCLR  ALIVE,%00000010,RNDN_LNDN
  LDAA   #LNID
  LDAB   #RNID
  JSR    UpdateEntry
  BRA    RN_AllDown

RNDN_LNDN     ; both neighbors are down, try through zone controller
  BRCLR  ALIVE,%00000001,RN_AllDown
  LDAA  #ZCID
  LDAB  #RNID
  JSR   UpdateEntry     ; change entry to route through right neighbor

RN_AllDown         ; every connection is down, we are isolated.
  LDX  #RNDownStr

$ELSEIF
  ; if the right is currently down, try to route through the
  ; LB. First, make sure LB is alive
  BRCLR  ALIVE,%00000010,RBDN_LBDN      ; test if LB is up
  LDAA   #LNID
  LDAB   #RB_ID1   ; update Left neighbor entry
  JSR    UpdateEntry                    ; map messages through RB
  LDAA   #LNID
  LDAB   #RB_ID2
  JSR    UpdateEntry

RBDN_LBDN
  LDX #RBDownStr

$ENDIF

  JSR  PrintString
  BRA  HPTOut_fin

HPTOut_RNUp              ; the right neighbor / block is up!
$IFNOT ZONE_CONTROLLER
  ; we need to restore the routing table values for the right neighbor
  LDAA #RNID
  JSR  RestoreEntry
  LDX  #ClrStrRN
$ELSEIF
  ; we need to restore the routing table values for the right block
  LDAA #RB_ID1
  JSR  RestoreEntry
  LDAA #RB_ID2
  JSR  RestoreEntry
  LDX  #ClrStrRB
$ENDIF
 JSR  PrintString

HPTOut_fin
  MOVB  ALIVE,ALIVEOLD
  BCLR  TMSK1,%01000000    ; turn off interrupt enable
  CLR   ALIVE
  RTS

$ENDIF

;---------------------------------------------------------------------------
; Virtual.ASM
;
; Block / Zone Controller --> HC12 --> Virtual Vehicle Protocol routines
;
; Overview: Protocol routines for creating a virtual vehicle. Typically, the
; zone controller begins the protocol by sending a VV-Gen message to
; a block controller. The VV-Gen message type contains information for
; a vehicle record. Upon receiving a VV-Gen message, the block controller
; calls the vehicle manager, giving it the vehicle record stored in the
; message. The block controller then replies to the zone with a VV-Ack
; message. Currently, if the ZC has not received a VV-Ack by a time-out,
; then an error message is displayed to the LCD.
;---------------------------------------------------------------------------
$IFNOT INCLUDE_FLASH
GenVVAck
HandleVVGEN
HandleVVACK
HanVVTimeout
GenVVMess
$ELSEIF
```

```
;-----------------------------------------------------------------------
; GenVVMess:        Generates a VVMess and sends it to the specified block
;                   controller. Also loads an event for the VV timeout.
;                   Assumes that the caller is the zone controller
; Arguments:        A contains the message destination, and the vehicle data
;                   is stored in temporary vehicle recrod: VehicleRec.
; Modifies:         MessBuffer
;------------------------------------------------------------------------
GenVVMess
  PSHA                  ; save destination
  LDX   #MessBuffer
  LDY   #Message_Size
  JSR   ClearBuffer
  PULA                  ; restore destination

  LDX   #MessBuffer
  LDAB  #VirtualVehGen
  STAB  MessCode,X      ; store message type

  STAA  MessDest,X      ; store destination
  LDAA  #MYIDLN        ; store source ID
  STAA  MessSrc,X

  LDAA  #Message_Size
  STAA  MessSize,X

  ; now copy vehicle event record information into data field of message
  LDY  #VehicleRec
  LDAA VID,Y
  STAA MessData,X
  INX
  LDAA  VPOS,Y
  STAA MessData,X
  INX
  LDD  VVEL,Y
  STD  MessData,X

  ; now transmit the data
  LDY  #MessBuffer
  LDAA  MessDest,Y
  TFR   A,X
  JSR  RouteMessage

  ; add add event record for the timeout
  ; bit 7 is the flag given to the event handler, bit 6 is to signify that
  ; there is a timeout pending and bit 1 is set when an ACK is received.
  MOVB  #%01000000,VVMFlag  ; we are going to have a timeout pending

  LDY   #EHRecord
  LDAA  #%10000000       ; use bit 7 as the timeout flag
  STAA  EHFlagMask,Y
  LDD   #VVMTimeCnt      ; generate timeout in .25s
  STD   EHTime,Y
  LDD   #VVMFlag
  STD   EHFlagAddr,Y
  JSR   AddEvent

  RTS                   ; we are now done

;-----------------------------------------------------------------------
; HandleVVAck - To acknowledge the VV-ACK, we must set the VVMFlag ACK
;               received bit which is bit 0. Remember, bit 7 is set by the
;               event handler when the timeout occurrs.
; Arguments   - QueueHead points to the ACK message
; Modifies    - VVMFlag bit 0
;
;-----------------------------------------------------------------------
HandleVVAck
  ; make sure a timeout was pending
  BRCLR  VVMFLAG,%01000000,HandleVVAck_fin    ; ignore if no timeout pending
  BSET   VVMFlag,$01

HandleVVACK_fin
  RTS

;-----------------------------------------------------------------------
; GenVVAck:   With a VVGen message pointed to by QueueHead, this routine
;             generates a VV-ACK message and sends it back to the gen sender.
; Arguments:  VVGen message pointed to by QueueHead
; Modifies:   MessBuffer
```

```
;---------------------------------------------------------------------------
GenVVAck
  LDX   #MessBuffer
  LDY   #Message_Size
  JSR   ClearBuffer

  LDX   #MessBuffer
  LDY   QueueHead
  LDAA  #VVAckMess
  STAA  MessCode,X           ; store message type
  LDAA  MessSrc,Y
  STAA  MessDest,X

  LDAA  MessDest,Y
  STAA  MessSrc,X           ; sender knows who we are so use that as source
  LDAA  #Message_Size
  STAA  MessSize,X

  ; the data field is left empty
  LDAA  MessSrc,Y
  LDY   #MessBuffer
  TFR   A,X
  JSR   RouteMessage
  RTS

;---------------------------------------------------------------------------
; HandleVVGen:      Handles a VV-Gen message by loading the data contents
;                   into VehicleRec and invoking add message of the vehicle
;                   manager. Also calls GenVVAck.
;
; Arguments:        Message starts at QueueHead
;---------------------------------------------------------------------------
HandleVVGen
  LDY   #VehicleRec
  LDX   QueueHead
  ; copy vehicle data contents into vehicle Rec
  LDAA  MessData,X
  STAA  VID,Y
  INX
  LDAA  MessData,X
  STAA  VPOS,Y
  INX
  LDD   MessData,X
  STD   VVEL,Y
  JSR   AddVehicle        ; call the vehicle manager and add the vehicle

  JSR   GenVVAck
  RTS

;---------------------------------------------------------------------------
; HanVVTimeout - Handles the virtual vehicle creation timeout. If bit 0
;                  of VVMFlag is not set, we print an error to the screen.
; Modifies:        VVMFlag, LCD Display
; Arguments:       None
;---------------------------------------------------------------------------
HanVVTimeout
  BRSET   VVMFlag,$01,HanVVTimeout_0
  ; otherwise we did not receive the ACK
  ; print out an error message
  LDAA  #$C0
  JSR   SetLCDAddr
  LDX   #VVMString
  JSR   PrintStrToBanner

HanVVTimeout_0
  CLR   VVMFlag
  RTS

VVMString  FCB  'VVC Failed!',0

$ENDIF

;---------------------------------------------------------------------------
; VehExch.ASM
; Block Controller --> HC12 --> Vehicle Exchange Protocol Routines
;---------------------------------------------------------------------------

$IFNOT  INCLUDE_FLASH
GenVehExch
GenVehExchAck
HanVehExch
```

```
HanGenVehExchAck
HanGenVehExchTO     ; time out
GenVehExchBlock
HanGenVehExchBlock
$ELSEIF

;------------------------------------------------------------------------------
; GenVehExch:    Generate a Vehicle Exchange Message. This message is always
;                sent to the left neighbor as vehicles flow from left to
;                right. It begins the protocol of exchanging a vehicle
;                between two blocks.
; Arguments:     X points to a vehicle record for the vehicle to be exchanged.
;                the vehicle is in the last position of the current block.
; Modifies:      MessBuffer
;------------------------------------------------------------------------------
GenVehExch
  PSHX
  LDX  #MessBuffer
  LDY  #Message_Size
  JSR  ClearBuffer
  PULX

  TFR  X,Y              ; copy pointer of vehicle data
  LDX  #MessBuffer
  LDAA  #VehicleExchange
  STAA  MessCode,X

  LDAA  #LNID
  STAA  MessDest,X
  LDAA  #MYIDLN
  STAA  MessSrc,X

   LDAA  VID,Y
   STAA  MessData,X
   STAA  ExchVID
   INX
   LDD  VVEL,Y
   STD  MessData,X
   STD  ExchVVEL

   LDX #LNID
   LDY #MessBuffer
   JSR RouteMessage

   ; add timeout to the event handler
   LDX  #EHRecord
   LDAA #$80            ; msb of flag is the VehExch time out pending flag
   STAA EHFlagMask,X
   LDD  #VehExchCnt
   STD  EHTime,X
   LDD  #VehExchFlag
   STD  EHFlagAddr,X
   JSR  AddEvent

   RTS
;------------------------------------------------------------------------------

;------------------------------------------------------------------------------
; HanVehExch     Called to handle reciept of a vehicle exchange message.
;                Takes the vehicle information, forms a vehicle record and
;                calls addvehicle.  Then returns a VehExchACK
;------------------------------------------------------------------------------
HanVehExch
;  BSET  VehExchFlag,$40     ; acknowledge block continue if applicable
  LDX   QueueHead
  LDY   #VehicleRec
  LDAA  MessData,X       ; extract the VID
  STAA  VID,Y
  INX
  LDD   MessData,X
  STD   VVEL,Y
  LDAA  #$01
  STAA  VPOS,Y
  ; now that vehicle record is ready, call AddVehicle
  JSR   AddVehicle
  ; test return value
  CMPA   #PositionOccupied
;  BEQ   GenVehExchBlock    ; position was blocked, generate block message

  ; otherwise, vehicle was added
  BRA  GenVehExchACK
```

```
;--------------------------------------------------------------------------
; GenVehExchAck  Acknowledging successful exchange. Recipient can now delete the
;                vehicle from the list. Message is sent from the right
;                neighbor to the left neighbor.
;                Also assumes queuehead points to a vehicle exchange message
; Modifies           MessBuffer
;--------------------------------------------------------------------------
GenVehExchACK
  CLR  PendingVID    ; we no longer have anyone waiting for first position
  LDX  #MessBuffer
  LDY  #Message_Size
  JSR  ClearBuffer

  LDX   #MessBuffer
  LDAA  #VehicleExchACK
  STAA  MessCode,X
  LDAA  #RNID
  STAA  MessDest,X
  LDAA  #MYIDRN
  STAA  MessSrc,X

  LDY  QueueHead
  LDAA MessData,Y         ; extract the VID
  STAA MessData,X

  LDY  #MessBuffer
  LDX  #RNID
  JSR  RouteMessage
  RTS

;--------------------------------------------------------------------------
; HanVehExchACK -  Process a vehicle exchange acknowledgment by removing the
;                  vehicle from the track. Calls remove message on the VID
;                  contained in the ACK messsage
; Modifies:        Vehicle list
;--------------------------------------------------------------------------
HanVehExchACK
  BSET  VehExchFlag,$01     ; acknowledge receipt of the message
  LDX   QueueHead
  ; we want to skip over to the message data portion
  LDAA  MessData,X    ; extract the VID
  LDAB  #LastPosition
  JSR   RemoveVehicle
  CLR   ExchVID
  LDX  #$00
  STX  ExchVVEL
  RTS

;--------------------------------------------------------------------------
; HanVehExchTO:   Vehicle Exchange time out handler. If bit 0 or eventually,
;                 bit 1 is set, do nothing, an ACK has been received.
;                 Otherwise resend the message.
;--------------------------------------------------------------------------
HanVehExchTO
  BRSET   VehExchFlag,$01,HanVehExchTO_fin
  LDX     #VehicleRec
  LDAA    ExchVID
  STAA    VID,X
  LDD     ExchVVEL
  STD     VVEL,X         ; genvehexch expects X to hold the pointer
  JSR     GenVehExch     ; resend request
  LDX    #VehExchString
  JSR    PrintStrToBanner

HanVehExchTO_fin
  BCLR     VehExchFlag,$01
  BCLR     VehExchFlag,$80     ; clear veh exch timeout pending bit
  RTS

VehExchString   FCB   'V ',0

;--------------------------------------------------------------------------
; GenVehExchBlock:  Generated in response to a failed attempt to meet
;                   a vehicle exchange because the position was blocked.
;                   Does NOT require an ACK, as the original sender will
;                   timeout and re-send VehExch if there is a failure
; Arguments:        Assumes Queuehead points to a VehExch message
;--------------------------------------------------------------------------
GenVehExchBlock
  LDX   QueueHead
```

```
  LDAA  MessData,X       ; extract the VID
  STAA  PendingVID       ; we have a pending block

  LDX  #MessBuffer
  LDY  #Message_Size
  JSR  ClearBuffer

  LDX  #MessBuffer
  LDAA  #VehicleExchBLK
  STAA  MessCode,X
  LDAA  #RNID
  STAA  MessDest,X
  LDAA  #MYIDRN
  STAA  MessSrc,X

  LDY  QueueHead        ; store VID in first byte of data field
  LDAA  MessData,Y
  STAA  MessData,X

  LDY   #MessBuffer
  LDX   #RNID
  JSR   RouteMessage
  RTS

;---------------------------------------------------------------------------
; HanVehExchBLK:     Handles a VehicleExchBLK message
;                    Toggle the VehExch timeout flag (bit 7) to acknowledge
;                    the response. Nothing else.
; Arguments:         Message pointed to by queue head
;---------------------------------------------------------------------------
HanVehExchBLK
   BSET    VehExchFlag,$01     ; acknowledge response
   RTS

;---------------------------------------------------------------------------
; GenVehExchCont:    Generates message instructing the right neighbor to
;                    continue with the previously blocked exchange. Call
;                    only when the first position in in the block becomes free.
;                    Set a timeout on this message to make sure receiver gets
;                    it. Receiving a Vehicle Exchange message clears this flag.
; Arguments:         None. PendingVID holds VID of incoming vehicle
;---------------------------------------------------------------------------
GenVehExchCont
  LDX   #MessBuffer
  LDY   #Message_Size
  JSR   ClearBuffer

  LDX   #MessBuffer
  LDAA  #VehicleExchCont
  STAA  MessCode,X

  LDAA  #RNID
  STAA  MessDest,X
  LDAA  #MYIDRN
  STAA  MessSrc

  LDAA  PendingVID
  STAA  MessData,X

  LDY   #MessBuffer
  LDX   #RNID
  JSR   RouteMessage

  ; now create timeout
   LDX  #EHRecord
   LDAA #$40              ; 6th bit is the VehExchCont time out pending flag
   STAA EHFlagMask,X
   LDD  #VehExchCnt
   STD  EHTime,X
   LDD  #VehExchFlag
   STD  EHFlagAddr,X
   JSR  AddEvent

  RTS

;---------------------------------------------------------------------------
; HanVehExchCont        Resend HanVehExch Message
; Arguments             None
;---------------------------------------------------------------------------
HanVehExchCont
  LDX      #VehicleRec
```

```
  LDAA    ExchVID
  BEQ      HanVehExchCont_fin ; if VID == 0, we do NOT have a vehicle to send
  STAA    VID,X
  LDD     ExchVVEL
  STD     VVEL,X          ; genvehexch expects X to hold the pointer
  JSR     GenVehExch      ; resend request
HanVehExchCont_fin
  RTS


;---------------------------------------------------------------------------
; HanVehExchContTO:  Vehicle Exchange continue time out handler. If we time out,
;                    clear flags, and transmit vehicle continue again
; Arguments:         None
; Modfies:
;---------------------------------------------------------------------------
HanVehExchContTO
  BRSET   VehExchFlag,$40,HanVehContTO_fin
  JSR     GenVehExchCont            ; resend request
  LDX    #VehExchContString
  JSR    PrintStrToBanner

HanVehContTO_fin
  BCLR     VehExchFlag,$01
  BCLR     VehExchFlag,$80     ; clear veh exch timeout pending bit
  RTS
VehExchContString    FCB  'C ',0

;---------------------------------------------------------------------------
; InitVehExch:         Initializes the vehicle exchange protocol
; Modifies:            VehExchFlag
;---------------------------------------------------------------------------
InitVehExch
  CLR      PendingVID
  CLR      VehExchFlag
  CLR      ExchVID
  LDD      #$00
  STD      ExchVVEL
  RTS
```

Miscellaneous:

```
;-----------------------------------------------------------------------
; Display.ASM
;
; Display Routines: Specific routines for the LCD Display
;
; Public Methods: PrintStatus, UpdateQSize,
; All methods assume InitLCD has previously been called
;-----------------------------------------------------------------------
$IFNOT INCLUDE_FLASH
PrintStatus
UpdateQSize
UpdateChkFail
DisplayVehicle
ClearVehicle
DisplayTrack
$ELSEIF

;-----------------------------------------------------------------------
; DisplayVehicle:   Prints a vehicle on the track
; Arguments:        The track position (0-0A) is stored in ACCA
; Modifies:         nothing
;-----------------------------------------------------------------------
DisplayVehicle
   ; need to convert the vehicle position which is a binary number
   ; to an actual address. i.e. having bit 1 set implies address of 1
   ; for printing the vehicle
   CMPA   #$00
   BEQ    DisplayVehicle_fin
   LDAB   #$00
DisplayVehicle_0
   LSRA
   BCS   DisplayVehicle_1
   INCB
   BRA   DisplayVehicle_0
DisplayVehicle_1
   ; when done, B has address to load into LCD
   TFR   B,A
   JSR   SetLCDAddr     ; assumes A already holds the vehicle position
   LDAA  #VehicleChar
   JSR   PrintChar
DisplayVehicle_fin
   RTS

;-----------------------------------------------------------------------
; ClearVehicle:  Replaces vehicle with the track character
; Arguments:     ACCA holds bit position to be cleared
; Modifies:      Nothing
;-----------------------------------------------------------------------
ClearVehicle
   CMPA   #$00
   BEQ    ClearVehicle_fin
   LDAB   #$00
ClearVehicle_0
   LSRA
   BCS   ClearVehicle_1
   INCB
   BRA   ClearVehicle_0
ClearVehicle_1
   TFR   B,A
   JSR   SetLCDAddr    ; assumes A already holds the position to be cleared
   LDAA  #TrackChar
   JSR     PrintChar
ClearVehicle_fin
   RTS

;-----------------------------------------------------------------------
; DisplayTrack:   Prints the track to the screen. Namely, prints 8 track
;                 characters to the screen starting at 0.
; Modifies:       Nothing
;-----------------------------------------------------------------------
DisplayTrack
  LDAA    #$00
  JSR     SetLCDAddr
  LDY     #$08
DisplayTrack_0
  LDAA    #TrackChar
  JSR     PrintChar
  DBNE    Y,DisplayTrack_0
```

```
  RTS

;------------------------------------------------------------------------
; PrintStatus  - Prints the status of the system to the screen.
;                Status information includes: the number of messages
;                currently in the Queue,
;
; Modifies:      ACCA,X,ACCB
;------------------------------------------------------------------------
PrintStatus
   JSR LCDClrDisp  ; clear display and place cursor in the home position
   LDAA  #$1E
   JSR  SetLCDAddr
   ; First print out "QSize" String
   LDX  #QSIZE_STRING
   JSR  PrintString

   ; Print the Qsize string
   JSR  UpdateQSize

   ; Print the Check Sum Error String
   LDAA  #$DE                   ; print string at start of second line
   JSR  SetLCDAddr
   LDX  #ERROR_STRING
   JSR  PrintString
   JSR  UpdateChkFail
   RTS

;----------------------End of PrintStatus--------------------------------

;------------------------------------------------------------------------
; UpdateQSize: Prints the number of messages in the queue
; Assumes:     The qsize is in variable: QSize
; Modifies:    ACCA, X
;------------------------------------------------------------------------
UpdateQSize
   ; set the appropriate characters position
   LDAA #26       ; ninth character on the line
   JSR SetLCDAddr
   LDAA QSIZE
   JSR PrtNum         ; print the qsize
   RTS
;---------------End of UpdateQSize---------------------------------------

;------------------------------------------------------------------------
; UpdateChkFail:  Prints the number of checksum failures
; Assumes:        # failures is in the variable ChkSumFailures
; Modifies:       ACCCA,X
;------------------------------------------------------------------------
UpdateChkFail
    LDX  #Error_String
    JSR  PrintStrToBanner
    LDAA ChkSumFailures
    JSR PrtNumToBanner
    RTS
;--------------------End of UpdateChkFail--------------------------------

QSIZE_STRING   FCB 'Q Size: '
               FCB 0
ERROR_STRING   FCB 'ChkSum: '
               FCB 0
$ENDIF

;------------------------------------------------------------------------
; CheckSum.ASM
; Block Controller --> HC12 --> Checksum Routines
;
; Public Methods: CompChkSum
; Private Methods: None
;------------------------------------------------------------------------

;------------------------------------------------------------------------
; CompChkSum: Computes a 2 byte checksum on a message. Result is passed back
;             with high byte in A and low byte in X
; Arguments:  X contains the address of the first byte of the message
;             ACCA contains the number of bytes to be computed in the
;             checksum.
; Note: Result is returned in ChkSumHi and ChkSumLo. SHARED VARIABLES!!
;       So this routine cannot be called from within an interrupt.
; Modifies:   ChkSumHi, ChkSumLo, ACCA,ACCB,X
;------------------------------------------------------------------------
```

```
CompChkSum
   CLR     ChkSumHi
   CLR     ChkSumLo      ; clear our two byte accumulator
   CMPA    #00
   BEQ     CompChkSum_Done  ; 0 bytes needed to be computed

CompChkSum_0
   LDAB    ,X
   ADDB     ChkSumLo          ; add to low byte
   STAB     ChkSumLo
   CLRB
   ADCB    ChkSumHi          ; add carry bit to high byte of checksum
   STAB    ChkSumHi

   INX                      ; advance to next byte to be computed
   DECA
   BNE     CompChkSum_0      ; finished?

CompChkSum_Done
   RTS                       ; return result

;------------------------------------------------------------------------
; LCD.ASM
;
; LCD Routines: InitLCD, PrintString, PrintChar, PrintNum, LCDClrDisp
;               SetLCDAddr
;------------------------------------------------------------------------

$IFNOT INCLUDE_FLASH
HandleLCD
InitLCD
LCDClrDisp
SetLCDAddr
PrintChar
Delay5ms
PrintString
PrtNum
$ELSEIF

;------------------------------------------------------------------------
; HandleLCD:  Called to handle a Timer 7 interrupt. We must examine contents
;             of the LCDBuffer and determine if there are commands or actions
;             to perform. Also updates compare register for T7 to next
;             appropriate value. (Turns it off, if buffer is empty)
; Modifies:   TMSK1,LCDBuff,LCDHead,X,Y,ACCD
;------------------------------------------------------------------------
HandleLCD
  LDX  LCDHead
  CPX  LCDTail
  BEQ  HandleLCD_Empty    ; make sure buffer has something to send

  ; So buffer is not empty. Display first character in buffer.
  LDAA ,X
  CMPA  #ClearDispCode
  BNE  HandleLCD_0
  ; otherwise, clear the display
  JSR  LCDClrDisp_
  BRA  HandleLCD_4
HandleLCD_0
  CMPA #ChgAddrCode
  BNE  HandleLCD_1          ; if it isn't, it must be a char
  INX
  ;  CPX  #LCDBuffer+LCDBufferSize
  CPX   #LCDBuffer+$40
  BNE   HandleLCD_1A
  LDX   #LCDBuffer

HandleLCD_1A                ; need to load the new address to go to
  LDAA  ,X                  ; load the new address
  JSR   SetLCDAddr_
  BRA   HandleLCD_4           ; done
HandleLCD_1               ; otherwise, it is a character, so display the char.
  JSR  PrintChar_
HandleLCD_4             ; now increment head to next value, testing for wraparound
  INX
  STX   LCDHead
;  CPX  #LCDBuffer+LCDBufferSize
  CPX   #LCDBuffer+$40
  BNE   HandleLCD_fin
  LDX   #LCDBuffer
  STX   LCDHead
```

```
HandleLCD_Empty          ; the buffer is empty
  BCLR  TMSK1,%10000000   ; turn off interrupt for T7.

HandleLCD_Fin
  RTS

;----------------------------------------------------------------------
; PrintString --> Copy the null terminated string into the LCDBuffer.
;                Assumes X has the address of the first byte in the string
; Modifies       LCDBuffer,LCDTail,
;----------------------------------------------------------------------
PrintString
  LDY   LCDTail
PrintString_0
  LDAA  ,X                 ; load character
  BEQ  PrintString_2      ; test if value is NULL
  STAA ,Y
  INX
  INY
 ; test for wrap around
;  CPX  #LCDBuffer+LCDBufferSize
  CPY  #LCDBuffer+$40
  BNE  PrintString_1
  LDY  #LCDBuffer     ; wrap tail around
PrintString_1
  STY  LCDTail        ; store new value for Tail
  BRA  PrintString_0
PrintString_2
  BRSET 7,TMSK1,PrintString_fin
  BSET TMSK1,%10000000   ; otherwise, turn the timer on
PrintString_fin
  RTS

;----------------------------------------------------------------------
; PrintChar    Places a single character held in ACCA and puts it into the
;              LCDBuffer.
; Modifies     LCDBuffer,LCDTail
;----------------------------------------------------------------------
PrintChar
  LDX  LCDTail
  STAA ,X
  INX
  ; test for wrap around
;   CPX  #LCDBuffer+LCDBufferSize
  CPX  #LCDBuffer+$40
  BNE  PrintChar_1
  LDX  #LCDBuffer     ; wrap tail around
PrintChar_1
  STX  LCDTail        ; store new value for Tail
  ; otherwise, turn interrupt on
  BRSET 7,TMSK1,PrintChar_fin
  BSET TMSK1,%10000000   ; otherwise, turn the timer on
PrintChar_fin
  RTS

;----------------------------------------------------------------------
; PrintChar_    Actually prints the character currently pointed to by LCDTail,
;               to the LCD Display. Assumes the LCD is NOT busy.
; Arguments:    ACCA has the value of the character to be printed
;----------------------------------------------------------------------
PrintChar_
  MOVB  #$FF,DDRA
  MOVB  #$FF,DDRB
  STAA  PORTA
  MOVB  #%10000000,PORTB
  BSET  PORTP,%00100000   ; enable entry mode set
  BCLR  PORTP,%00100000
  LDD  TCNT            ; load current time
  ADDD  #Add50us       ; time until next operation on LCD can be performed
  STD  TC7
  RTS

;----------------------------------------------------------------------
; LCDClrDisp - Inserts command to clear the LCD display and place the cursor
;              in the home position into the LCDBuffer.
;
; Modifies:    LCDBuffer,LCDTail
;----------------------------------------------------------------------

LCDClrDisp
```

```
   LDAA  #ClearDispCode
   LDX   LCDTail
   STAA  ,X              ; store command in buffer
   INX                   ; increment tail
  ; test for wrap around
;   CPX  #LCDBuffer+LCDBufferSize
   CPX  #LCDBuffer+$40
   BNE  LCDClrDisp_0
   LDX  #LCDBuffer       ; wrap tail around
LCDClrDisp_0
   STX  LCDTail          ; store new value for Tail
   BRSET 7,TMSK1,LCDClrDisp_fin ; test if interrupt is on or not
   BSET TMSK1,%10000000 ; otherwise turn the interrupt on
LCDClrDisp_fin
   RTS

;---------------------------------------------------------------------------
; LCDClrDisp_   Actually performs a clear display. Assumes the LCD is NOT busy
;               performing any operations.
;
; Modifies      LCDBuffer,LCDHead,ACCA,X
;---------------------------------------------------------------------------
LCDClrDisp_
   MOVB #$FF,DDRA
   MOVB #$FF,DDRB
   LDAA #%00000001
   STAA PORTA
   CLR  PORTB
   BSET  PORTP,%00100000   ; enable entry mode set
   BCLR  PORTP,%00100000
   LDD  TCNT             ; load current time
   ADDD  #Add2ms         ; delay until next operation can be performed on LCD
   STD  TC7
   RTS

;--------------------------------------------------------------------------
; SetLCDAddr: Places command and address into the LCD Buffer for execution
; Arguments:   ACCA holds the desired address where 00 = home position and
;              $C0 = first position of 2nd row.
;--------------------------------------------------------------------------
SetLCDAddr
   LDAB  #ChgAddrCode
   LDX   LCDTail
   STAB  ,X              ; store command in buffer
   INX                   ; increment tail
  ; test for wrap around
;   CPX  #LCDBuffer+LCDBufferSize
   CPX  #LCDBuffer+$40
   BNE  SetLCDAddr_0
   LDX  #LCDBuffer       ; wrap tail around
SetLCDAddr_0
;   STX  LCDTail          ; don't store new tail till ALL the data is stored
   STAA ,X               ; store the address
   INX
   ; test for wrap around
;  CPX  #LCDBuffer+LCDBufferSize
   CPX  #LCDBuffer+$40
   BNE  SetLCDAddr_1
   LDX  #LCDBuffer
SetLCDAddr_1
   STX  LCDTail
   BRSET 7,TMSK1,SetLCDAddr_fin ; test if interrupt is on or not
   BSET TMSK1,%10000000 ; otherwise turn the interrupt on
SetLCDAddr_fin
   RTS

;------------------------------------------------------------------------
; SetLCDAddr_:  Sets the DGRAM address for the LCD.
; Arguments:    ACCA holds the desired character position where 00 = the
;               home position and C0 = the first char of the 2nd row.
;               Assumes LCD Display is NOT busy.
; Modifies:     ACCA, X
;------------------------------------------------------------------------
SetLCDAddr_
   MOVB    #$FF,DDRA
   MOVB    #$FF,DDRB
   ORAA    #%10000000
   STAA    PORTA
   CLR     PORTB
   BSET  PORTP,%00100000   ; enable entry mode set
   BCLR  PORTP,%00100000
```

```
   LDD  TCNT           ; load current time
   ADDD  #Add50us      ; time until next operation on LCD can be performed
   STD  TC7
   RTS

;-----------------------------------------------------------------------
; InitLCD: Initilization routines for the LCD Display. Also prepares the
;          LCDBuffer which is where applications place characters to be
;          displayed to the screen.
;
; Assumes: 15ms have passed since Vcc rose to 4.5V
;          Interrupts are turned OFF!!!!
;          A and X are expendable
;          RS --> PB.7, R/W --> PB.6, Enable --> PP.5, D0:D7 --> PA0:PA7
;-----------------------------------------------------------------------
InitLCD
   JSR InitLCDLow             ; low level routines
   JSR LoadVehicleChars
  ; Now prepare the LCDBuffer
  LDX   #LCDBuffer
  LDY   #LCDBufferSize
  JSR   ClearBuffer
  LDX   #LCDBuffer
  STX   LCDHead
  STX   LCDTail

  ; now initialize Timer 7 to be in Compare mode. Give it a value
  ; to trigger an interrupt after every 40us (min. time for LCD).
  BSET  TSCR,%10010000      ; make sure timer is enabled with fast flag clear
  BCLR  TMSK2,%00001111      ; timer ticks every 125nseconds
  BSET  TIOS,%10000000      ; Ouput compare for channel 7
  BCLR  TCTL1,%11000000      ; disconnect Channel 7 from output pin logic
  BCLR  TMSK1,%10000000      ; Leave Timer 7 interrupt off for now

  RTS

InitLCDLow
  BCLR  PORTB,%11000000
  MOVB  #%00110000,PORTA
  BSET  PORTP,%00100000   ; generate enable pulse
  BCLR  PORTP,%00100000

  JSR  DELAY5ms

  BSET  PORTP,%00100000   ; enable the instruction again
  BCLR  PORTP,%00100000
  JSR DELAY5MS             ; need to kill 100us

  BSET  PORTP,%00100000   ; enable the instruction a third time
  BCLR  PORTP,%00100000
  JSR  delay5ms

  MOVB  #%00111100,PORTA          ; function set
  BSET  PORTP,%00100000   ; enable the function set
  BCLR  PORTP,%00100000
  JSR  delay5ms

  ; Turn Display Off
  MOVB #%00001000,PORTA
  BSET  PORTP,%00100000   ; enable the function set
  BCLR  PORTP,%00100000
  JSR delay5ms

  ; Clear Display
  MOVB  #%00000001,PORTA
  BSET  PORTP,%00100000   ; enable clear display
  BCLR  PORTP,%00100000
  JSR   delay5ms

  ; Entry Mode Set --> Increment, shift,
  MOVB  #%00000110,PORTA
  BSET  PORTP,%00100000   ; enable entry mode set
  BCLR  PORTP,%00100000

  MOVB  #%00010100,PORTA   ; cursor/display/shift
  BSET  PORTP,%00100000   ; enable cursor/display/shift
  BCLR  PORTP,%00100000

  JSR   delay5ms
  MOVB  #%00001100,PORTA   ; display on, cursor off, blink off
  BSET  PORTP,%00100000   ; enable cursor/display/shift
```

```
  BCLR  PORTP,%00100000

  RTS
;-----------------------------------------------------------------
; VehicleChars - Write the CGRAMS for the vehicle and for the track.
;-----------------------------------------------------------------
LoadVehicleChars
   JSR    delay5ms
; set up character for vehicle (VH1)
   BCLR    PORTP,%00100000
   BCLR    PortB,%11000000
   MOVB    #$40,PORTA
   BSET    PORTP,%00100000
   BCLR    PORTP,%00100000
   BSET    PORTB,%10000000
   JSR     Delay50usec

   MOVB    #$1F,PORTA            ; top row, back part of vehicle
   JSR Delay50usec
   MOVB    #$15,PORTA
   JSR Delay50usec
   MOVB    #$15,PORTA
   JSR Delay50usec
   MOVB    #$1F,PORTA
   JSR Delay50usec
   MOVB    #$15,PORTA
   JSR Delay50usec
   MOVB    #$15,PORTA
   JSR Delay50usec
   MOVB    #$1F,PORTA            ; back wheel
   JSR Delay50usec
   MOVB    #$00,PORTA
   JSR     Delay50usec

   MOVB   #$02,PORTA             ; front car
   JSR    Delay50usec
   MOVB   #$1F,PORTA
   JSR    Delay50usec
   MOVB   #$01,PORTA
   JSR    Delay50usec
   MOVB   #$01,PORTA
   JSR    Delay50usec
   MOVB   #$01,PORTA
   JSR    Delay50usec
   MOVB   #$1F,PORTA
   JSR    Delay50usec
   MOVB   #$02,PORTA
   JSR    Delay50usec
   MOVB   #$00,PORTA
   JSR    Delay50usec

   ; track character
   MOVB   #$1F,PORTA
   JSR    Delay50usec
   MOVB   #$11,PORTA
   JSR    Delay50usec
   MOVB   #$11,PORTA
   JSR    Delay50usec
   MOVB   #$11,PORTA
   JSR    Delay50usec
   MOVB   #$11,PORTA
   JSR    Delay50usec
   MOVB   #$11,PORTA
   JSR    Delay50usec
   MOVB   #$1F,PORTA
   JSR    Delay50usec
   movb   #$00,porta
   JSR    Delay50usec
   RTS

;-----------------------------------------------------------------
; Delay5ms - Performs a 5ms Delay.
;-----------------------------------------------------------------

DELAY5MS
          LDAA  #$C0
outer     CLRB
inner     DECB        ; DEC and BNE take 3 cycles each or 6 cycles (6*1us)
          BNE inner   ; execute inner loop 256 times (256*6*1us)
          DECA
          BNE outer  ; want to repeat the inner loop 32 times (49.44ms)
```

```
        RTS

Delay50usec   ; also transmits data
  BSET   PORTP,%00100000
  BCLR   PORTP,%00100000
  CLRB
L1
  DECB
  BNE  L1
  RTS

;------------------------------------------------------------------------
; PrtNum:      Takes a 1 byte number, and displays the hex equivalent to the
;              current screen position
; Arguments:   ACCA contains the number to be printed
; Modifies:    ACCA, X
;------------------------------------------------------------------------
PrtNum
  TAB     ; place copy in B
  ANDA  #$F0              ; mask off the low nibble
  LSLA
  LSLA
  LSLA
  LSLA
  ROLA
  TFR   A,X
  LDAA  Nibble2Hex,X
  JSR  PrintChar          ; print high nibble

  ; print out the low nibble
  TBA
  ANDA  #$0F              ; mask of the high nibble
  TFR   A,X
  LDAA  Nibble2Hex,X      ; look up hex value
  JSR  PrintChar
  RTS
;--------------------End of PrtNum--------------------------------------

Nibble2Hex  FCB   '0'
            FCB   '1'
            FCB   '2'
            FCB   '3'
            FCB   '4'
            FCB   '5'
            FCB   '6'
            FCB   '7'
            FCB   '8'
            FCB   '9'
            FCB   'A'
            FCB   'B'
            FCB   'C'
            FCB   'D'
            FCB   'E'
            FCB   'F'
$ENDIF

;--------------------------------------------------------------------------
; Banner.ASM
;
; Block/Zone Controller --> HC12 --> Error Banner Routines
;
; Banner: Part of the second line on the LCD Display is dedicated for
; reporting error messages. Error messages begin printing at the end
; of the banner window and shift every interval time one character
; to the left......
; Current Window Size: 20 characters
; Current Time Till Shift: .1s  (using event handler
;--------------------------------------------------------------------------

$IFNOT INCLUDE_FLASH
InitBanner
UpdateBanner
PrintStrToBanner
PrintCharToBanner
PrintNumToBanner
$ELSEIF

;--------------------------------------------------------------------------
; InitBanner  - Clears banner display on LCD, clears banner buffer and
;               sets tail and head pointers appropriately
; Assumptions:  LCD must already be initialized!! WindowSize < BufferLength
```

```
; Arguments:     None
; Modifies:      LCD, BannerBuffer, WindowHead, BufferTail, BannerChars
;----------------------------------------------------------------------------
InitBanner
  LDX  #BannerBuffer
  LDY  #BannerBufSize
  JSR  ClearBuffer

  LDD  #BannerBuffer
  STD  WindowHead
  ADDD #WindowSize
  STD  BannerTail    ; tail must start WindowSize away from head
  CLR  BannerChars
  CLR  BannerFlag    ; used by event handler
  RTS
;----------------------------------------------------------------------------
; PrintCharToBanner   Takes a character in A and adds it to the banner
; Arguments:          ACCA holds the character to be added.
; Modifies:           BannerBuffer, BannerTail
;----------------------------------------------------------------------------
PrintCharToBanner
  LDX  BannerTail
  STAA ,X
  INX
  ; CPX #BannerBuffer+#BannerBuffSize
  CPX  #BannerBuffer+$28
  BNE  PCBanner_1
  LDX  #BannerBuffer

PCBanner_1
  STX   BannerTail
  INC   BannerChars
  LDAA  BannerChars
  CMPA  #$01
  BNE   PCBanner_fin
  ; otherwise, we need to create the first event to print the first char.
  LDY  #EHRecord
  LDAA #%00000001
  STAA EHFlagMask,Y
  LDD  #BannerInterval
  STD  EHTime,Y
  LDD  #BannerFlag
  STD  EHFlagAddr,Y
  JSR  AddEvent

PCBanner_fin
  RTS

;----------------------------------------------------------------------------
; UpdateBanner:     Updates the display by shifiting window over by 1
; Arguments:        None
; Assumptions:      BannerFlag.0 was set by the event handler
; Modifies:         BannerBuffer, WindowHead, BannerChars,
;----------------------------------------------------------------------------
UpdateBanner
  LDAA  BannerChars
  BEQ   UpdateBanner_fin  ; nothing to update

  ; Set LCD address
  LDAA  #$C0          ; start at 2nd row, first column
  JSR   SetLCDAddr

  LDX   WindowHead     ; X will act as our marker
  INX
  CPX  #BannerBuffer+$28
  BNE  UpdateBanner_A
  LDX  #BannerBuffer
UpdateBanner_A
  LDAB  #WindowSize

UpdateBanner_0
  LDAA  ,X
  BNE   UpdateBanner_0A
  ; if the character was $00, then we need to convert it to a blank
  LDAA  #' '
UpdateBanner_0A     ; print the character
  PSHB
  PSHX
  JSR  PrintChar
  PULX
  PULB
```

```
  INX
  ; CMPX  #BannerBuffer+#BannerBuffSize
  CPX   #BannerBuffer+$28
  BNE  UpdateBanner_1
  LDX  #BannerBuffer       ; wrap around
UpdateBanner_1
  DBNE B,UpdateBanner_0    ; any more characters in Window?

  LDX  WindowHead
  LDAA ,X
  BEQ  UpdateBanner_2
  DEC  BannerChars        ; a character has left the window

UpdateBanner_2
  LDAA  #$00              ; zero out entry
  STAA  ,X
  INX
  ; CPX  #BannerBuffer + #BannerBuffSize
  CPX  #BannerBuffer+$28
  BNE  UpdateBanner_3
  LDX  #BannerBuffer

UpdateBanner_3
  STX WindowHead
  ; now determine if tail is windowSize away from head...make sure it is
  LDAB  #WindowSize
UpdateBanner_3a
  INX
  CPX #BannerBuffer+$28
  BNE UpdateBanner_4
  LDX #BannerBuffer
UpdateBanner_4
  CPX  BannerTail
  BNE UpdateBanner_5

  LDY  BannerTail
  INY   ; increment tail
  CPY #BannerBuffer+$28
  BNE UpdateBanner_4a
  LDY #BannerBuffer
UpdateBanner_4a
  STY BannerTail

UpdateBanner_5
  DBNE  B,UpdateBanner_3a

; if there are still characters to be added, create an event
;  LDAA BannerChars
;  BEQ  UpdateBanner_fin
  LDY  #EHRecord
  LDAA #%00000001
  STAA EHFlagMask,Y
  LDD  #BannerInterval
  STD  EHTime,Y
  LDD  #BannerFlag
  STD  EHFlagAddr,Y
  JSR  AddEvent

UpdateBanner_fin
  MOVB  #$00,BannerFlag
  RTS

;---------------------------------------------------------------------------
; PrintStrToBanner:   Copies a null terminated string into the banner buffer
; Arguments:          X has address of the first byte in the string
; Modifies:           BannerBuffer, BannerTail
;---------------------------------------------------------------------------
PrintStrToBanner

PStrBanner_0
  LDAA  ,X           ; load a character
  BEQ   PStrBanner_1     ; test if null
  PSHX
  JSR   PrintCharToBanner
  PULX
  INX
  BRA   PStrBanner_0

PStrBanner_1
  ; null terminator encountered so we are done
  RTS
```

```
PrtNumToBanner
  TAB     ; place copy in B
  ANDA  #$F0              ; mask off the low nibble
  LSLA
  LSLA
  LSLA
  LSLA
  ROLA
  TFR   A,X
  LDAA  Nibble2Hex,X
  JSR  PrintCharToBanner          ; print high nibble

  ; print out the low nibble
  TBA
  ANDA  #$0F              ; mask of the high nibble
  TFR   A,X
  LDAA  Nibble2Hex,X      ; look up hex value
  JSR  PrintCharToBanner
  RTS
;--------------------End of PrtNum-------------------------------------

$ENDIF

;----------------------------------------------------------------------
; Misc.ASM
;
; Block / Zone Controller --> HC12 --> Miscellaneous Routines
;
; Public Routines: ClearBuffer, PrepareMessage
;----------------------------------------------------------------------

;----------------------------------------------------------------------
; ClearBuffer:  X has address of first byte in buffer. Y has size of buffer.
;               ClearBuffer, clears out the buffer
; Modifies:     the input buffer, X, A and B
;----------------------------------------------------------------------
ClearBuffer
  LDAB #00
ClearBuffer_0
  STAB  ,X
  INX
  DEY
  BNE  ClearBuffer_0

  RTS
;-------------------------End of ClearBuffer--------------------------

;----------------------------------------------------------------------
; PrepareMessage: Copies a message into the destination buffer. In the
;                 process, it adds a header byte and converts the message
;                 to ASCII. It is assumed that the message is of size
;                 MESSAGE_SIZE and the destination buffer is of size
;                 2*message_size+1.
; Arguments:      X has src of message. Y has destination of prepared message.
; Modifies:       A,X,destination buffer,MessSrc,MessDest,MessSize
; Note: MessDest,MessSize,MessSrc are vars re-used by Chksum routine.
;--------------------------------------------------------------------------
PrepareMessage
  ; Again, X has the source, Y has the destination address
  LDAA   #HeaderByte
  STAA   ,Y
  INY

  LDAB   #Message_Size

  ; now convert each byte to ASCII and store it in the destination buffer
PrepareMessage_0
  LDAA   ,X
  LSRA
  LSRA
  LSRA
  LSRA                ; isolate the high nibble and shift down
  STAA   ,Y           ; store high nibble
  INY

  ; now we need to handle the low nibble
  LDAA   ,X
  ANDA   #$0F         ; mask of low nibble
  STAA   ,Y           ; store the low nibble
```

```
INY
INX
DECB

; determine if we are done yet
BNE   PrepareMessage_0
RTS                ; otherwise we are done
```