School Of Engineering

REMOTE AD HOC SENSOR NETWORKS

Li-Wen Yip

Bachelor of Engineering

in

Computer Systems Engineering

Thesis

October 2004

JAMES COOK UNIVERSITY

SCHOOL OF ENGINEERING

EG4010

Computer Systems Engineering

REMOTE AD HOC SENSOR NETWORKS

Li-Wen Yip

Thesis submitted to the School of Engineering in partial fulfilment of the requirements for the degree of

Bachelor of Engineering with Honours

(Computer Systems)

11 November 2005

ABSTRACT

Australia's vast geographical expanses present challenging communications problems for data collection, which often require the use of high power radio links. Many of these scenarios lend themselves to the use of multi-hop ad hoc sensor networks, which can provide an alternative, more flexible and economical solution. However, little if any research has been conducted into developing network protocols suitable for these scenarios, which are characterised by remote and inaccessible locations, sparse topologies, and link ranges of up to 10 kilometres.

The first goal of this project was to develop a dynamic address allocation algorithm to realise the goal of zero configuration, with minimal energy overhead. The second goal was to develop an energy efficient media access control protocol, to maximise the battery life of each node.

The MAC protocol features a synchronised wakeup, which suits the periodic bulk data transfer typical of data logging applications. During inactive periods, a preamble sampling technique provides dramatic power savings whilst maintaining minimal network connectivity. The sampling frequency is dynamically optimised for maximum power savings, based on anticipated variations in local traffic levels.

The address allocation protocol is based on an existing technique, in which each node possesses a block of addresses. These addresses may be autonomously allocated to other nodes without distributed agreement, which results in a protocol with extremely low control overhead and minimal power consumption. This technique has been optimised to further reduce the control overhead for each operation, and avoid unnecessarily flooding the network with search requests.

A microcontroller based hardware platform was constructed, and was used to develop, implement and test the aforementioned protocols. The MAC protocol has been implemented and tested on the microcontroller, whilst the addressing protocol has been implemented in the ns-2 wireless network simulator. The design of the hardware platform has also been improved, improving energy efficiency and performance.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Mr John Wicking. Without his invaluable advice and direction, this thesis project would not have been possible. I would also like to thank the ECE technical staff, whose immeasurable knowledge, experience, and common sense have saved me from many headaches.

To my friends, for sharing in the insane antics which have helped me maintain my humanity and sense of humour throughout this year, and for reminding me that there is more to life than thesis.

Finally, to my family, for supporting and encouraging me through the greater part of my life.

STATEMENT OF ACCESS

SOURCE DECLARATION

I declare that this Thesis is my own work and has not been submitted in any for another Degree or Diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references given.

Li-Wen Yip Date

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 BACKGROUND

Ad hoc networking is currently a very active area of research at many institutions around the world. A subclass of ad hoc networks are *ad hoc sensor networks*, which are typically characterised by long battery life, low traffic rates, low power radios, and dense, well connected topologies. The sensor nodes integrate data collection, data processing and communications functions into a tiny package with long battery life.

There are several existing projects which are at an advanced level of development, including Berkley University Smart Dust, Crossbow Motes, and the TinyOS operating system. There have also been numerous publications presenting new protocols and techniques for media access control, power management, routing, and address allocation. However, little or no research has been specifically conducted into ad hoc sensor networks which feature sparse, minimally connected topologies, and a large geographical distance between nodes. The target application is an ad hoc radio network for environmental monitoring stations on the Great Barrier Reef, which will allow data to be collected and relayed back to the Australian Institute of Marine Sciences at Cape Cleveland. This data is currently collected by high power / long range radio links, which requires expensive and bulky batteries and radios, as well as a costly radio license.

The sparse topology, remote location, and typically harsh environmental conditions present a new set of challenges, including vulnerability to the elements, varying link quality, multi-path propagation, poor SNR and interference, propagation delays, and low maintenance requirements. This scenario requires a different set of techniques and protocols to achieve the desired goals of minimal power consumption, zero configuration and planning, dynamic reconfiguration, and reliable connectivity.

1.2 PREVIOUS WORK

This thesis project was started in 2003 by Nigel Sim [1], who researched a very broad range of topics related to ad hoc networking. The primary focus of this work was developing a simulation environment written in Python. Several basic protocols were developed, implemented, and tested in this simulation environment, operating atop both a microprocessor based physical layer and a software simulated physical layer.

Steven Sloots [2] continued this work in 2004, focussing his efforts on developing an energy efficient routing protocol. A dynamic source routing (DSR) based protocol was developed, and modified to distribute routing information over the entire route to reduce the size of the network header and associated energy overhead. A power aware routing optimisation (PARO) was implemented, which dynamically modifies routes and transmission powers to minimise the energy cost of each route. The hardware, physical, and data link layers were also redesigned based on recommendations made by Sim.

1.3 PROPOSED WORK

The goal of this ongoing project is to develop a complete system which can be deployed on the Great Barrier Reef. With this in mind, several areas of work were identified which were required to move closer to this goal:

1. Development and implementation of a media access control protocol which conserves energy by placing the nodes' radio transceivers in 'sleep' mode.
2. Development and implementation of a dynamic address allocation protocol, with primary goals of zero configuration and minimal power consumption.
3. Improvement of existing hardware and software where opportunities exist to improve efficiency, power consumption, or performance.

Chapter 2

RESEARCH

2.1 POWER CONSERVATION TECHNIQUES

A primary concern in this project is to minimise the power consumption of each node, so as to obtain the longest possible battery life, reducing maintenance requirements. Steven Sloots [2] addressed this problem at the network layer by developing a power aware routing protocol, which dynamically adjusts routes and transmission powers to minimise the total energy cost of each route.

Research has shown that the energy required to monitor the medium for activity is only slightly less than that required for active communications [3]. This is supported by the specifications for the X2010 transceiver currently employed in this project: The nominal supply current is 7 mA in receive mode, 8 mA in transmit mode, and merely 1 μA in sleep mode [4]. Network traffic in ad hoc sensor networks is typically very low, thus significant power savings can be made by placing the nodes' radios in sleep mode. This technique is typically implemented as a media access control protocol or routing extension which operates the nodes' radios on a low duty cycle, to minimise the amount of energy wasted monitoring the channel for activity. The role of such a protocol is to determine which nodes place their radios into sleep mode, and when [5].

Power conservation in ad hoc networks is currently a very active area of research, and as such there are a plethora of existing protocols. As it would be impossible to research every existing protocol, the remainder of this section will discuss various techniques, and some of their more prominent protocol implementations.

2.1.1 On-Demand Wakeup

In on-demand wakeup, nodes may sleep until they are woken up by another node which wishes to communicate with it. This is typically facilitated by a secondary radio device, which can much more efficiently monitor its channel for activity due to reduced complexity requirements. Systems used for the secondary radio include RFID [6], and low power ISM band radios [7]. However, this introduces additional hardware requirements, which add to the size and cost of each node. Of greater concern is that the secondary radio must operate over the same range as the primary radio [5]. This is a particular problem in a long range ad hoc network, where distances between nodes are up to 10km. As such, this technique is not applicable to this project.

2.1.2 Scheduled Rendezvous

A scheduled rendezvous protocol is one in which nodes' sleep schedules are synchronised, such that there is a communication window where all nodes are monitoring the medium. One of the problems with this technique is that adjacent nodes which have non-overlapping sleep schedules may never discover each other [5]. Another problem discussed in [5] is that this technique is not suitable for multi-hop ad hoc networks, as distributed clock synchronisation is difficult to achieve. However, this technique would be implemented at the link layer, thus clock synchronisation is only necessary between adjacent nodes.

2.1.2.1 802.11b IBSS Power Saving Mode

In *802.11b IBSS Power Saving Mode* [8], synchronisation is facilitated by having nodes contend to transmit the synchronisation beacon with backoff. These beacons signal the start of a window used exclusively for handshaking, known as an *ad hoc traffic indication message* (*ATIM) Window*. Data packets are exchanged in the intervals between the ATIM windows. During the ATIM window, nodes which wish to communicate exchange ATIM messages and acknowledgements (Figure 2.1). Once the window has elapsed, nodes which sent or received ATIM messages remain awake to exchange packets, whilst all other nodes return to sleep.

Note that this protocol in its original form is not suited for multi-hop ad hoc networks, as the synchronisation mechanism will not work unless all nodes in the network can hear each other.

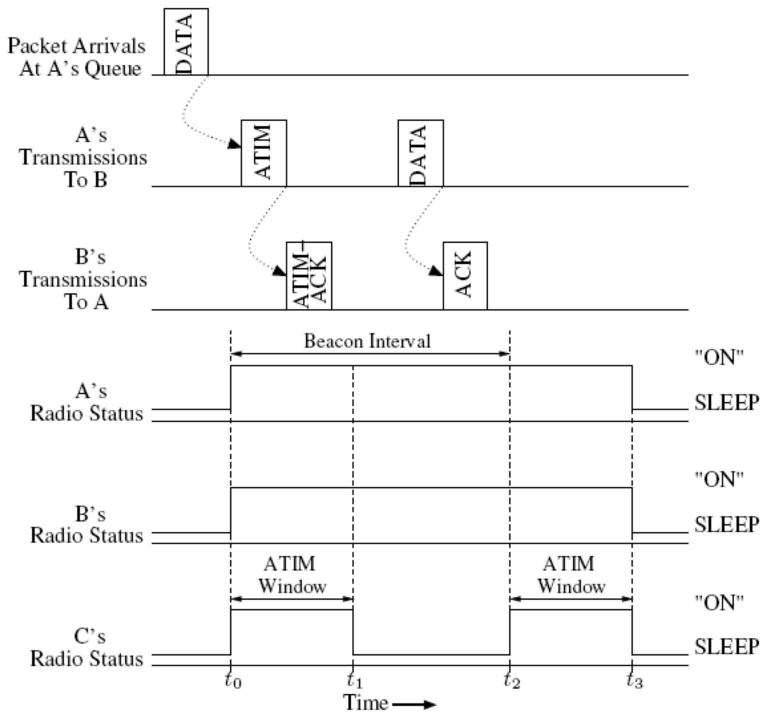


Figure 2.1 - IEEE 802.11 IBSS Power Save Mode [8]

2.1.3 Asynchronous Wakeup

An asynchronous wakeup protocol is one in which nodes' wakeup schedules are not synchronised, instead relying on the schedules of neighbouring nodes overlapping to create a communication window for those two nodes.

2.1.3.1 Contiguous Wakeup Schedule

Feeney [9] has developed an asynchronous protocol where each sleep cycle has a single contiguous "on" period, and a duty cycle of more than 50%. Therefore, neighbouring nodes' sleep schedules are guaranteed to overlap without requiring any synchronisation.

Each wake interval is divided into two ATIM windows and one transfer interval, as indicated in Figure 2.2. At least one of the ATIM windows is guaranteed to fall within the wake interval of each neighbouring node. During the ATIM period, nodes which wish to communicate exchange messages which allow them to estimate their phase difference, and if necessary adjust their phase to increase the amount of overlap to accommodate higher traffic. The ATIM windows are also used for broadcast and multicast traffic, although a broadcast message must be transmitted in both ATIM windows to guarantee that it will be received by all adjacent nodes.

The fundamental limitation of this approach as noted by the author is that each node must be awake for at least fifty percent (50%) of the time.

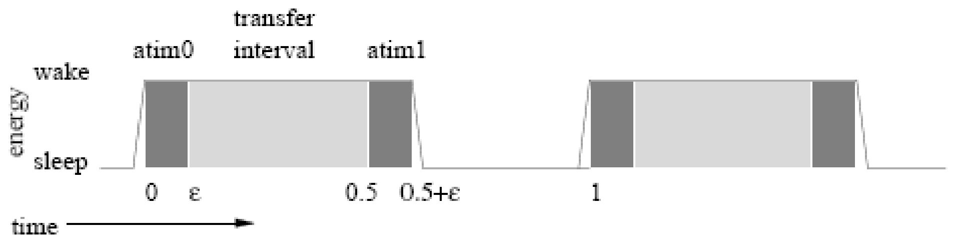


Figure 2.2 - Sleep/Wake Cycles [9]

2.1.3.2 Arbitrary Wakeup Schedule

Zheng et al [5] have developed a protocol in which each node's sleep cycle is divided into a number of slots, each of which may be an "on" slot or a "off" slot. The authors demonstrate that for a cycle of a particular slot length, there exists an optimal *wakeup schedule function* which guarantees at least one slot overlap between any two nodes regardless of phase difference, provided that the slot boundaries are aligned. The authors go on to demonstrate that if the slot boundaries are not aligned, neighbours are guaranteed to detect each other's beacons transmitted at the start of each "on" slot (Figure 2.3). These beacons include information that allows other nodes to calculate the sender's schedule relative to their own, and when their schedules overlap. Outgoing packets are buffered, and transmitted when both sender and receiver are awake.

This protocol overcomes the limitation of [9] that limits the duty cycle to at least 50%. The authors give an example in which the wakeup schedule function has 73 slots, of which only 9 are "on" slots – a duty cycle of 12.3%.

However, like [9], a single message cannot be guaranteed to reach all adjacent nodes. The options for broadcasting would be to broadcast the message immediately after each beacon, or to unicast the message to all known neighbours.

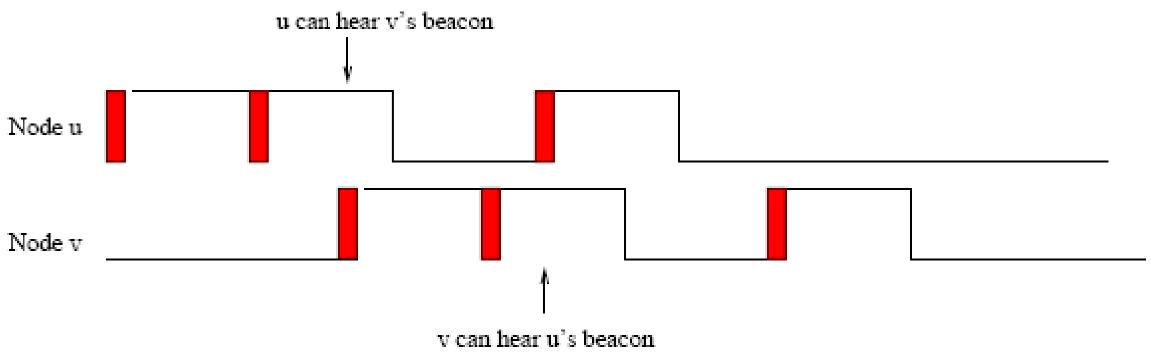


Figure 2.3 - Beacons in the absence of slot boundary alignment [5]

2.1.4 Sleep Based Routing Extensions

Several routing extensions have been developed based on the observation that there are several possible routes for a given source and destination. These protocols operate by allowing nodes which are redundant for routing purposes to conserve more power.

2.1.4.1 SPAN

In SPAN [10], "coordinator" nodes remain awake to buffer and relay packets for other nodes, forming a low-latency backbone for the network. Other nodes are thus able to operate in an asynchronous low power mode. Each node decides individually whether to become a coordinator, based on how much energy it has available, and how much energy will be saved by its neighbours. The role of coordinators is periodically rotated, such that power savings are distributed equally to all nodes.

The salient features of SPAN include improved QoS due to the formation of a backbone; however this is not important in an ad hoc sensor network. The authors also note that this technique is only useful for a minimum node density; thus it is not applicable to sparse or long range ad hoc sensor networks. This also applies to other similar protocols based on the concept of a dominating set of nodes.

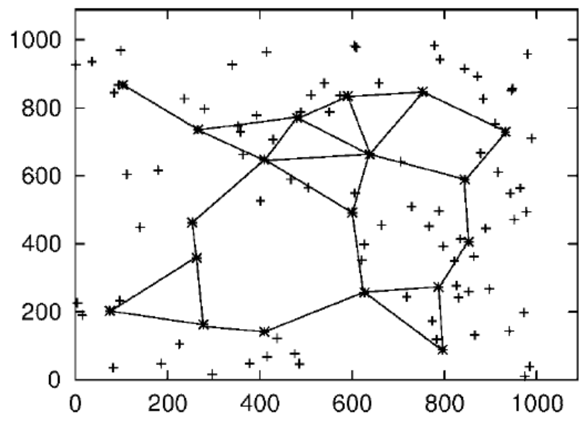


Figure 2.4 - A SPAN network with 100 nodes and 19 coordinators [10]

2.1.4.2 Basic Energy Conservation Algorithm (BECA)

In *BECA* [11], idle nodes asynchronously alternate between listening and sleeping states. If a node either transmits or receives a packet, it transitions to an active state for a predetermined timeout. Whilst a node is active it continuously monitors the medium in anticipation of further network activity. The active timeout is set to be slightly longer than the retry interval for network level requests (e.g. RREQ and AREQ messages), so that involved nodes will remain active until the operation is complete. The sleep interval is set to be an integer multiple k of the network level retry interval, which guarantees that a sleeping node will be woken after $k + 1$ retries.

2.1.4.3 Adaptive Fidelity Energy Conservation Algorithm (AFECA)

AFECA [11] is an extension of BECA which uses information about the local node density to dynamically modify sleep schedule duty cycles. The number of redundant routes for a particular source and destination increases with the node density, thus the duty cycle of nodes in a high density area can be decreased whilst maintaining a constant amount of routing redundancy.

As with SPAN, this technique has limited usefulness in sparse networks.

2.1.5 Preamble Sampling

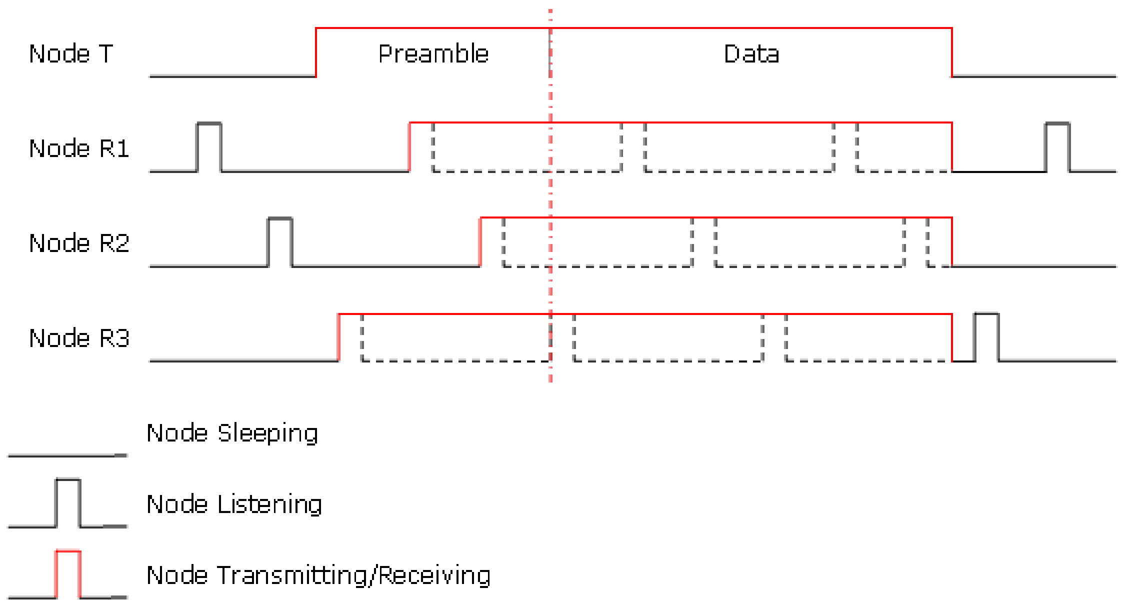


Figure 2.5 - Preamble Sampling

Preamble sampling (Figure 2.5) is an asynchronous technique which can be considered as an on-demand wakeup mechanism, as it uses the packet preamble to wake up sleeping nodes. Nodes periodically wake up to check for a wakeup signal (i.e. preamble). If a wakeup signal is detected, the node remains awake to receive the rest of the packet.

For a node to be woken up regardless of the phase of its wakeup schedule, the preamble must be at least the same length as one sleep cycle [12-15]. Therefore, the power savings made by the receiving node are partially offset by the energy required to transmit and receive the longer preamble.

El-Hoiydi [13] models the performance of preamble sampling when used in conjunction with Aloha [16] and CSMA [8] contention protocols, assuming fixed node densities and packet lengths. However, these results are not very useful, as they are more oriented towards optimising packet latency and throughput

Hill and Culler [15] demonstrate that by quickly measuring the energy in the channel to detect the wakeup signal, the awake interval can be reduced to 50 µs, and the wakeup signal is reduced to a long RF pulse. Conversely, if the start of frame sequence is used as a wakeup signal, the required awake interval is up to two packet lengths, or 108 ms at 10 kbps.

WiseMAC [14], developed by El-Hoiydi and Decotignie is a preamble-sampling based protocol designed for the downlink of infrastructure wireless sensor networks. It is similar to the downlink of 802.11b BSS Power Saving Mode [8], in that the access point learns the sampling schedules of each node and hence is able to minimize the length of its packet preambles. This also has the advantage of minimizing overhearing of packets by nodes which are not the intended recipients.

B-MAC [12], developed by Polastre et al. uses adaptive preamble sampling to reduce the duty cycle for optimal power savings. However, reconfiguration is controlled by protocols and services running at higher layers. For example, this allows network layer information to be used in the decision to reconfigure, such as whether a reply is expected to the packet just sent.

2.1.6 Discussion

To determine the best approach to take toward designing a media access control protocol for this project, it is worthwhile to consider the typical deployment scenario for this type of network.

The nodes will typically be very sparse, with minimal connectivity, although there may be small clusters of nodes situated on reefs. The distance between adjacent nodes may be up to 10km, thus RF amplification will be used for transmission, which will make the energy cost of transmission significantly higher than that of receiving / listening. The traffic rates will be very low, with most activity occurring as a result of network layer operations (e.g. route discovery and address allocation) and data retrieval. Data retrieval will comprise every sensor node routing logged data back to a single *sink* node, which will occur at regular and predetermined intervals. Any MAC layer protocol also has to work in conjunction with the DSR/PARO routing protocol previously implemented by Sloots [2].

The last point is particularly important, as the PARO routing extension relies on nodes being able to overhear packets not intended for them. Therefore, both asynchronous protocols discussed [5, 9] would not be suitable, as they prevent overhearing of messages by non-involved nodes. Additionally, neither protocol supports an efficient broadcasting method. Due to the increased energy cost of transmissions, [9] is slightly better in that it only requires beacons to be transmitted when there is an impending packet transmission, however [5] requires periodic transmission of beacons.

The minimal connectivity of the network would mean that there would be very little routing redundancy, severely limiting the usefulness of both routing extensions discussed [10, 11]. Additionally, both would interfere with the PARO extension previously implemented in this project [2].

This means that the power save protocol should be based on either scheduled rendezvous, or preamble sampling. In fact, a combination of the two techniques may be useful, as follows in the conclusion.

2.1.7 Conclusions

Recall that the majority of network traffic will be caused by either network level operations or transfer of data from sensors nodes to the sink node. Since a reactive routing protocol is used, route requests will only occur when there are impending data packets. Therefore, the only network traffic which will not occur on a predictable schedule is as a result of addressing operations, which mostly occur when the network is initially deployed.

Consider an example, where data is collected from sensor nodes once every six hours, and that data collection can be completed in one minute. Therefore, if a network-wide rendezvous is scheduled for one minute every six hours, the duty cycle of the system is reduced to approximately 0.28%, with no detrimental effects on packet latency or throughput.

Synchronizing the clocks of adjacent nodes to within one second is a non-trivial problem, and the propagation of synchronisation messages can be initiated by the sink node. However, having such a long sleep interval makes it impossible for asynchronous network operations to occur during this period, including address allocation and the initial synchronisation of nodes.

A possible solution is instead of defining global “awake” and “sleep” periods, define periods as “high traffic” and “low traffic” respectively. A preamble sampling technique can be used during the low traffic periods. Communication during these periods will be more energy expensive due to the increased preamble length; however the network is able to retain minimal connectivity. By using adaptive techniques to anticipate network activity similar to those used in [11, 12], the preamble energy overhead during the power save interval could be significantly reduced for network operations such as address requests.

For example, when an initialising node sends an address request, the network layer may instruct the MAC layer to wait the duration of the retry interval before reverting to power save mode, and instruct the MAC layer in neighbouring nodes not to use the extended preamble.

Moreover, the energy sampling technique used in [15] is equally applicable to the X2010 transceiver currently employed in this project. The X2010 data sheet [4] shows that the time from power up required to obtain a valid data is 5 ms, whereas the time required to obtain a valid RSSI reading is only 1ms. These concepts will be further developed in section 3.4 (p. 50).

2.2 DYNAMIC ADDRESS ASSIGNMENT PROTOCOLS

2.2.1 Introduction

Automatic configuration is a desirable feature for any system, as it eliminates the need for a skilled technician to assist with the deployment. One of the important functions of automatic configuration is *dynamic address assignment*, also known as *dynamic address allocation*, or simply *dynamic addressing*. Dynamic addressing refers to the process by which nodes automatically obtain a routable address upon joining a network. This is as opposed to static addressing, whereby a node is manually configured with an address before it joins a network. Dynamic addressing has many advantages over static addressing:

1. Automatic configuration – allows easy deployment.
2. Allows nodes to move freely between networks.
3. Allows addresses to be recycled if the corresponding nodes have left the network.
4. It can cope with the dynamic nature of ad hoc networks.

In traditional TCP/IP networking, dynamic addressing is achieved exclusively with the ubiquitous *Dynamic Host Configuration Protocol* (DHCP), however the lack of static infrastructure in an ad-hoc network precludes its use. The following sections shall go on to investigate existing research into developing dynamic addressing protocols for ad hoc networks.

2.2.1.1 Ad Hoc Network Dynamics

Ad hoc networks are of a very dynamic nature, with constantly changing network conditions, which the dynamic addressing protocol must have mechanisms to deal with. The changes that may occur can be classified into five distinct network events:

1. Network creation – an unconfigured node creates a new network, in the event that it cannot find any existing networks within communication range.
2. Node join – An unconfigured node joins the network. The protocol must ensure that a unique address is allocated to the node.
3. Node part – A node leaves the network. The protocol must ensure the release of the node's address to make it available for reallocation.
4. Network partition – The network is split into two parts either by a broken link or crashed node. The protocol must ensure that each partition is still able to continue operating as an independent network.
5. Network merge – Nodes from two independent networks come within communication distance of each other. The protocol must detect and eliminate duplicate addresses to allow the two networks to be merged.

2.2.1.2 Objectives

Sun and Belding Royer [17] list the following objectives for an ad hoc network dynamic address assignment protocol:

1. Dynamic Address Configuration – Addresses are allocated without manual or static configuration.
2. Unique Address Allocation – Each node is allocated a network-unique address.
3. Robustness – The protocol should have mechanisms to deal with network merges and partitions.
4. Scalability – The protocol should be applicable to networks of varying sizes without an adverse impact on its performance. The relevant performance metrics include timely address allocation, and minimal control overhead during address allocation.

The intended application of this thesis project is the monitoring of remote and inaccessible outdoor locations. Such a network would comprise sparsely distributed nodes, with inter-node hops of up to several kilometres. A likely scenario is that the nodes will be deployed in fixed locations, and network maintenance will consist of replacing nodes whose batteries have expired.

Therefore, the primary goal is to minimise the need for network maintenance by minimising the power consumption of the network. The largest consumer of energy in an ad hoc network is radio communications [18], especially in sparse networks where energy consumption is a function of the distance squared. Pottie and Kaiser [19] illustrate this concept with an example, where transmitting 1 kilobyte of data a distance of 100m uses the same amount of energy as performing 3 million instructions in a 100MIPS/W processor.

There are two ways in which the addressing algorithm can reduce the power consumption of the network:

1. *Minimise the control overhead of network operations*. This may be achieved by reducing the number of packets that must be transmitted to complete a network operation.
2. *Minimise the network addressing overhead*. Network addresses are transmitted in the network header of every packet transmitted, introducing network overhead, and associated energy consumption. This overhead may be significantly reduced by limiting the address space [20].

2.2.1.3 Classification of Address assignment protocols

The address assignment protocols developed to date may be divided into three classifications according to their fundamental concept of operation:

1. *Decentralised* protocols, in which no one node has definitive knowledge of all the addresses in use.
2. *Leader based* protocols, in which a single node keeps a definitive record of every address in use.
3. *Hybrid* protocols, where every node possesses a subset of the total address space, which it can allocate parts of to other nodes.

2.2.1.4 Terminology

Here are definitions of the terms to describe node operations and states in this work.

1. Requesting Node: The node which wishes to join a network
2. Attachment Agent / Initiator node: The node which is responsible for requesting an address on behalf of the Requesting Node.
3. Address request (AREQ): the message broadcast to the network to enquire if the candidate address is unique
4. Address reply (AREP): the message unicast back to the requesting node informing whether the address is unique

2.2.2 Decentralised Protocols

In decentralised protocols, no one node has a definitive list of every address in use throughout the network, so there must be a decentralised method of allocating unique addresses.

Most decentralised protocols rely on some form of *Duplicate Address Detection* (DAD). In the simplest form of DAD, the requesting node randomly selects a tentative network address, and broadcasts an *address request (AREQ)* message to determine if the address is already in use. Nodes which receive the AREQ may issue an *address reply (AREP)* message to indicate that the address is already in use. Such an approach is employed in *IPv4 Link-Local Dynamic Address Allocation (ZeroConf*) [21]. However, there are a number of problems with applying this approach to ad hoc networks, which shall be discussed in the following sections.

2.2.2.1 Routing To Uninitialised Nodes

The ZeroConf protocol is only applicable to link-local networks, i.e. all the nodes in the network are totally connected by physical or logical links, thus AREP messages can be reliably delivered to the requesting node. However, in a multi-hop ad-hoc network, if the tentative address is not unique, AREP messages may not be correctly routed.

Perkins et al. [22] propose a scheme which attempts to reduce the probability of this event. In addition to selecting a candidate address, requesting nodes select a temporary source address from an address range reserved for this process, thus guaranteeing that the source address will not be in use by any existing nodes. However, there still exists the small possibility that two requesting nodes may simultaneously select the same temporary source address, in which case AREP messages may not be correctly routed [23, 24].

The protocols proposed in [20, 24] solve this problem by using an *attachment agent* to perform the duplicate address detection on behalf of the requesting node. The attachment agent is already has a unique address, thus the network can reliably route AREP messages to it. The attachment agent and the requesting node can reliably exchange messages, as they have share a direct link.

2.2.2.2 Limited Address Space

It can be observed that for simple DAD to perform efficiently, the ratio between the size of the available address space and the size of the network must be quite large. In [21, 22] the 169.254/16 address range (approx. 65,000 addresses) is reserved for this purpose, giving a host joining a network comprising 1300 hosts a 98% chance of selecting an unused address on its first attempt [21]. However, bandwidth is a very limited resource in ad hoc sensor networks [25], thus it is desirable to conserve bandwidth by reducing the address length and thus the addressing overhead.

In MANETconf [24], each node maintains a record of the addresses it knows to be in use, allowing the attachment agent to select an address which has a high probability of being unused. All nodes which receive the AREQ from the attachment agent, must acknowledge it; this enables the attachment agent to release addresses which it does not receive replies from, providing a mechanism to detect crashed nodes. Together, these features make MANETconf applicable to networks with a limited address space. In fact, it restricts usage to networks with a limited address space, as every node must have sufficient memory to store the address of every node in the network.

Boleng [20] proposes a variable length addressing scheme in which the address space grows with the size of the network. All nodes maintain two addressing parameters: ADDR_LEN, the current address length in use, and HIGH_ADDR, the highest address in use. This allows addresses to be allocated sequentially instead of randomly. However, the maintenance of these parameters adds overhead to the network, especially when the address space is incremented. Furthermore, the protocol does not provide a mechanism to recover addresses from crashed nodes.

2.2.2.3 Unbounded Delays

Nesargi and Prakash [24] discuss the importance of selecting timeout periods for DAD attempts. The timeout period is a compromise between timely address allocation, and the chance of failing to detect duplicate addresses too far away from the requesting node. The authors state that to ensure that all duplicate addresses are detected, the timeout period must be a function of the diameter of the network, which in the worst case can be $O(n)$, where n is the number of nodes.

Furthermore, Vaidya [26] presents the following theorem:

> *"Strong DAD cannot be guaranteed if message delays between at least one pair of nodes in the network are unbounded."*

In a sparse outdoor ad hoc network, partitions due to weather conditions or node failures may result unbounded message delays, thus causing DAD to fail. Considering that such partitions are usually temporary, the resulting message delays could be considered bounded, but the timeouts required to ensure success are impractically long.

MANETconf addresses this issue by maintaining state information, i.e. each node has a list of all the addresses it knows to be in use. If the initiator node receives affirmative AREP messages from all the addresses in its list, it can be fairly certain that the candidate address is not in use without waiting for the timeout to expire.

2.2.2.4 Control Overhead

As previously mentioned, bandwidth is a very limited resource in ad hoc sensor networks [25]. Every DAD based protocol discussed so far have one common disadvantage – each address allocation operation requires the network to be flooded with an AREQ message, thus a non-trivial amount of bandwidth is consumed each time a node joins. In particular, MANETconf [24] requires all nodes to unicast AREP messages in reply to a AREQ message. The amount of bandwidth consumed is related to the size of the network, thus the scalability of these approaches is generally poor. This is an intrinsic disadvantage of distributed DAD based approaches, and there is little that can be done to mitigate it without introducing some sort of centralisation. Sun and Belding-Royer propose such a scheme which combines DAD and centralisation [27], which shall be discussed in the following section.

2.2.3 Leader-based Protocols

Leader-based protocols employ a single node to maintain a definitive list of addresses in use throughout the network. This node may perform two functions:

1. Verify the uniqueness of a node's tentative address.
2. Directly allocate unique addresses to nodes.

The classic example of a leader-based protocol is the *dynamic host configuration protocol* (DHCP) [28], a client-server based protocol for automatic configuration of clients in TCP/IP networks. The protocol includes a dynamic address allocation mechanism, in which the DHCP server allocates addresses to clients on a renewable time-limited lease. If the client allows the lease to expire, the address will be made available for allocation to other clients, allowing addresses to be reused without being explicitly released by the client.

However, ad hoc networks are by definition devoid of static infrastructure, precluding the use of DHCP in its traditional form. The leader-based protocols researched [27, 29] all elect a leader from amongst the nodes which form the network. These protocols are discussed further in the following sections.

2.2.3.1 Dynamic Address Configuration Protocol (DACP)

Sun and Belding-Royer [27] propose a protocol based on [22] which combines DAD and a centralised *Address Authority (AA)*. (This protocol is referred to by the same authors in [17] as *DACP*). The first and second nodes to join the network respectively assume the roles of *Primary Address Authority (PAA)*, and the redundant *Backup Address Authority (BAA)*.

The PAA periodically broadcasts a beacon message, advertising its presence to the network. A node may detect a network partition if it stops receiving beacon messages from the PAA, in which case a new PAA is elected for the partitioned network. If the BAA is inside the partitioned network, it may automatically take over as the PAA. Conversely, a node may detect a network merge if it receives beacon messages from two different PAAs, in which case the two PAAs will exchange address lists and eliminate and reallocate duplicate addresses.

A requesting node selects a candidate address and a temporary sources address, and performs DAD in an identical manner to that described in [22]. If a duplicate address exists, either the node possessing the duplicated address or the PAA may reply with an AREP message, which allows for faster detection of duplicate addresses. However, there is no mechanism to speed up confirmation that the candidate address is unique.

Once the requesting node is satisfied that its candidate address is unique, it registers its address and requested lease duration with the PAA. The lease mechanism is identical to that used in DHCP [28], allowing an address to be released if the node does not renew its lease, whilst allowing temporarily disconnected nodes to retain their addresses.

The authors also argue that because routing information is accumulated by the AREQ messages, the AREP message can be routed back to the correct recipient even if two requesting nodes are concurrently using the same temporary source address. However, they do not consider the case where the two requesting nodes share an identical route, which has a reasonable probability in very small networks.

2.2.3.2 Optimised DACP (ODACP)

Sun and Belding-Royer propose in [17] an optimised version of their DACP protocol sans DAD and the associated overhead, resulting in a pure leader-based scheme. Instead of flooding the network with AREQ messages, the requesting node unicasts the AREQ message to the PAA.

2.2.3.3 Dynamic Address Allocation Protocol (DAAP)

Patchipulusu [30] proposes a scheme in which the last node to join the network takes on the responsibility of being the leader. Addresses are assigned sequentially, and all nodes keep a record of the highest address in use (the address of the leader). All nodes periodically broadcast "hello" messages containing a network identifier, which facilitates the detection of network merges and partitions.

2.2.4 Hybrid Protocols

In a hybrid protocol, every node is a leader with the authority to allocate addresses from a subset of the address space.

2.2.4.1 Nigel Sim's Solution

Nigel Sim [1] proposes a scheme based on a binary tree, where each node has the authority to assign two child addresses, which consist of a single bit appended to the parent's address. This concept has the advantage of variable address length. However, this scheme does not consider the case where a requesting node is unable to establish a direct link to a parent which has not already assigned both its child addresses, nor does it consider the topology of the network. Consider the best case scenario, where the topology of the network is a complete binary tree. The required address length would be $log_2 n$, i.e. there would be few wasted addresses. However, consider the worst case scenario, where the network has a linear topology, which would require *n bit* long addresses, i.e. there would be $2^n - n$ wasted addresses. This scheme is clearly only suitable for dense networks where the topology remains static for the lifetime of the network. However, [23, 31] (which are discussed below) make use of this concept of each node having authority over a subset of the address space resulting in low maintenance addressing schemes with little overhead.

2.2.4.2 Address Pool Protocols

Both Tayal and Patnaik [23] and Hu and Li [31] have presented protocols based on the address pool concept. Instead of a single address, each node is allocated a set of addresses, the first address of which it uses for itself. Upon request, it will relinquish part of its address pool to a requesting node. This operation does not require multi-hop routing or distributed agreement, thus mitigating the disadvantages common to other protocols such as flooding messages, non-trivial timeouts and delays, protocol modifications, overhead of periodical maintenance messages, and major address leaks [31]. Another major advantage is that partitioning and subsequent remerging of a network need not incur any control overhead. The two hybrid protocols reviewed here differ primarily in their solution to the address depletion problem, where a joining node is not in range of an existing node which possesses an available address.

2.2.4.3 ZAL: Zero-Maintenance Address Allocation

ZAL [31] takes a proactive approach to the address depletion problem, using the *ZAL Distribution Equalization* (*ZAL/DE*) algorithm. *ZAL/DE* promotes even distribution of addresses throughout the network by attempting to ensure there are a predetermined number of available addresses within transmission range of any node in the network. Nodes which possess surplus addresses will attempt to distribute them to other nodes. However, this process will only occur when absolutely necessary to maintain the minimum number of available addresses in an area, thus there is no control overhead in a stable network.

In the event that a requesting node cannot obtain an address from any of its immediate neighbours, it randomly selects a temporary address from a reserved address range until it can obtain a permanent unique address. However, this assumes the node is mobile, and will eventually come into direct contact with a node which has an available address. This obviously will not work for a sensor network in which all the nodes are static. Furthermore, this solution does not provide a mechanism to reclaim leaked addresses.

2.2.4.4 The Tayal and Patnaik solution

The solution proposed by Tayal and Patnaik [23] takes a reactive approach to the address depletion problem. When a requesting node cannot contact a node that has available addresses, the attachment agent floods the network with a request for an available address pool. This process also provides a mechanism to reclaim leaked addresses; any nodes which no not respond to the address request within a bounded timeframe can have their address pools reclaimed. Any reclaimed address pools are always repossessed by a node which contains an adjacent address block. This eases the memory requirements on the nodes, as the address pools will always be a contiguous block, which can be defined by its start and end addresses.

2.2.5 Discussion

To determine the best approach to the address allocation problem for this project, it is worthwhile to consider the most likely deployment situation for this type of network.

The network will most likely be deployed outdoors, with a distance between the nodes of up to 10km. Most of the nodes will be deployed when the network is initially set up, and additional nodes may be added or removed throughout the life of the network. Nodes may crash as their batteries fail, or due to equipment failure. Weather conditions may temporarily affect the link quality, potentially causing temporary network partitions and subsequent remerges. Other devices operating in the same ISM band may cause interference, which may interfere with the operation of one or more nodes, potentially causing temporary network partitions.

A key point here is that most network partitions will be temporary. Whilst the network partitions must be able to continue operating until they are remerged, it is a potential waste of energy to establish new network infrastructure for each partition if they are inevitably, going to be remerged. The hybrid protocol concept is very suitable in this respect, as there is no control overhead for a network partition and subsequent remerging.

The salient feature of the Tayal and Patnaik solution [23] is its simplicity, and that all operations remain local unless address depletion occurs. The principle of contiguous address spaces is very desirable, as it eases memory requirements, and reduces the bandwidth required for nodes to exchange address spaces.

The other impressive feature of ZAL [31] is that it proactively ensures even distribution of addresses throughout the network. The authors claim that the network requires zero maintenance once it has reached a stable state, however reaching a stable state comes with a non-trivial energy cost, particularly in a sparse network.

However, ZAL does have some passive features which promote even distribution of addresses. For example, a requesting node does not accept the first address offer it receives. Instead, it waits for all replies to arrive, then accepts the largest offer. This increases the time it takes to obtain an address, but it does not make the operation any more energy expensive.

A potential area in which the Tayal and Patnaik protocol can be improved is the method used to search the network for available addresses. The authors specify that if a node receives an address request but does not possess any available addresses, it immediately floods the network with an address search message. Therefore, even if other nodes which received the address request are able to provide addresses to the requesting node, this message will be flooded to the network. This method could be improved to only propagate the search message when it is absolutely necessary, thereby conserving energy.

2.2.6 Conclusions

It is clear that the address allocation protocol for this project should be based on the hybrid protocols [23, 31], primarily because of their ability to deal with temporary network partitions, and low control overhead. The protocol to be developed will attempt to incorporate the salient features of both protocols, as well as adaptations to conserve energy in a sparse long range network. The operation of this protocol is discussed further in section 3.5 (p. 60).

Chapter 3

DEVELOPMENT

3.1 Hardware Architecture

The hardware platform previously designed by Sloots [2] worked successfully, and as such will only be subject to minor improvements, the main goal of which will be decreasing power consumption. A block diagram of the 2004 hardware platform is shown below in Figure 3.1.

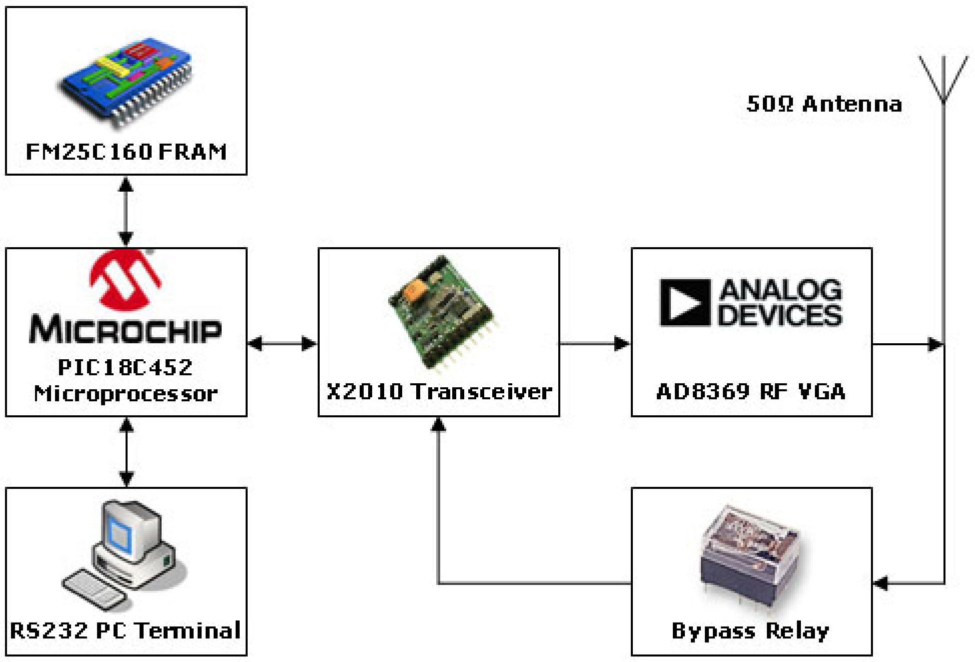


Figure 3.1 - 2004 Hardware Block Diagram

3.1.1 VGA Bypass Problem

As can be seen in Figure 3.1, an X2010 transceiver module is used in conjunction with an AD8369 Variable gain amplifier (VGA) to provide programmable output power, as required to implement the power aware routing extension developed in 2004. The transceiver only has one antenna connection; hence a relay provides a return signal path around the VGA to allow reception. This was an improvised solution, and as a result is far from ideal:

1. The relay contact closing time causes an additional delay when switching between receive and transmit modes. This delay was found to be between 5 and 15 milliseconds [2], which exceeds the X2010 transceiver's inherent switching delay of 5 milliseconds.
2. The current required to reliably latch the relay contacts is typically 550mA, which the microprocessor must drive via a switching transistor [2]. More importantly, this incurs a power consumption of 2.75W[1], which is clearly unacceptable in an application where power conservation is an important criterion.

Several solutions to this problem were considered. One option was to retain the same configuration, but use a low power relay. The lowest power relay available through Farnell InOne is the NEC EF2 ultra low power relay [32], which has a power consumption of 50mW, and an operation time of 4ms. However the device is expensive, costing $14.51 (AUD) for a single unit.

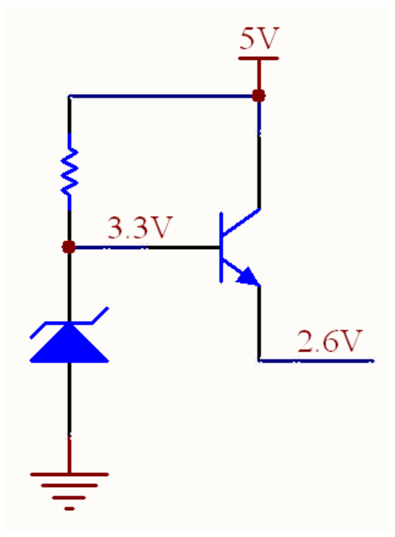
A second option was to eliminate the bypass relay by using separate transmitter and receiver modules, which allows for two separate antennae. However, this would increase the size and cost of each node.

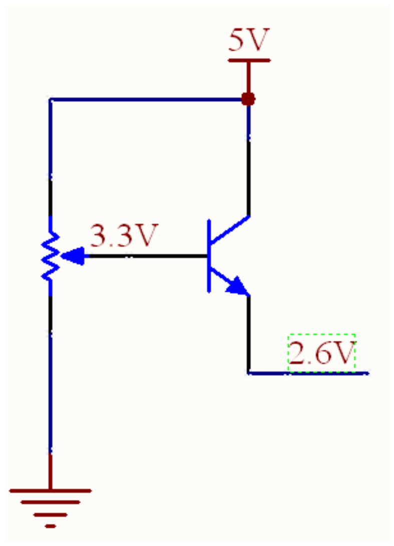
[1] Power consumption is based on a power supply of 5 Volts DC.

A third option was to use a transceiver module with programmable output power. The LPRS ER400TRS "Easy Radio" transceiver module [33] features programmable output power, frequency, and data rate, and handles low level functions such as Manchester encoding and buffering. This would eliminate the bypass relay, the AD8369 VGA, and the software Manchester encoder/decoder. Unfortunately, the maximum output power for the device is only 10mW. To maximise communications range, the nodes should be able to transmit at the maximum allowable power for licence free transmission, which for the Australian 433 MHz band is 25 mW. If this device were available with a higher transmission power, it would be the ideal solution.

The chosen solution was to replace the relay with a solid state CMOS switch, which is far smaller and consumes an insignificant amount of power. General purpose CMOS switches are unsuitable for switching high frequency signals due to impedance problems. However, Analog Devices produces a range of wideband analogue switches specially designed for RF applications (ADG918/919, [34]). These devices provide low insertion loss in the 'ON' state and high port separation in the 'OFF' state at frequencies up to 1 GHz, and have very low power consumption (~1uA). Unfortunately, the devices only operate over 1.65V – 2.75V, and will hence require a separate power supply.

A suitable power supply for a low current device is an emitter-follower voltage source. Sources of this type would typically use a Zener diode to provide a reference voltage (Figure 3.2a). However, to operate a Zener diode in its Zener region requires approximately 10mA of current, which is not acceptable overhead for this application. The alternative is to use a resistor voltage divider to provide the reference voltage (Figure 3.2b). Achieving a precise ratio with discrete resistors is difficult at best, thus a trimmer potentiometer will be used in the prototype allowing the voltage to be tuned to the correct level.





a) Zener diode as reference b) Resistor voltage divider as reference

Figure 3.2 - Emitter Follower Voltage Sources

3.1.2 New Hardware Requirements

3.1.2.1 Real Time Clock

A real time clock is to be included in this hardware design for several reasons:

1. To enable accurate scheduling of data sampling.
2. To allow data samples to be time stamped.
3. To allow accurate timing of sleep-wakeup schedules.

The RTC device should feature programmable alarms that are capable of generating hardware interrupts for the PIC microcontroller. A suitable device was found in the Maxim-Dallas DS1305 Serial Real Time Clock [35], which features two programmable alarms with separate interrupt lines, and a SPI interface.

3.2 Software Architecture

3.2.1 The OSI Model

The *Open Systems Interconnect Model* (OSI Model) divides the functions of a communication system into 7 layers of abstraction. This model allows a communication system to be designed as a 'stack' of protocols, each of which implement the functionality of a layer.

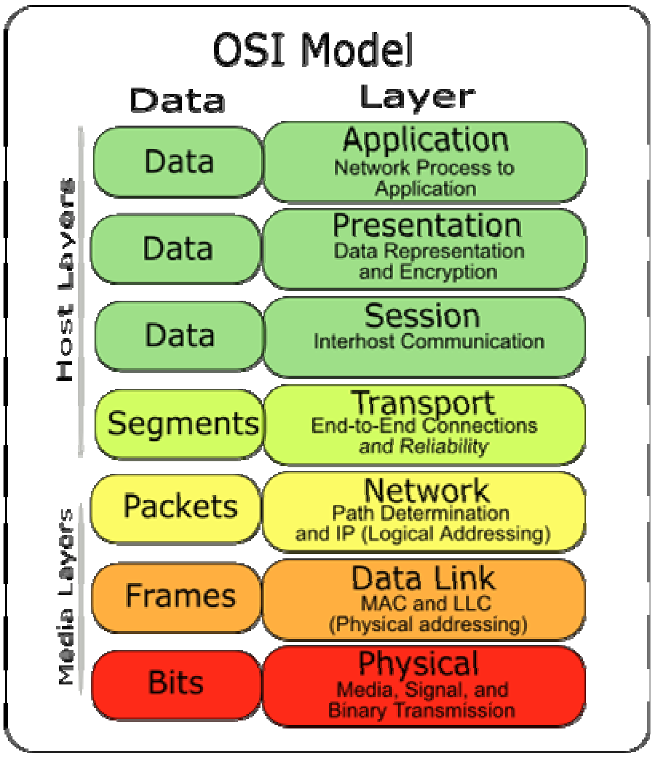

OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical addressing)
Bits
Physical
Media, Signal, and Binary Transmission

Figure 3.3 - The 7 Layers of the OSI Model [36]

The key concept of the OSI Model is encapsulation, which defines the way in which the layers interact with each other. Each layer is only dependent on the functionality of the layer immediately below it, and only provides functionality to the layer immediately above it.

The protocol layers in different devices communicate with each other using a *Protocol Data Unit* (PDU), which comprises a header containing information specific to that protocol layer, and a data section which encapsulates the PDU of the next higher layer. The header is removed before the PDU is passed up to the next higher layer, so a particular layer will only ever see its own PDU. This allows a communications system to encompass many devices using different communications protocols and transmission media.

This thesis project is only concerned with developing a network access module, which comprises the first three layers of the OSI Model. The descriptions of these layers according to Wikipedia [36] are given below.

3.2.1.1 Physical Layer

The physical layer defines and implements the mechanical and electrical interface between devices. Its primary function is to convert the data as it is represented within the device into a signal suitable for the communication medium, i.e. an electrical, light, or radio signal, and to convert it back at the receiver.

Important aspects of the physical layer for a radio based system include:

1. Transceiver hardware.
2. Transmission frequency.
3. Baud rate.
4. Encoding at the transmitter, and decoding and synchronisation at the receiver.
5. Establishment and termination of transmissions.

3.2.1.2 Data Link Layer

The data link layer ensures the reliable transmission of frames between two devices which share a point-to-point link. The data link layer is further divided into two sub-layers: the Media Access Control (MAC) layer, and the Logical Link Control (LLC) layer.

3.2.1.2.1 Media Access Control Layer

The MAC layer converts the raw bit stream received by the physical layer into frames, which are the data link layer PDU. This includes:

1. Delimiting the frames at the transmitter, and extracting frames from the bit stream at the receiver.
2. MAC Addresses, filtering frames not intended for the device.
3. Calculation, appending, and verification of the CRC.

4. Controlling access to the media i.e. Carrier Sense Multiple Access (CSMA).

3.2.1.2.2 Logical Link Control Layer

The LLC layer is responsible for converting between network layer packets and data link layer frames. This includes:

1. Fragmenting network layer packets which are too large into smaller MAC layer frames, and reassembling them at the receiver. (Type 1, 2, 3)
2. Ensuring reliable point-to-point delivery of frames. (Type 2, 3)
3. Reassembling frames in the correct order. (Type 2)

The LLC layer is not implemented in this project.

3.2.1.3 Network Layer

The network layer is responsible for conveying a message between any two nodes in the network. This includes:

1. Global network layer addressing.
2. Dynamic address allocation.
3. Resolving network layer addresses to MAC layer addresses.
4. Routing.

3.2.2 Relocatable Code

Assembly language programs for Microchip PIC microprocessors are traditionally compiled as absolute code, meaning that the executable code is generated directly from a single monolithic source file, and program and data memory addresses are assigned at compile time.

The software developed by Steven Sloots was written in this manner, with the source file containing over three 3000 lines of code, 120 cryptic variables, and 350 code labels and constants. This made the code very difficult to understand and work with, and was exacerbated by a lack of comprehensive commenting and documentation.

The introduction of MPASM v2.00 and MPLINK v1.00 [37] allows the generation of *relocatable code*, meaning that several source files are compiled into individual object modules. These object modules are then combined by a linker to generate executable code (Figure 3.4). This has several advantages over absolute code generation:

1. The program can be split into several independent and reusable code modules.
2. Each module has a separate namespace for program and data memory labels.
3. Only the affected modules need to be recompiled when changes to the program are made.

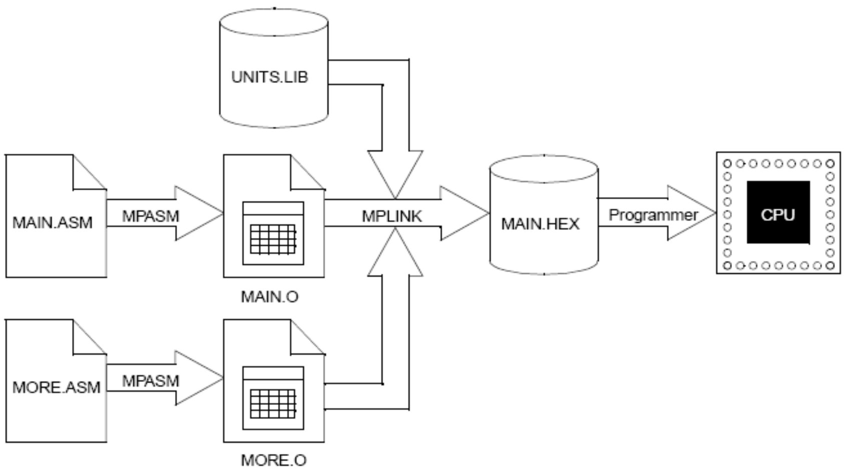


Figure 3.4 - Generating Executable Code from Object Modules [37]

The software will be redeveloped for this project using relocatable code, with these specific objectives:

1. Make the code modular, reusable, and easy to understand.
2. Limit the scope of each code module to a single protocol, so that each protocol can be worked on independently.
3. Create a well defined interface for each code module, to promote encapsulation and reusability.
4. Consistent standard of comprehensive commenting, especially for each subroutine / code block (not just line-by-line comments).
5. Create comprehensive documentation in the form of module interaction diagrams and flow charts.
6. Allow the code to be easily extended in future projects.

3.3 MANCHESTER ENCODING AND DECODING

As stated earlier, the X2010 radio transceiver is AC coupled on its data pins, thus the data must be Manchester encoded to ensure that there is no DC component in the transmitted bit stream. This introduces the need for the following functions:

1. Manchester encoding.
2. Manchester decoding.
3. Clock synchronisation at the receiver.

3.3.1 Introduction

Most digital coding schemes involve multiplying (XOR) the data sequence with a code sequence. In many schemes, the length of the code sequence is equal to the length of one data bit, so that a high data bit is encoded a single iteration of the code sequence, and a low data bit is encoded as an inverted iteration of the code sequence. The encoded bits are referred to as chips. The effective data rate, or chip rate is given by:

$$\textit{Chip rate} = \textit{Data rate} * \textit{Code sequence length} \qquad (3.1)$$

Manchester encoding uses the code '10' thus a data bit 1 is encoded to 10, and a data bit 0 is encoded to 01. The code is two bits long, so the chip rate will be twice the data rate.

An equally valid alternative definition is based on transitions, with a data bit 1 being encoded to a rising edge, and a data bit 0 being encoded to a falling edge.

3.3.2 Manchester Encoding

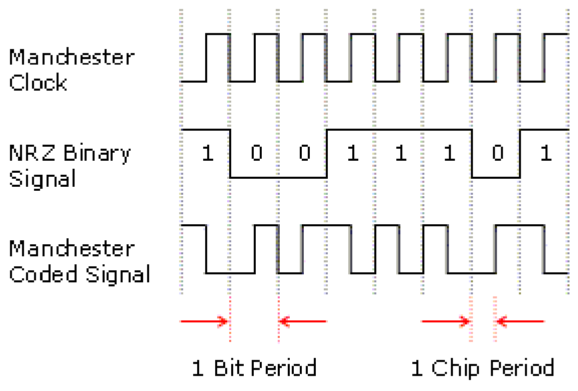


Figure 3.5 – Manchester Encoding

The hardware implementation of Manchester encoding is to multiply the raw data stream with a 50% duty cycle clock running at the data rate (Figure 3.5), which Sloots [2] successfully converted to a software implementation. This software is executed under a compare interrupt running at the chip rate, allowing the output to be manipulated once at the beginning of every chip period.

However, there is a minor flaw in this software - interrupt latency and the time taken to calculate the new output value may vary, causing the encoded clock signal to drift in phase (Figure 3.6).

No problems are anticipated unless the clock rate is increased relative to the processor speed, however there is a simple solution: The value of each chip can be calculated one chip period in advance, therefore as long as the calculations take less than one chip period, the clock will remain in perfect synchronization. This is easily implemented by programming a compare output pin to change at the instant the interrupt occurs.

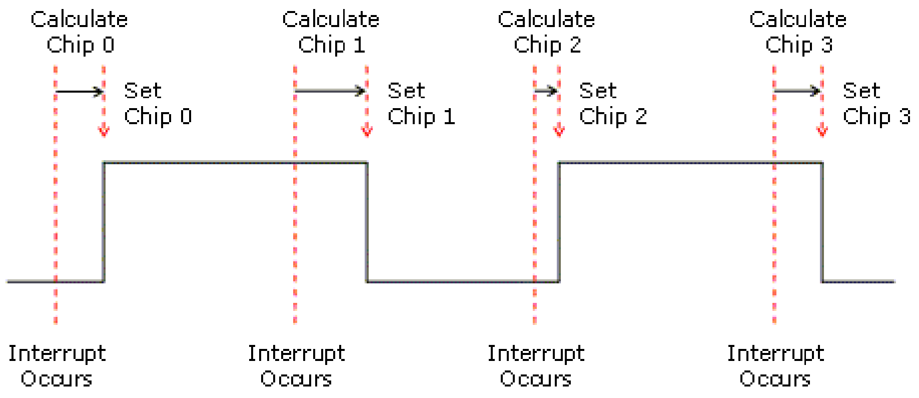


Figure 3.6 - Manchester Clock Drift

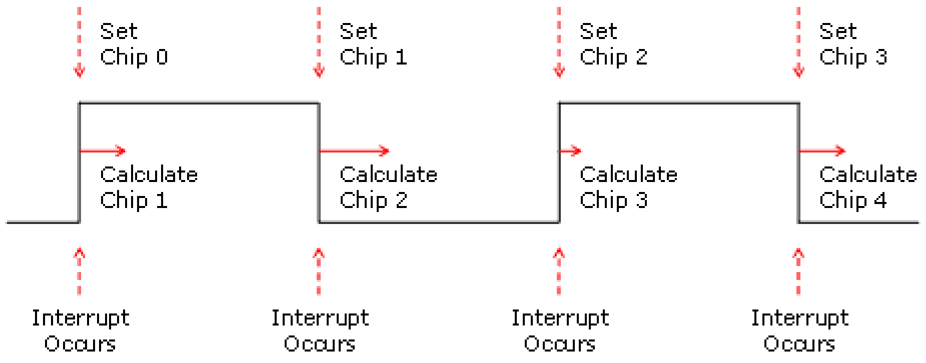


Figure 3.7 - No Clock Drift

3.3.3 Manchester Decoding and Synchronisation

Sloots [2] previously developed a software decoding algorithm based on asynchronous serial transmission. Both the start and stop bits are high (1→0), so that there is always a rising edge at the start of the byte. The decoder detects this edge via external interrupt, waits 1/2 a chip time, then takes 20 samples, 1 chip time apart. Framing errors are detected by verifying both the start and stop bits, and verifying that the second chip of each bit is inverted. However, there are some major flaws with this algorithm, as described below.

Firstly, the receive data pin of the X2010 transceiver does not have an mark state when a signal is not present, rather it outputs a stream of randomly spaced pulses. (Figure 3.8). The aforementioned decoding algorithm will detect one of these as a start bit after a carrier is detected, but before the incoming signal stabilises, which will likely cause a framing error. This means that there must be a mark state for at least one byte period after the incoming signal has stabilised, to allow the receiver to correctly detect the first start bit. However, there is no way to achieve this; when a DC signal is fed into the transmitter, the receiver still outputs a stream of randomly spaced pulses, albeit less frequently.

Secondly, because the samples are not processed until all 20 have been taken, the decoder has no way of detecting a framing error until one byte period has elapsed. This poses a problem with a Manchester encoded signal: during the synchronisation header, the clock embedded in the signal may trigger another sampling routine as soon as the previous one has finished, meaning the decoder may never synchronise to the start of the frame (Figure 3.9).

Thirdly, the start and stop bits on each byte are currently serving to synchronise the receiver to the start of each frame, which is essentially using an asynchronous synchronisation technique to decode a synchronous transmission. Much more precise synchronisation can be achieved by extracting the clock from the data signal, hence these bits unnecessarily incur a 25% bandwidth overhead.

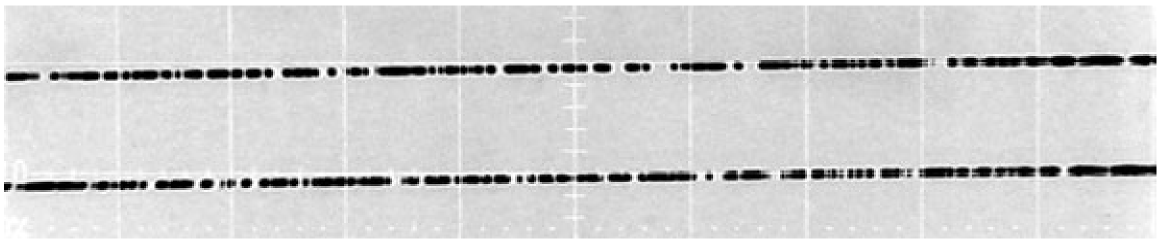


Figure 3.8 – RxD with no signal present; 5V/div; 1ms/div

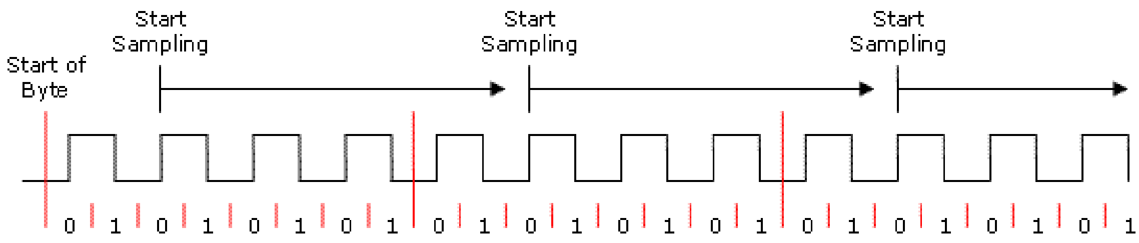
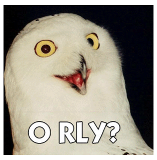


Figure 3.9 - Repeated framing errors

It is clear that the decoding algorithm must be improved so that synchronisation can be achieved more quickly, reliably, and with less overhead. The problem of decoding and synchronisation exists at two levels:

1. Clock synchronisation, i.e. synchronising to the start of each data bit. This is the responsibility of the physical layer.
2. Synchronising to the start of each frame (packet). This is the responsibility of the data link layer.



3.3.3.1 Clock Synchronisation

Sim [1] discussed three methods of clock synchronisation for a Manchester encoded data stream:

1. Over-sampling.
2. Edge Timing.
3. Synchronous logic, combining edge timing and sampling.

Sim [1] developed an over-sampling method, which takes 4 samples within each chip period. Clock synchronisation is ensured by waiting for the edge in the middle of each bit. However, the algorithm requires a high processor speed, which is not conducive to reducing power consumption.

The edge timing method decodes the data bits by measuring the time between edges, however it was discounted as it is overly complex.

Mr Sim also discussed a synchronous logic solution, which combines edge detection and sampling. The process is shown in Figure 3.10:

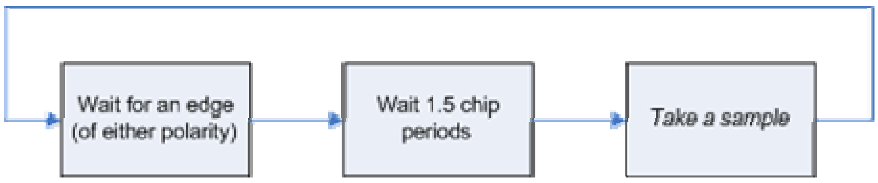


Figure 3.10 - Synchronous logic decoding process

This solution was originally discounted as a possible solution, due to additional external hardware requirements. However, it is in fact is a simplified version of Sim's over-sampling method [1], and is easily implemented in software using a capture/compare module.

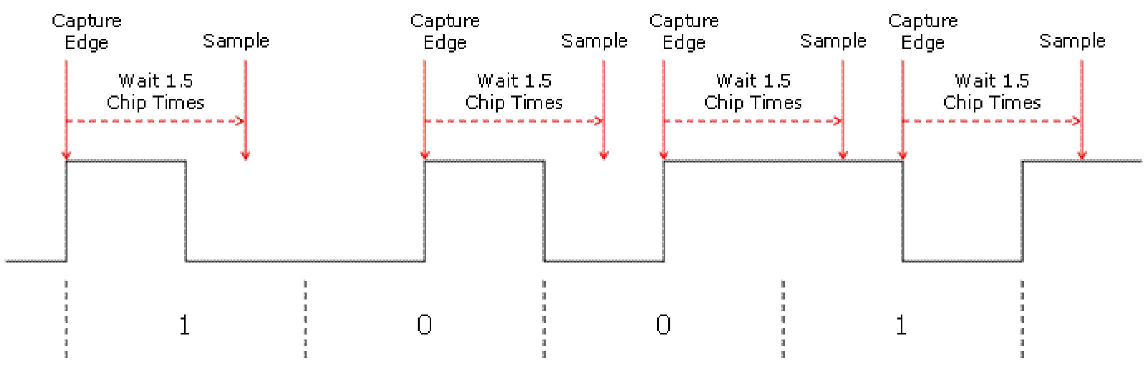


Figure 3.11 - Edge capture / sampling timing diagram

As can be seen in Figure 3.11, even if the clock is initially incorrectly detected, it will be correctly acquired as soon as the data bit changes from 1 to 0, or 0 to 1. This supports the assertion that the preamble would have to consist of alternating data bits, i.e. 0xAA or 0x55 [1].

A disadvantage of this method is that the second chip of each bit is not verified. However, this should not pose a problem; the effect of any interference would be to disrupt the clock rather than corrupt the data, and any data errors that do occur will be detected by the frame's CRC.

3.3.3.2 Clock Detection

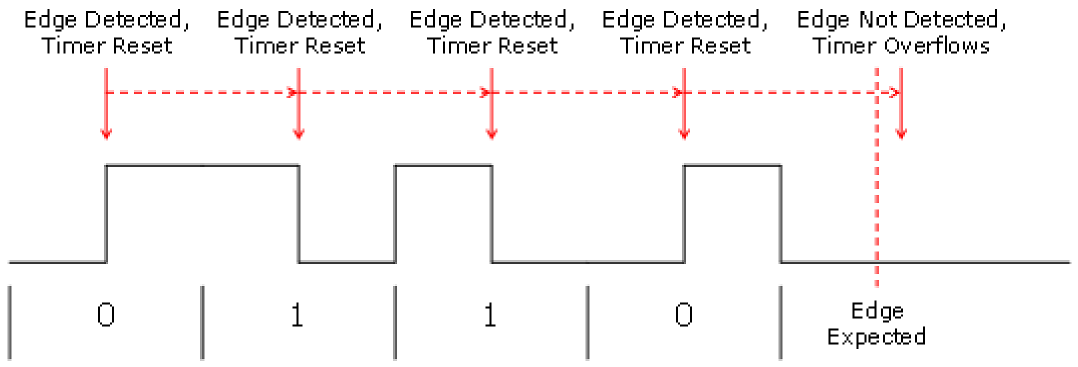

Edge Detected, Timer Reset
Edge Detected, Timer Reset
Edge Detected, Timer Reset
Edge Detected, Timer Reset
Edge Not Detected, Timer Overflows
0
1
1
0
Edge Expected

Figure 3.12 - Clock Detection Timing Diagram

A desirable additional feature is to be able to detect whether a valid clock signal is present. This would provide a more reliable alternative to the carrier detect pin of the X2010 transceiver, which is severely afflicted by noise (Figure 3.8, Figure 3.18). This is easily implemented by detecting the delay between clock edges, which should be no more than 2 chip times. Each time a clock edge is detected, a "watchdog" timer is set to overflow in 2.2 chip times, which allows for a clock rate tolerance of 10%. If the clock edges cease for more than 2.2 chip times, the timer will overflow, causing an interrupt (Figure 3.12). The advantage of this approach is that it only incurs an overhead of 6 instruction cycles for each clock period (to reset the timer), as no additional processing is required unless the clock is actually lost.

It is expected that random noise pulses may occasionally be detected as a valid clock signal, thus both the following conditions must be met to detect valid *data*:

1. At least 8 consecutive clock pulses have been detected.
2. Either:
    a. The shift register contains a preamble byte.
    b. The start of frame has been detected.

Therefore, the *valid data* flag will only be asserted by a valid preamble followed by a start of frame sequence and frame body.

3.3.3.3 Frame Synchronisation

3.3.3.3.1 Start of Frame Sequence Detection

The current mechanism for detecting the start of each frame is to prefix each frame with several preamble bytes, followed by a single *start of frame* (SOF) byte. When the receive buffer contains three preamble bytes and one SOF byte, the start of the frame is detected.

However, without start and stop bits, the receiver will not initially be synchronised to the start of each byte, thus the start of the frame cannot be detected by simply decoding the preamble and SOF bytes.

Initially, the data bits are shifted into a 16-bit shift register as they are decoded. Each time a data bit is received, the contents of the shift register are checked; if it contains one preamble byte followed by one SOF byte, the start of frame is detected. Once this occurs, the receiver is synchronised to the start of the first data byte, and can commence decoding bytes normally.

As mentioned previously, the preamble byte should consist of alternating bits to ensure synchronisation, i.e. 10101010. The Ethernet protocol uses 10101011 as a start of frame byte, however since this technique only checks 16 bits of data, the SOF byte should be as different as possible from the preamble byte. For Manchester encoded data, this would be consecutive ones or zeros, i.e. 0xFF or 0x00.

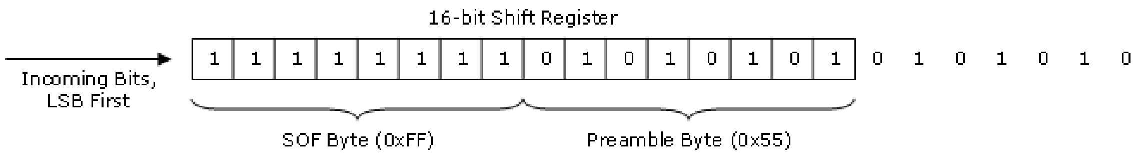


Figure 3.13 - Start of Frame Detection

3.3.3.3.2 Detecting the End of Frame

The current method for detecting the end of each frame is to use an end of frame (EOF) character. However, this prevents the use of the EOF character within the body of the frame, which cannot be enforced when the frame contains user data. Other protocols solve this problem using bit-stuffing techniques. However, it would be preferable to avoid such complexities in this project.

An alternative approach is to include the frame length as part of the header of each frame. The problem with this approach is that the frame length cannot be verified by the CRC until the entire packet has been received. However, consider the consequences of an erroneous frame length value:

1. If the received frame length is less than the actual frame length, the receiver will terminate reception prematurely, causing the CRC check to fail.
2. If the received frame length is more than the actual frame length, the receiver will continue receiving after the frame ends. This may have two outcomes:
    a. The receiver will be reset when it detects that the clock is no longer present, and the frame will be discarded as if it were incomplete.
    b. If for any reason a clock is still present, decoding will continue until the receive buffer is full, and the additional bytes will cause the CRC check to fail.

3.3.4 Software Buffers

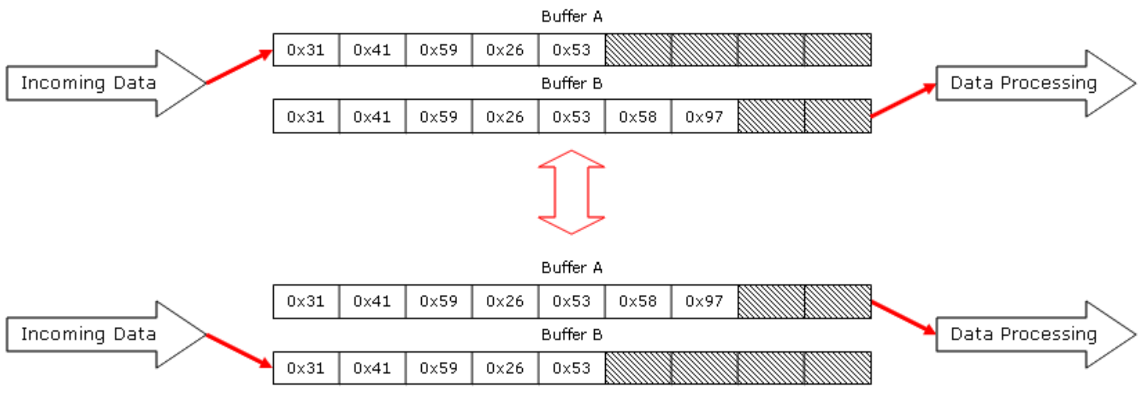


Figure 3.14 - Popcorn Buffering

Data buffering is an important part of the encoding and decoding processes. The current software only provides one buffer each for transmission and reception, which precludes simultaneous data communications and processing. That is, when a packet has been received, the next packet cannot be received until the previous one has been processed.

One technique used to allow simultaneous data processing and reception is known as popcorn buffering. This comprises two identical buffers, one used for reception, and one used for processing. Once a frame has been processed, and the next one fully received, the buffers are switched (Figure 3.14). This technique is also equally applicable to data transmission.

This switchover should ideally be transparent to the receiving and processing algorithms, thus a portable data structure is required to represent the buffers. An example linear buffer data structure (Figure 3.15) consists of:

1. A contiguous block of memory which contains the buffer data.
2. An end pointer, which points to the next available memory location.
3. A cursor pointer, which points to any memory location up to the end pointer.

The Size property always has the same as the value as the End pointer, and the Length property is obtained using the *scnsz* assembler directive in MPASM.

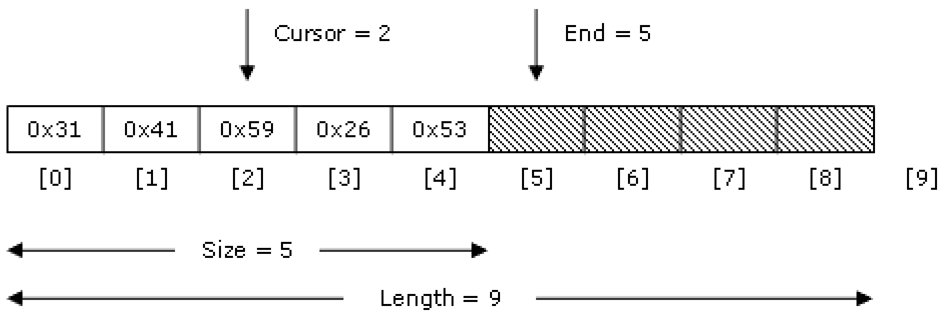


Figure 3.15 - Buffer Data Structure

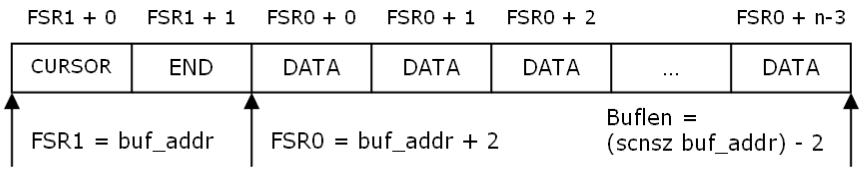


Figure 3.16 - Buffer Data Structure Memory Map

The data memory block, end pointer, and cursor pointer are all stored in a contiguous block of memory (Figure 3.16), which means that any block of memory at least three bytes long can be used as a buffer. A particular buffer is selected by calling a macro which loads up the addresses of the buffer to FSR0 and FSR1 (Figure 3.16). The functions which may be called on this buffer include:

1. Clear the buffer.
2. Set the cursor location.
3. Get the cursor location.
4. Get the size (end) of the buffer.
5. Set the end of the buffer.
6. Write a byte at the current cursor location and increment the cursor.
7. Read a byte from the current cursor location and increment the cursor.
8. Check if the buffer is full (End == Length).
9. Check if the cursor is at the end of the data (Cursor == End).

Each of these functions is implemented as a macro, which manipulates the buffer using FSR0 and FSR1.

3.4 Media Access Control

As concluded in the literature review, the power save component of the MAC protocol should implement a schedule consisting of alternating "high traffic" and "low traffic" periods. Preamble sampling is to be implemented during the low traffic period, with adaptive techniques used to optimise power savings for anticipated network traffic. The preamble sampling technique will be modelled, discussed, and optimised for static network parameters. Techniques for adaptively reconfiguring the protocol will be discussed and developed, including synchronising schedules across the network. This will be concluded with a discussion of techniques for media contention and collision management.

3.4.1 Preamble Sampling

Recall that in preamble sampling, nodes periodically wake up and sample the medium for activity; nodes remain awake if a preamble is detected, otherwise return to sleep (Figure 2.5). The preamble is the same length as one sampling cycle, which guarantees that it will be detected by all neighbouring nodes irrespective of their relative phases.

The energy required to transmit and receive the extended preamble partially offsets the power savings made during the sleep intervals, and also increases the probability of collisions due to increased packet length. Therefore, the energy saving provided by this technique will be dependent on the chosen sleep interval, as well as the network conditions and hardware specifications. A model will be developed, which can be used to determine the optimal sleep interval and maximum energy savings for a particular scenario.

There are other performance metrics which are also affected by the sleep interval, such as packet latency; however these are not of primary importance and thus shall not be analysed.

3.4.1.1 Modelling and Optimisation

The following set of parameters describes the preamble sampling technique, characteristics of the nodes, and network conditions:

1. The sleep interval, i.e. the time between each sample, t (seconds).

2. The power-up delay of the transceiver, i.e. the time required by the receiver to provide a valid RSSI reading, T_{PWUP} (seconds).

3. The time required to assess the channel, i.e. the time required by the microprocessor to perform an A/D conversion on the RSSI signal, $T_{A/D}$ (seconds).

4. The listen interval, $T = T_{PWUP} + T_{A/D}$ (seconds).

5. The power consumed whilst the node is transmitting, P_t (Watts).

6. The power consumed whilst the node is receiving, P_r (Watts).

7. The average number of packets transmitted per second, f (Hz).

8. The average number of nodes affected by each transmission, n, which is dependent on the network density.

The model will predict the *average power saving per node* for these parameters, which as opposed to *total power consumption* allows the removal of parameters such as baud rate and packet length. Other performance metrics, such as throughput and latency will not be modelled, as they are of secondary importance to power consumption. For now, the effects of packet collisions are ignored to simplify the model, as collisions are relatively unlikely in a sparse low traffic network. Collisions and subsequent retransmissions could be incorporated into the mode by defining a new parameter, actual packet rate f' (t, l, n, f), i.e. a variable dependent on the preamble length, packet length, node density, and packet rate.

The approximate average power consumption for an idle node is given by:

$$P_{saving} = \left(\frac{t}{T+t}\right)P_r \quad \text{[Watts]}. \tag{3.2}$$

The additional energy required to transmit the preamble is given by:

$$E_{transmit} = tP_t \quad \text{[Joules]}. \tag{3.3}$$

Assuming that on average each node will wake up halfway through the preamble, the additional energy required to receive the preamble is given by:

$$E_{receive} = \frac{tP_r}{2} \quad \text{[Joules]}. \tag{3.4}$$

Therefore, the net power saving per node is given by:

$$P_{net} = P_{saving} - f\left(E_{transmit} + E_{receive}\right) \quad \text{[Watts]}. \tag{3.5}$$

$$P_{net} = \left(\frac{t}{T-t}\right)P_r - f\left(tP_t + n\frac{tP_r}{2}\right) \quad \text{[Watts]}. \tag{3.6}$$

Optimising this equation for the largest power saving gives the following expression for the optimal value of *t*:

$$t = \sqrt{\frac{2TP_r}{f(2P_t + nP_r}} - T \quad \text{[Seconds]}. \tag{3.7}$$

The net power saving given in Eq. (3.5) is plotted against sleep interval for several different network densities and packet rates in Figure 3.17. The definitions of the various traffic rates used are presented in Table 3.1, and the fixed parameters derived from data sheets [4, 38], are presented in Table 3.2.

Table 3.1 - Classification of traffic rates

Traffic Rate	Packet Rate (Hz)
Low	1/1800
Medium	1/60
High	1

Table 3.2 - Hardware Parameters

Parameter	Value	Derivation
T	1.05 ms	T_{PWUP} = 1 ms [4] $T_{A/D}$ = 50 μs [39]
P_t	215 mW	I_q(VGA) = 35mA [38] I_{tx}(X2010) = 8mA [4] V_{cc} = 5V
P_r	35 mW	I_{rx}(X2010) = 7mA [4] V_{cc} = 5V

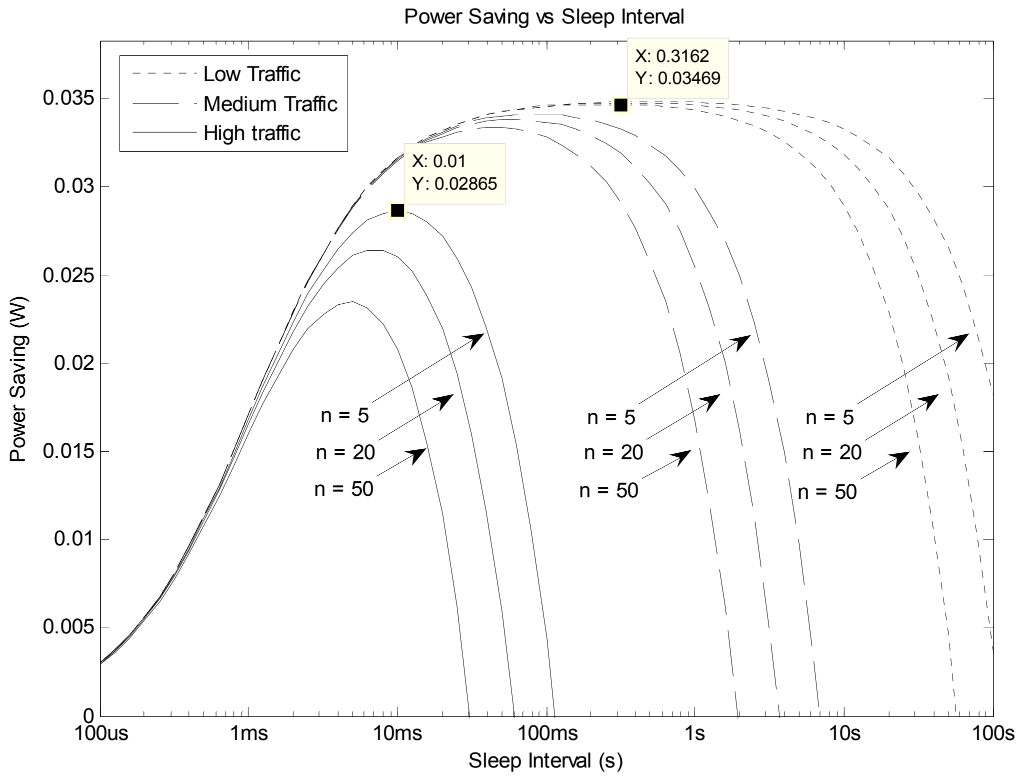


Figure 3.17 - Power Saving vs. Sleep Interval

Note in Figure 3.17 that the both the optimal sleep interval and potential power savings are highly dependent on the amount of network traffic. Also note that the saving is increased for lower values of *n*, thus this technique is well suited to sparse networks.

Without preamble sampling, the idle power consumption of the radio is 35 mW (Table 3.2). However, with preamble sampling, a power saving of up to 34.69mW is possible, thus the power consumption per node is a mere 310 μW, plus the power consumption of transmitting packet bodies. Even with a high traffic rate, a power saving of 28.65 mW is possible. However the actual power savings would be slightly less, due to a higher of probability packet collisions and subsequent retransmission.

3.4.1.2 Dynamic Reconfiguration

As discussed in Section 2.1.7 (p. 14), the power save protocol will extend the preamble sampling concept, by including both scheduled and adaptive reconfiguration to optimise the sleep interval for the anticipated traffic rate.

3.4.1.2.1 Scheduled Reconfiguration

Bulk data transfer will be constrained to brief intervals which occur at predetermined times, thus an optimised reconfiguration can be scheduled for these periods. Section 2.1.7 (p. 14) discussed defining these “high traffic” periods so that all nodes are continuously awake for the entire period. However, not all parts of the network will be active at all times during this period. Therefore, rather than all nodes being continuously awake, it may in fact be more energy efficient to optimise the preamble sampling technique for the higher traffic rates. The optimal sleep interval would have to be calculated beforehand at the application layer, based on the amount of traffic anticipated during the high traffic period. However, depending on the actual traffic rate, it may actually be more energy efficient for nodes to not sleep during this period. This is especially true considering the increased probability of collisions.

3.4.1.2.2 Adaptive Reconfiguration

During network operations such as address allocation, replies from neighbouring nodes will cause a period of localised high traffic immediately after the request. By providing an interface for higher level protocols to dynamically reconfigure the MAC layer, the sleep interval can be optimised based on this anticipated traffic. The network layer header will include a “high traffic timer” field indicating how long, if at all the source node will remain active for, so that neighbouring nodes can use a shorter preamble for packets it sends to that node.

Note that this technique is particularly effective for broadcast requests such as address allocation and route discovery, as every neighbouring node makes a power saving, whilst only the requesting node consumes more power.

3.4.2 Schedule Synchronisation

The dynamic preamble scheduling scheme presented in the previous section provides a good mechanism to optimise power savings during anticipated high traffic periods, and maintains network connectivity during low traffic periods at a higher transmission energy cost. However, a mechanism is needed to allow the traffic schedule to be globally synchronised. Three such mechanisms shall be discussed:

1. An active synchronisation mechanism which is propagated globally and initiated by a single node.

2. A phase discovery and correction mechanism which counters the effects of clock drift.

3. A synchronisation request mechanism which allows nodes to resynchronise if they believe they are completely out synchronisation with the rest of the network.

3.4.2.1 Active Synchronisation

Although an ad hoc sensor network is by definition infrastructure-less, there will be a sink node which all sensors periodically transfer their data to. The sink node will likely be permanent, and hence is a logical point to initiate active synchronisation operations from.

The active synchronisation operation will commence with the sink node broadcasting a synchronisation message, which includes the value of its clock at the exact time the message was *transmitted* (as opposed to the time when the message was *queued for transmission*). Nodes which receive this message will update their clocks, and set a timeout during which they will ignore all synchronisation messages. They will then rebroadcast the message and include the exact time that *they* transmitted the message. Synchronisation messages will use the longer preamble so that they will be received by neighbouring nodes regardless of which traffic mode they are in.

A likely traffic schedule would be 5 minutes of high traffic followed by 6 hours of low traffic, hence timing will be performed by a real time clock chip, which has a precision of one second. Because the DS1305 real time clock chip [35] resets its internal counter when the seconds register is written, even in the absence of any delays there will be a lag error of up to one second with each hop. Assuming the values of the RTC chips' internal counters are random and uniformly distributed, the average lag will be 0.5 seconds per hop. Considering that the propagation and processing delays will add to this error, the absolute value of the error can be reduced by adding one second to the clock value each time a synchronisation message is broadcast.

3.4.2.2 Phase Discovery and Correction

Consider the high traffic period to be divided into one second slots. The network header for each packet transmitted during the *scheduled* high traffic period includes the number of the slot it was transmitted in with respect to the source node. This information allows neighbouring nodes to calculate the relative phase of their clock schedules with a precision of one second, and if necessary adjust their schedule to counter the effects of clock drift.

This concept of phase discovery and adjustment is similar to that used in [9]. However, in this case the phase information is piggybacked onto regular packets, and hence incurs minimal overhead.

3.4.2.3 Synchronisation Request

Immediately after a node joins the network and is allocated an address, it should request synchronisation from its neighbours using the low traffic preamble, so that it may synchronise to the global traffic schedule. This should also be performed if a node otherwise believes it has completely lost synchronisation with the rest of the network, e.g. if it did not detect any activity during the last high traffic period.

3.4.3 Media Contention and Collision Management

In 2004, Sloots [2] developed a Carrier Sense, Multiple Access protocol with exponential backoff. Before a frame is transmitted, the CD pin of the transceiver is tested to check if the media is free. If the media is busy, the node generates a timeout with exponential backoff:

$$\textit{Timeout} = 0.5\text{s} \times 2^{n} \tag{3.8}$$

Where n is the number of times the node has tried to transmit. When n reaches a predetermined retry limit, the transmission is abandoned.

There are two problems with this protocol which will be addressed in the following sections.

3.4.3.1 Fixed Backoff Period

The first problem is that the initial timeout period is fixed at 0.5 seconds. If two nodes unsuccessfully attempt to transmit at the same time, their subsequent retries will also coincide, causing repeated collisions. The Ethernet protocol has dealt with this problem by randomising the timeout according to the formula:

$$\textit{Timeout} = k \times \textit{rand()} \times 2^{n} \tag{3.9}$$

Where k is the slot time, and *rand()* is a random number between 0 and 1. A random number can be quickly generated by taking the CRC, which is pseudo-random for each packet, and XORing it with TIMER3, which is pseudo-random for each retry. This number will then be truncated using a bit mask, to limit the maximum timeout to $k \times 2^{n}$.

3.4.3.2 Clear Channel Assessment

The second problem is that any noise present on the transceiver frequency may be incorrectly be demodulated as a valid carrier, which has previously been noted by Sloots [2].

This theory was tested by observing the voltage on the active low carrier detect (/CD) line with no signal present. When a signal is present, the /CD voltage is stable at 0V. However, when a ¼ Wavelength antenna is connected with no signal present, the transceiver detects significant amounts of noise as a valid carrier (Figure 3.18).

As can be seen in Figure 3.18, the /CD voltage regularly drops below 0.8V (TTL logic low), which would cause the node to incorrectly detect that a carrier is present. There are several alternatives to using the /CD line to detect if a valid signal is present:

1. Low pass filter the /CD line, and perform an A/D conversion on it. If the result is more than zero, then there is no valid signal present.
2. Perform an A/D conversion on the RSSI line. If the result is less than the noise floor, then there is no valid signal present.
3. Use the *clock detect* flag as an indicator of whether a valid signal is present.

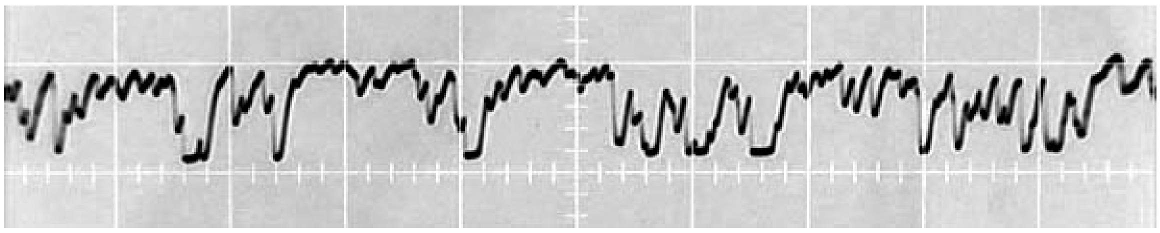


Figure 3.18 - Carrier Detect (Active Low), No signal present, 5V/div, 1ms/div

3.5 Address Allocation

As discussed in section 2.2.6 (p. 28), the address protocol for this protocol shall be based on the concept of address spaces, and attempt to incorporate the salient features of both [23, 31]. The improvements made to [23] can be summarised as follows:

1. A requesting node does not accept the first address offer it receives. Instead, it waits for all replies to arrive, then accepts the largest offer (from [31]). This promotes more even distribution of available addresses throughout the network.
2. When a requesting node receives several offers from neighbouring nodes, it does not explicitly reject each individual offer it did not accept. Instead, a single broadcast message is used to accept the successful offer, which implicitly rejects all other offers. This significantly reduces the number of messages required to complete an address allocation operation.
3. The network will not be flooded with an address search message unless the first address request is unsuccessful. This is achieved by including a retry number in the request message, allowing recipients of the message to determine how many unsuccessful requests have been made. This prevents unnecessarily flooding the network with an address search message.

The operation of this protocol is described in the following sections.

3.5.1 Network Setup

For this network event, the protocol is identical to Tayal and Patnaik [23]. When the very first node starts up, it will broadcast an AREQ (address request message), and wait for a reply (Figure 3.19). Since there are no other nodes in the network, it will not receive any replies, and will rebroadcast the AREQ message AREQ_RETRY_LIMIT times. After this time it will conclude that it is not in range of an existing network, and create a new network by taking possession of the entire address space.

3.5.2 Node Join

When a node starts up, it will enter the INIT (initialisation) state and assume the address 0x00, which is designated as an anonymous address for initialising nodes. Since the node does not have a MAC address, it randomly generates an 8-bit unique id (UID) to distinguish it from other nodes which are simultaneously initialising. The node will broadcast an AREQ message, and wait for replies from neighbouring nodes (Figure 3.19).

Neighbouring nodes which receive this message will check if they have any available addresses. If a node has spare addresses, it will enter the ALLOC (allocation) state, and mark half of its address space as under allocation. It will then send an AREP (address reply message) indicating the address range it is willing to allocate (Figure 3.20).

Two distinct cases may arise from this scenario.

1. At least one neighbouring node has available addresses, and sends an AREP message to the requesting node (Local Allocation).
2. There are no neighbouring nodes with available addresses. The requesting node receives no replies, and hence rebroadcasts the AREQ message. Neighbouring nodes receive the second request and infer that the first request failed, so they initiate a global address search.

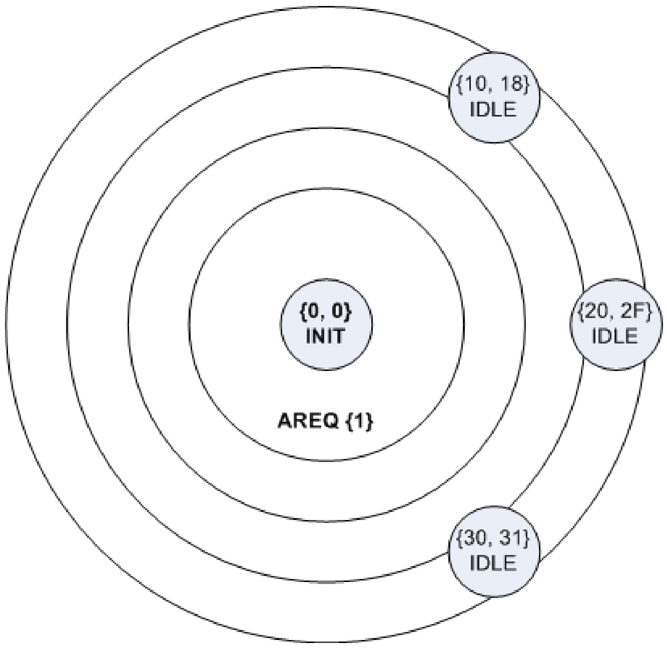


Figure 3.19 - Address Request

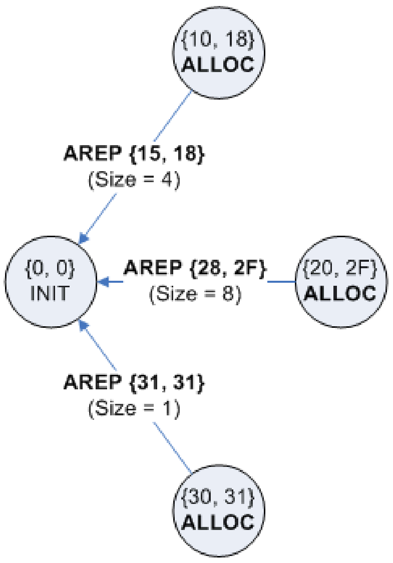


Figure 3.20 - Address Offers

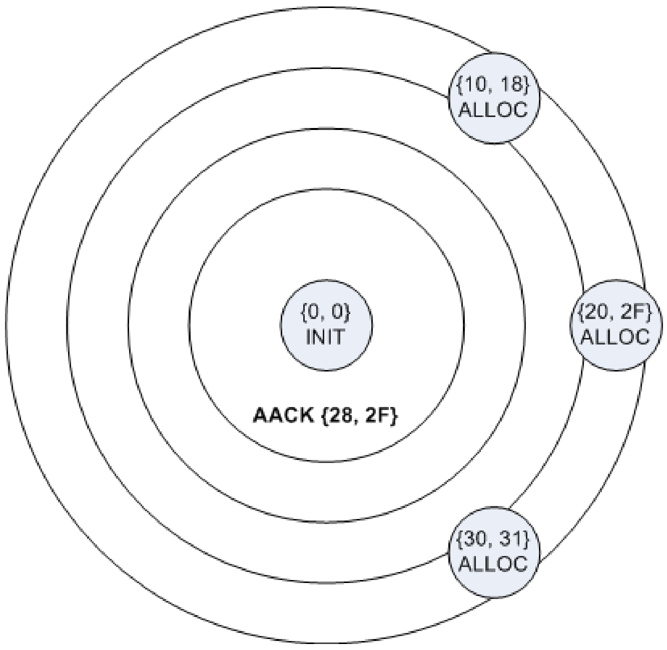


Figure 3.21 - Offer Accepted

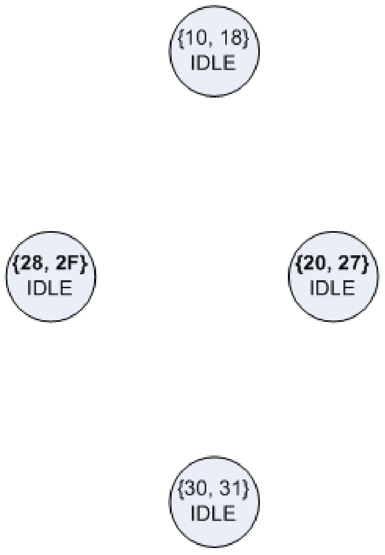


Figure 3.22 - Idle

3.5.3 Local Allocation

In this case, the requesting node does not immediately reply to any AREP messages it receives, rather it caches them until a timeout expires (Figure 3.20). This timeout is set so that the requesting node has the opportunity to receive replies from all its neighbours before deciding which offer to accept. This feature promotes a more even distribution of addresses throughout the network, as implemented in ZAL [31].

Once the reply timeout has expired, the requesting node will broadcast an AACK (address accept message) indicating which offer it is about to accept (Figure 3.21). It will take possession of the largest address space it was offered, and enter the IDLE state (Figure 3.22).

Assuming all allocating nodes received the AACK message, the node whose offer was accepted will delete its addresses under allocation, whilst all other nodes will repossess their address under allocation (Figure 3.22). Note in [23] that the requesting node unicasts an AREJ (address reject message) to every node whose offer it did not accept. In this protocol the rejection is implicit, significantly reducing the number of transmissions required to complete the operation.

3.5.4 Global Address Search

A global address search will be initiated when a node receives an address request with a retry number of two or higher, and does not have any available addresses. Firstly, it will send a negative reply to the requesting node to acknowledge that it has received the request. Secondly, it will broadcast an ASRCH (address search) message. Nodes which receive this message and have available addresses will send an address reply back to the requesting node. The requesting node will reply with either an AACK or an AREJ. Nodes which do not have available addresses will send a negative reply back to the requesting node, rebroadcast the ASRCH message, and ignore all future ASRCH message originating from the same node. In this way, the ASRCH message will be flooded throughout the network until an available address space is found.

[23] Does not specify how remote nodes are able to route their replies back to the requesting node, and vice versa. Here are two techniques that could be used to achieve this:

1. Each ASRCH message includes a route back to the requesting node (Figure 3.23). Therefore, each time a node broadcasts an ASRCH message it appends its own address to this route. This technique is used in [27]. A disadvantage of this technique is that in a large and sparse network, the address list could get quite long, introducing additional overhead into the operation.

2. When a node broadcasts an ASRCH message, it enters a *proxy* state, in which it forwards all AREP and NREP messages back toward the requesting node. In this way, a well-defined connection is formed between the requesting and allocating nodes. However, each proxy node only knows about the next node in the route, thus routing information is not needed in the reply messages. This is very similar to the DSR adaptation developed by Steven Sloots [2]. This technique introduces an opportunity to reduce the amount of traffic sent back to the requesting node: If a proxy node receives two positive replies, it can reject the smaller offer locally, and only send the larger offer back to the requesting node. The disadvantage of this technique is its complexity.

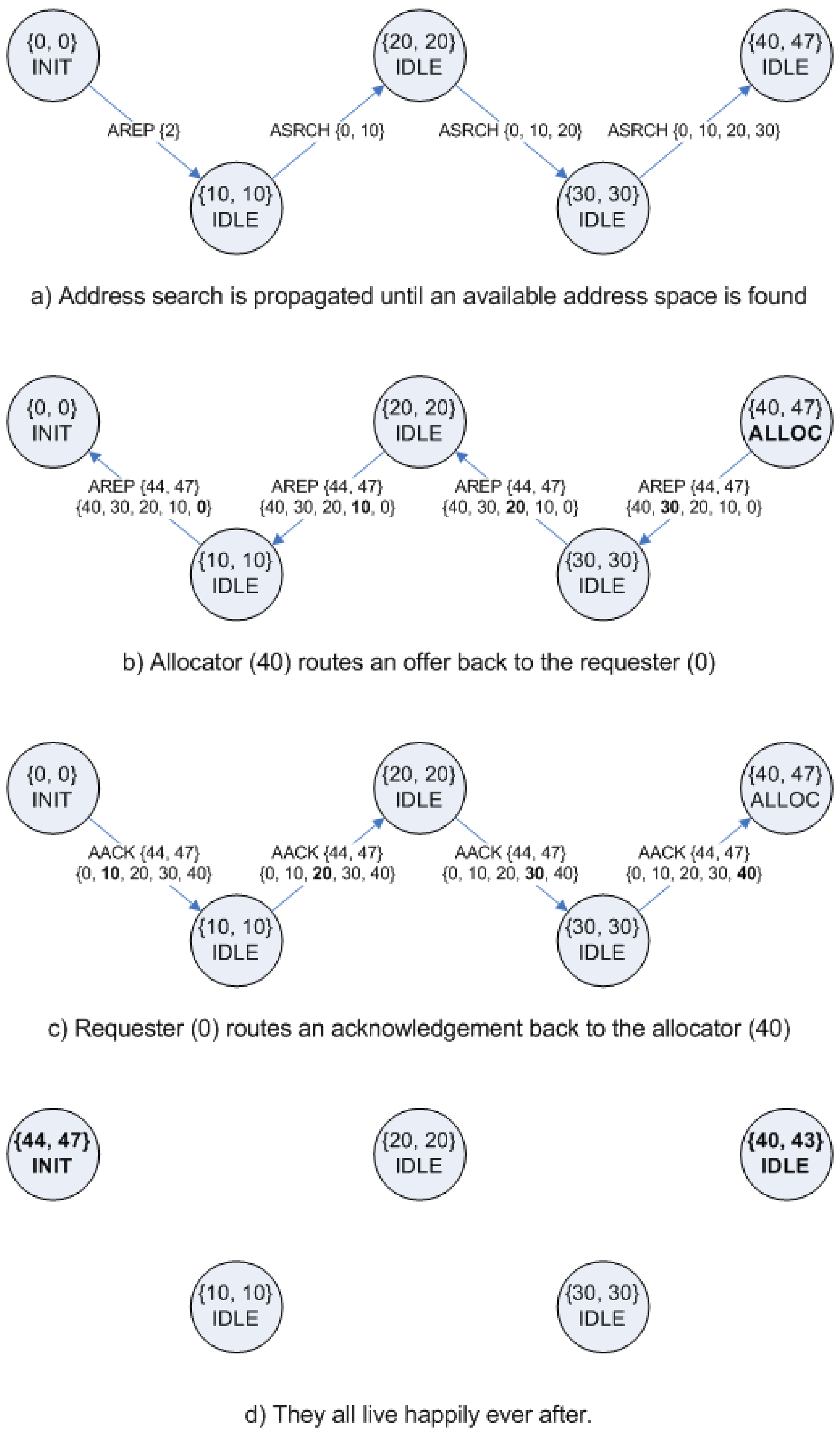

{0, 0} INIT
{20, 20} IDLE
{40, 47} IDLE
AREP {2}
ASRCH {0, 10}
ASRCH {0, 10, 20}
ASRCH {0, 10, 20, 30}
{10, 10} IDLE
{30, 30} IDLE
a) Address search is propagated until an available address space is found
{0, 0} INIT
{20, 20} IDLE
{40, 47} ALLOC
AREP {44, 47} {40, 30, 20, 10, 0}
AREP {44, 47} {40, 30, 20, 10, 0}
AREP {44, 47} {40, 30, 20, 10, 0}
AREP {44, 47} {40, 30, 20, 10, 0}
{10, 10} IDLE
{30, 30} IDLE
b) Allocator (40) routes an offer back to the requester (0)
{0, 0} INIT
{20, 20} IDLE
{40, 47} ALLOC
AACK {44, 47} {0, 10, 20, 30, 40}
AACK {44, 47} {0, 10, 20, 30, 40}
AACK {44, 47} {0, 10, 20, 30, 40}
AACK {44, 47} {0, 10, 20, 30, 40}
{10, 10} IDLE
{30, 30} IDLE
c) Requester (0) routes an acknowledgement back to the allocator (40)
{44, 47} INIT
{20, 20} IDLE
{40, 43} IDLE
{10, 10} IDLE
{30, 30} IDLE
d) They all live happily ever after.

Figure 3.23 - ASRCH message contains a routing information

Chapter 4

IMPLEMENTATION AND TESTING

4.1 HARDWARE PLATFORM

4.1.1 Node PCB Design and Construction

The hardware designed for this project was based on the hardware designed by Steven Sloots [2] in 2004 (Figure 3.1, p29), with the major modifications listed below:

1. The VGA bypass relay was replaced with an ADG918 CMOS switch [34].
2. An emitter follower voltage to supply the ADG918 with 2.75 Volts.
3. A DS1305 Real Time Clock Chip [35] added to the microcontroller's SPI bus.
4. An RC network to reduce the noise on the X2010 transceiver's RSSI line.
5. An RJ12 socket enabling compatibility with the Microchip ICD2.

Although the same component interconnections were retained, the PCB was completely redesigned to accommodate the additional hardware, with an emphasis on features which would assist debugging and modifications to both hardware and software. Some of these features include:

1. Strict rules for track sizes, which allows easy identification of power tracks and RF tracks.
2. Easily accessible test points for transceiver data pins and power rails.
3. Additional status LEDs and a power indicator LED.
4. Grouping of related components and tracks, and use of component dress where practical.

Microcontroller IO pins were not assigned until the PCB design stage, which allowed tracks to be routed with minimal use of vias and allowed all tracks connected to a particular devices to be grouped together into a bus. An 8-pin DIP IC socket was placed in series with the transceiver's four data lines, to allow certain functions to be isolated and / or externally controlled. The connections are normally made with short lengths of bell wire.

Care was taken to separate components and tracks which may cause interference with each other, and separate ground planes were poured for the digital, analogue, and RF sections of the circuit.

The Microchip ICE2000 in-circuit emulator was used as the microprocessor development platform, which connects to the PCB via a 44-pin PLCC socket. An RJ-12 socket was included to allow compatibility with the Microchip ICD2 in-circuit debugger, however was not used due to problems obtaining a PIC18C452 device in a 44-pin PLCC package.

Each PCB was constructed in stages, with exhaustive testing at each stage of completion. In particular, after the power supply components and ground vias were installed, all nets on the PCB were tested to ensure that they were connected to the correct voltage levels. All external peripherals were tested in software, to ensure that they were functioning and communicating correctly with the microcontroller. A particular problem encountered was that the interrupt line of the DS1305 clock chip did not have a 4.7 kΩ pull-up resistor as required, which caused it to be continuously asserted.

Construction of two functional nodes was planned, so that MAC and network layer protocols could be developed and tested on a real hardware platform. Unfortunately, after the first node was fully constructed, it was discovered that the 44-pin PLCC footprint used from Mr Sloots' PCB libraries was incorrect, rendering the entire PCB useless. As a result, the PCB had to be disassembled, and redesigned to accommodate the correct footprint.

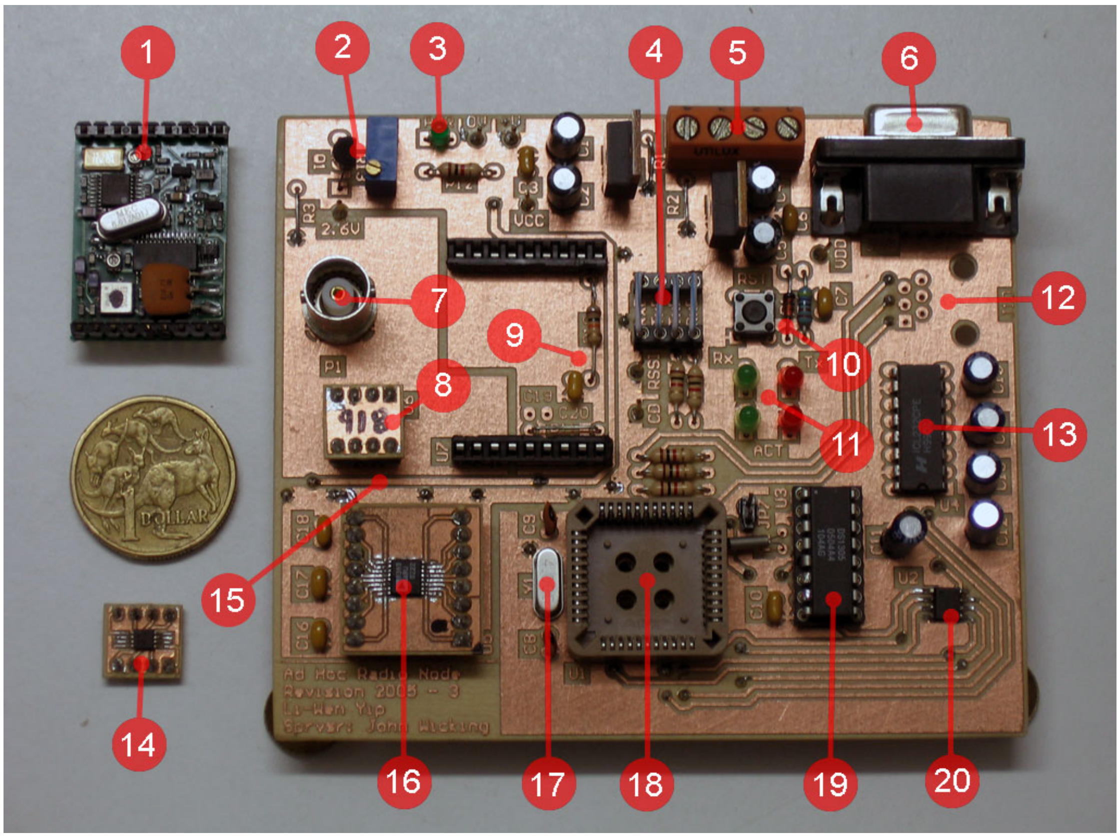


Figure 4.1 - Ad hoc radio node PCB; Revision 3 as constructed

1	X2010 transceiver module.
2	Emitter follower voltage source.
3	Power LED.
4	Communications isolation / external connection header.
5	Power connection.
6	RS-232 interface.
7	50Ω antenna connection.
8	ADG918 CMOS switch carrier board.
9	RSSI low pass RC network.
10	Reset switch and slow power-on reset circuit.
11	Indicator LEDs.
12	RJ12 connector for Microchip ICD2 (not installed).
13	MAX232 level shifter.
14	ADG918 CMOS switch carrier board (reverse side).
15	Power track.
16	AD8369 variable gain amplifier.
17	External 4Mhz crystal oscillator.
18	44-pin PLCC socket for Microchip ICE2000.
19	DS1305 Real Time Clock.
20	FM25C160 FRAM.

4.1.2 ADG918 / ADG919 Wideband CMOS Switch

The ADG918 / ADG919 were not available through any of the university's regular suppliers, so several samples had to be ordered directly from Analog Devices. The devices were only available in a tiny MSOP-8 Package, which is extremely difficult to solder by hand, and impractical to transfer to a new PCB if required. To solve this problem, a carrier board was designed to plug into a regular DIP-8 IC socket (Figure 4.2). Unfortunately, this approach had some unforseen consequences, which will be discussed shortly. The ICs was soldered to the carrier boards with the aid of a magnifying glass and some nail varnish, and thoroughly checked for continuity and short circuits.

Once the carrier boards were constructed, they were tested to ensure they were providing sufficient off isolation, i.e. port separation in the off state. The switch is connected between the input and output of the RF amplifier, creating a positive feedback loop. To prevent oscillation or distortion, the gain of the loop must be much less than one i.e. the off isolation must be much greater than the amplifier gain. Due to the licence-free power limit in the 433 MHz band, the amplifier will be operated with a maximum gain of 14dB.

A 433 MHz, -60dBm signal was fed into the common port of the switch via the PCB mounted BNC connector, and a spectrum analyser connected to port 1 via an alligator clip terminated coaxial cable. The signal strength was measured in both the "on" and "off" states, with the difference being the inferred port separation.

The ADG918/919 data sheet [34] specifies the off isolation at 433 MHz to be approximately -43dB. However, the initial test results indicated this isolation to be a mere -10dB. It was observed the alligator leads terminating the coaxial cable were unshielded and approximately ¼ λ (17 cm) in length. It was therefore inferred that crosstalk from these leads was affecting the result, so the experiment was repeated with a coaxial cable soldered directly to the carrier board (Figure 4.3). This second test indicated the off isolation to be approximately -30dB. The carrier board was then soldered directly to the PCB to eliminate any impedance mismatching or crosstalk caused by the IC socket. A subsequent third test indicated the off isolation to be approximately -35dB (Figure 4.4, Figure 4.5). Whilst this separation is sufficient to prevent oscillations in the amplifier, the result could be improved with better RF design, e.g. sufficient shielding between RF lines.

A final test was performed to measure the current consumption of the device in the on state. The specified current consumption for the device is 1 μA, however the multimeter used had a precision 10 μA, and as expected did not record a reading. Despite this, it is clear that this device provides a vast power saving over the previously used relay.

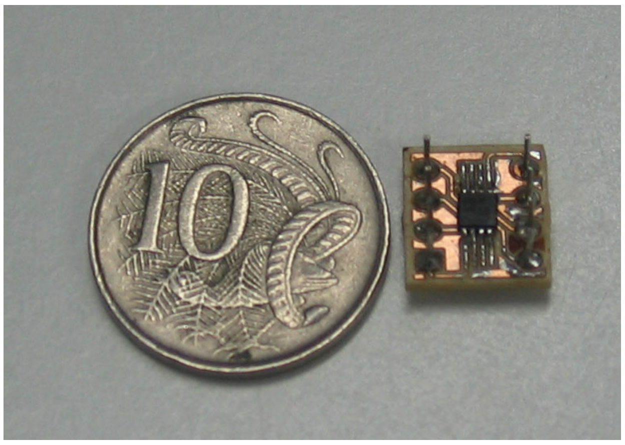


Figure 4.2 - ADG918/ADG919 Carrier Board

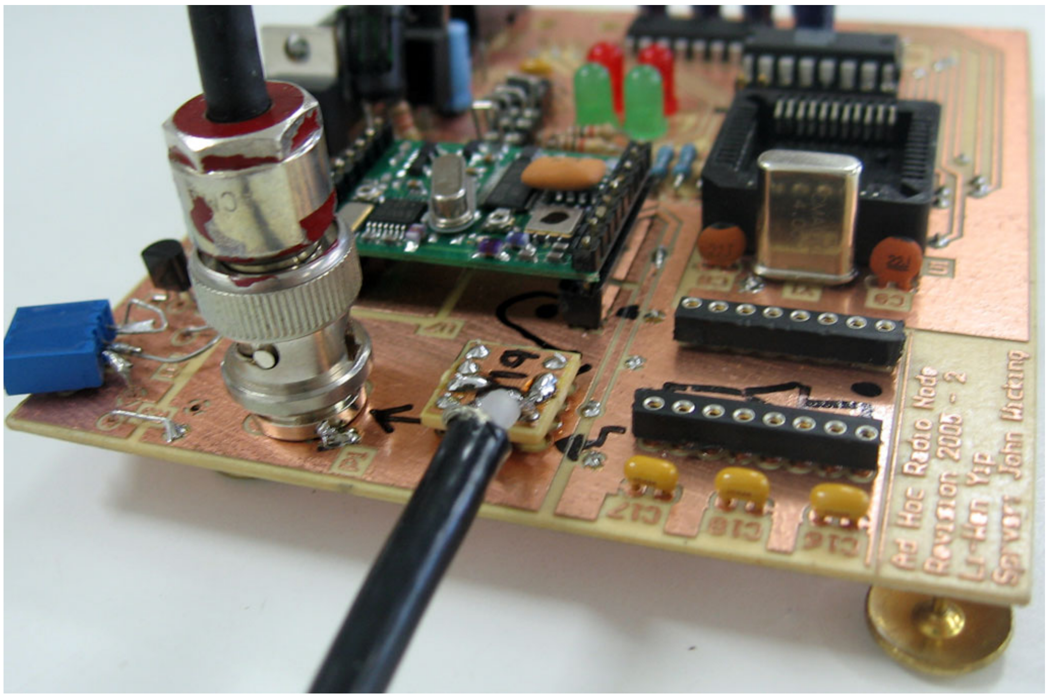


Figure 4.3 - Carrier board soldered directly to PCB;
Coaxial cable soldered directly to carrier board

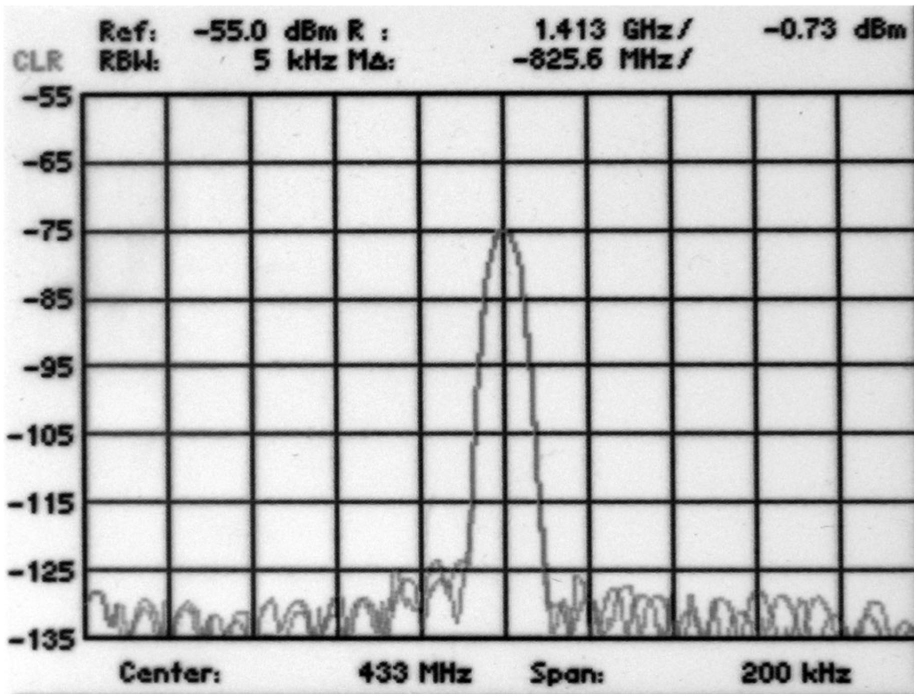

Ref: -55.0 dBm R : 1.413 GHz/ -0.73 dBm
CLR RBW: 5 kHz MΔ: -825.6 MHz/
-55
-65
-75
-85
-95
-105
-115
-125
-135
Center: 433 MHz Span: 200 kHz

Figure 4.4 - Signal strength in ON state (-75dBm)

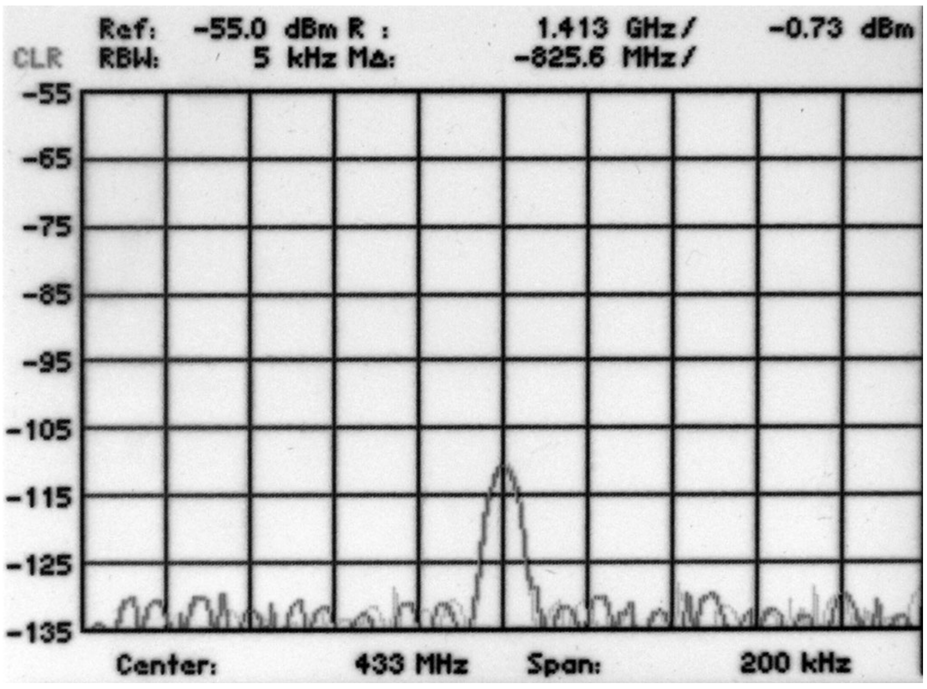

Ref: -55.0 dBm R : 1.413 GHz/ -0.73 dBm
CLR RBW: 5 kHz MΔ: -825.6 MHz/
-55
-65
-75
-85
-95
-105
-115
-125
-135
Center: 433 MHz Span: 200 kHz

Figure 4.5 - Signal strength in OFF state (-110dBm)

4.2 MANCHESTER ENCODING AND DECODING

4.2.1 Manchester Encoding

4.2.1.1 Development

A number of problems were encountered during the development of the encoding algorithm, mainly due to undocumented problems with the operation of the PIC18C452 Capture/Compare modules.

1. The 'Toggle output on match' compare mode does not work in the MPSIM simulator (MPSIM v8.62.01.0).
2. The 'Generate software interrupt on compare match (CCPIF bit is set, CCP pin is unaffected)' mode forces the output low when the CCPxCON register is written. (ICE2000)

The 'Generate software interrupt' mode was used when there were two consecutive similar chips, as the output did not need to be changed. Consequently, when there were two consecutive high chips, the output was forced low halfway through the first chip, corrupting the data.

The solution was to skip a chip period if there are two consecutive similar chips, rather than programming the output not to change. This approach also has some additional benefits:

1. The CCP module does not need to be reprogrammed for every chip; rather it can be left in 'toggle on match' mode.
2. Each time a chip period is skipped, the microprocessor is freed up for one chip period.

4.2.1.2 Maximum Encoding Speed

The Manchester encoding software was set up to continuously transmit 0xFF at 5000 baud, which requires one transition every 100µs. The microprocessor is running at 4 MHz, therefore 1 instruction cycle = 1µs. The code was run with a breakpoint at the end of the high priority interrupt service routine, and the TMR3 and CCP2R registers were examined to determine how many processor cycles elapsed between the interrupt occurring and resumption of normal program flow. The results of these tests are presented in Table 4.1.

Even though each data chip takes 47 cycles to encode, the encoder is able to run faster than this because each chip is calculated one chip period in advance, thus the encoder can accumulate up to one chip period of slack and still remain in synchronisation. Therefore, the minimum theoretical chip length should be equal to the average number of cycles required to calculate one chip. Each byte is composed of 8 clock chips, 8 data chips, and one byte fetch, which adds up to 676 cycles, or an average of 42.25 cycles per chip, therefore the encoder should be able to run at a chip length of 43µs, or 11628 baud.

The software was set up to continuously transmit 0xFF at various baud rates, and the resulting waveform was observed on a cathode ray oscilloscope. To provide an additional indication of whether the byte was properly encoded, the frequency and duty cycle were also measured using a digital multimeter. These results are presented in Table 4.2.

As expected, the encoder produced a perfect square wave at up to 11628 baud, or a chip length of 43µs, whereas when the chip length was reduced to 42µs, the square wave became unstable, presumably due to the encoder not being able to keep up with the baud rate.

Note that this experiment did not include the data link layer; if the data link layer is included it takes 702 cycles to encode a single byte, or 43.875 cycles per chip. This should not affect the encoder’s ability to run at 9600 baud.

Table 4.1 - Instruction Cycles for Encoding Operations

Operation	**Instruction Cycles**
Encode Clock Chip	37
Encode Data Chip	47
Encode Data Chip + Fetch Next Byte	51
Fetch byte from buffer (DL Layer)	Up to 26

Table 4.2 - Frequency and Duty cycle measurements

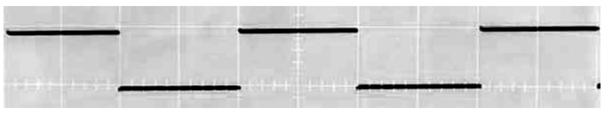
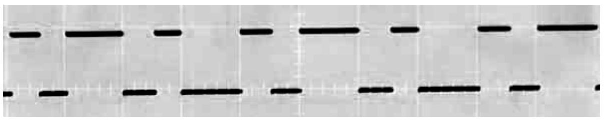
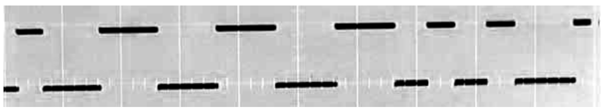
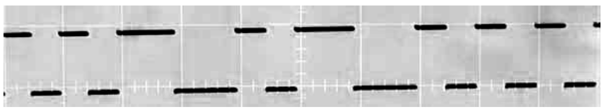
Baud	**Chip Period**	**Frequency**	**Duty Cycle**	**Waveform**
1200	416 Cycles	1.202kHz	50.0%	Stable Square Wave
2400	208 Cycles	2.404kHz	50.0%	Stable Square Wave
4800	104 Cycles	4.809kHz	50.0%	Stable Square Wave
9600	52 Cycles	9.618kHz	50.0%	Stable Square Wave
11628	43 Cycles	11.632kHz	50.0%	Stable Square Wave
11905	42 Cycles	11.720kHz	50.0%	Unstable Square Wave
12500	40 Cycles	16.00Hz	Unstable	Groups of 10 pulses.

4.2.1.3 Correct Byte Transmission

The encoding software was set up to continuously transmit a variety of bytes at 5000 baud, and the waveform was viewed on a cathode ray oscilloscope to verify the byte was being correctly transmitted.

All the tested were completed successfully, as shown in Table 4.3.

Table 4.3 - Byte Waveforms

Hex	**Binary**	**Waveform**
0xAA	10101010	
0x66	01100110	
0xEA	11101010	
0x42	01000010	

4.2.2 Manchester Decoding

4.2.2.1 Clock Synchronisation

The clock synchronisation function of the Manchester decoder was tested by setting the decoder to run at 5000 baud, and feeding a square wave into the receive data input. The frequency of the square wave was gradually increased, and noting the frequencies at which the *clock detect* flag changed state.

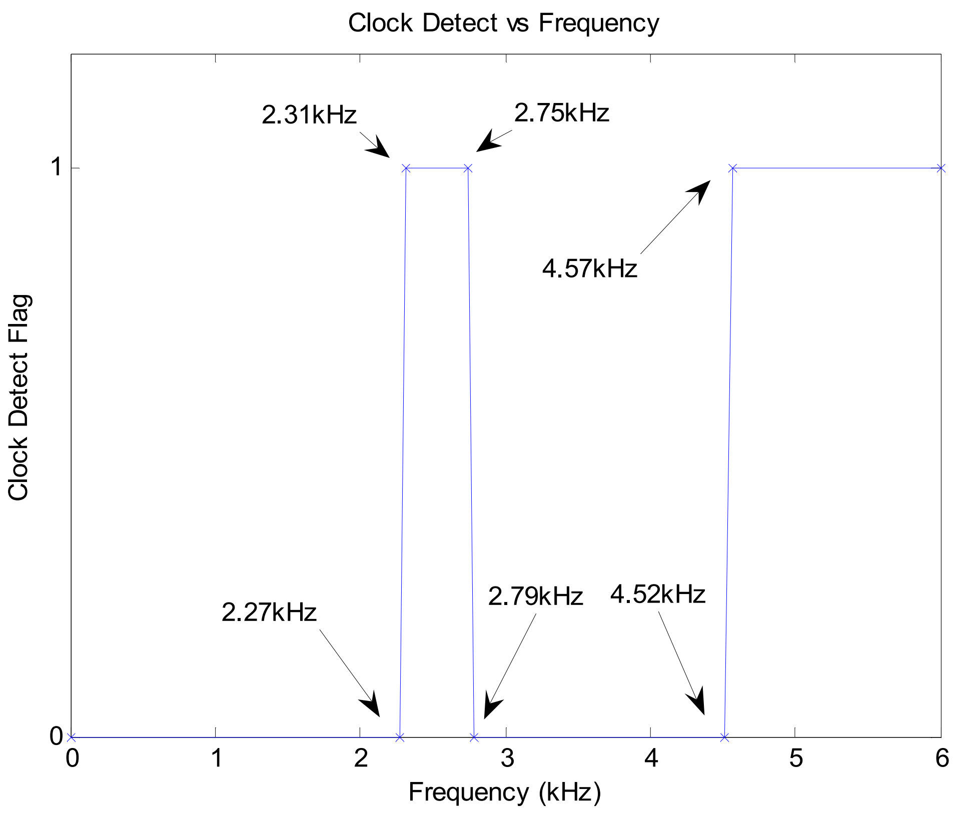


Figure 4.6 - Clock Detect Flag vs. Frequency

The first spike corresponds to a 2.5 kHz square wave, which is decoded as a continuous transmission of 0xAA or 0x55. The second spike corresponds to a 5.0 kHz square wave, which is decoded as continuous transmission of 0x00 or 0xFF. A breakpoint was set in the code to stop the program when 8 bits have been received, and the contents of the shift register were checked to verify that these waveforms were being correctly decoded.

4.2.2.2 Maximum Decoding Speed

The Manchester decoding software was run with a breakpoint at the end of the high priority interrupt service routine, and the number of cycles required to perform various decoding operations was calculated by examining the values of TMR3 and CCPR1.

Table 4.4 - Instruction Cycles for Decoding Operations

Operation	**Instruction Cycles**
Decode Clock Chip	41
Decode Data Chip	39
Decode Data Chip + Save Byte	45
Detect SOF	Up to 20
Process a byte (DL Layer)	Up to 30

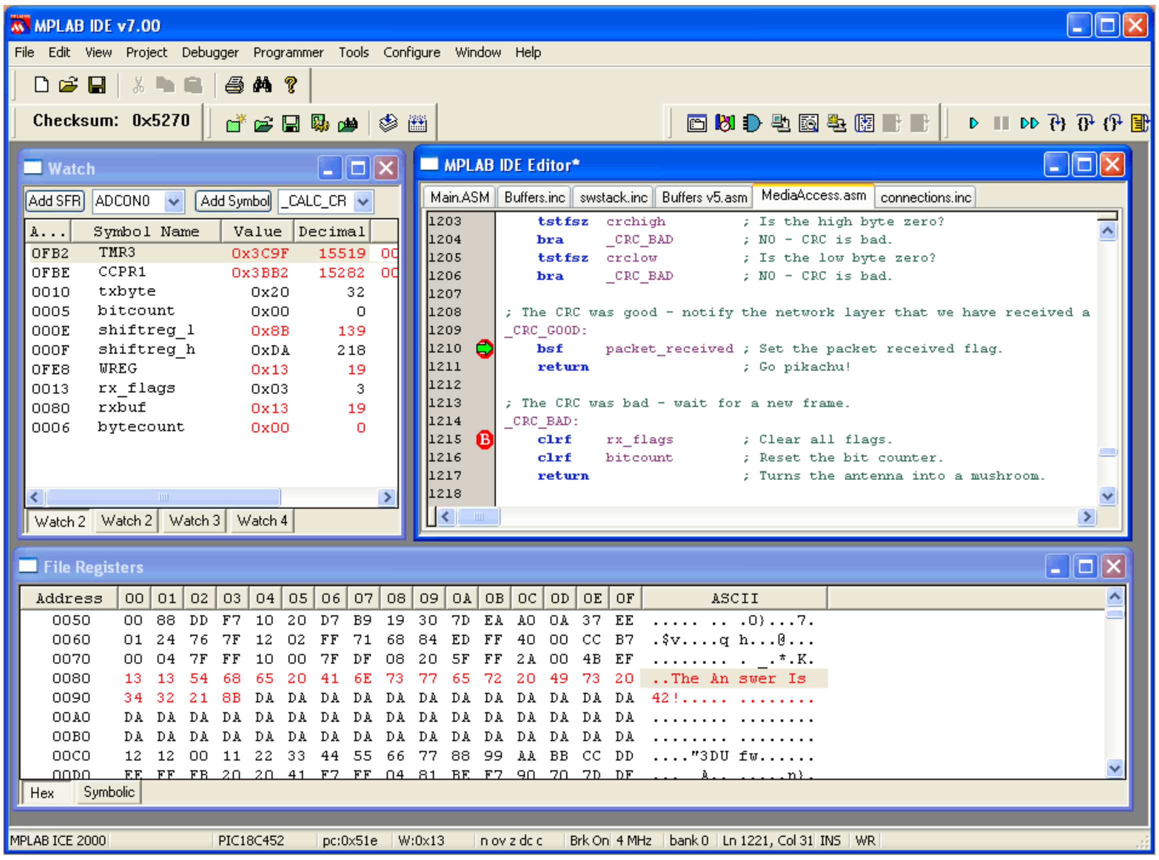
Detecting the SOF involves decoding one clock chip, one data chip, and running the SOF detect routine – a total of up to 100 cycles, or 50 cycles per chip.

Decoding one byte involves decoding 8 clock chips, 8 data chips, saving one byte, and processing one byte – a total of 676 cycles, or 42.25 cycles per chip.

Clearly in both cases, the decoder can comfortably run at 9600 baud, or 52 cycles per chip.

4.2.2.3 Packet Reception

For this experiment, a second node was set up to transmit a short packet containing a plain text message. The first node was set up to decode the message, and break when it had decoded the message and verified the CRC. A screenshot showing the received packet stored in RAM is presented in Figure 4.7.


MPLAB IDE v7.00
File Edit View Project Debugger Programmer Tools Configure Window Help
Checksum: 0x5270
Watch
Add SFR ADCON0 Add Symbol _CALC_CR
Watch 2 Watch 2 Watch 3 Watch 4
MPLAB IDE Editor*
Main.ASM Buffers.inc swstack.inc Buffers v5.asm MediaAccess.asm connections.inc
File Registers
Hex Symbolic

A...	Symbol Name	Value	Decimal
0FB2	TMR3	0x3C9F	15519
0FBE	CCPR1	0x3BB2	15282
0010	txbyte	0x20	32
0005	bitcount	0x00	0
000E	shiftreg_l	0x8B	139
000F	shiftreg_h	0xDA	218
0FE8	WREG	0x13	19
0013	rx_flags	0x03	3
0080	rxbuf	0x13	19
0006	bytecount	0x00	0

```
        tstfsz  crchigh         ; Is the high byte zero?
        bra     _CRC_BAD        ; NO - CRC is bad.
        tstfsz  crclow          ; Is the low byte zero?
        bra     _CRC_BAD        ; NO - CRC is bad.

; The CRC was good - notify the network layer that we have received a
_CRC_GOOD:
        bsf     packet_received ; Set the packet received flag.
        return                  ; Go pikachu!

; The CRC was bad - wait for a new frame.
_CRC_BAD:
        clrf    rx_flags        ; Clear all flags.
        clrf    bitcount        ; Reset the bit counter.
        return                  ; Turns the antenna into a mushroom.
```

```
Address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F        ASCII
  0050   00 88 DD F7 10 20 D7 B9 19 30 7D EA A0 0A 37 EE   ..... .. .0}...7.
  0060   01 24 76 7F 12 02 FF 71 68 84 ED FF 40 00 CC B7   .$v....q h...@...
  0070   00 04 7F FF 10 00 7F DF 08 20 5F FF 2A 00 4B EF   ........ . _.*.K.
  0080   13 13 54 68 65 20 41 6E 73 77 65 72 20 49 73 20   ..The An swer Is
  0090   34 32 21 8B DA DA DA DA DA DA DA DA DA DA DA DA   42!..... ........
  00A0   DA DA DA DA DA DA DA DA DA DA DA DA DA DA DA DA   ........ ........
  00B0   DA DA DA DA DA DA DA DA DA DA DA DA DA DA DA DA   ........ ........
  00C0   12 12 00 11 22 33 44 55 66 77 88 99 AA BB CC DD   ...."3DU fw......
  00D0   EE FF FB 20 20 41 F7 FF 04 81 BE F7 90 70 7D DF   ...  A.. .....p}.
```

```
MPLAB ICE 2000   PIC18C452   pc:0x51e   W:0x13   n ov z dc c   Brk On  4 MHz   bank 0   Ln 1221, Col 31  INS   WR
```

Figure 4.7 - Message successfully received (The answer is 42!)

4.2.3 Software Buffers

The popcorn buffering technique described in chapter 3 was not implemented due to time restrictions. However, the buffer data structures were implemented, and proved to be very handy for buffering incoming and outgoing data, as well as general purpose data storage.

4.3 MEDIA ACCESS CONTROL

4.3.1 Clear Channel Assessment

Clear channel assessment (CCA) is an essential function for both the preamble sampling and CSMA protocols at the MAC layer. As noted previously, the Carrier Detect line is unsuitable for this function as it is severely affected by noise. The proposed solution was to perform an analogue to digital conversion on the RSSI (received signal strength indicator) signal and compare it to the average noise floor.

Unfortunately, it was observed during testing that the noise generated by the ICE2000 caused the carrier detect to be continuously asserted, and the RSSI line to indicate a very strong signal. This was unexpected, as the PCB was designed to avoid interference between digital and analogue circuitry. However, further investigation showed that the interference had the similar effect even when the ICE2000 was completely electrically isolated from the transceiver.

Due to this problem, the "clock detect" flag of the Manchester decoder was used as a clear channel assessment function. As a result, the sampling period had to be increased to 10 ms to allow enough for the receiver to stabilise, and the decoder to acquire and verify the Manchester clock. Hopefully, this problem will not occur with an actual microprocessor, so the RSSI can be used for CCA and the sampling period reverted to 1.05 ms.

4.3.2 Preamble Sampling

The preamble sampling algorithm was implemented using TIMER0 of the PIC18C452 microprocessor, which allows a sleep interval of up to 16.777 seconds (With T_{cycle} = 1 μs and a 1:256 pre-scaler). The sleep interval and preamble length were stored as variables as RAM, which allowed them to be dynamically reconfigured. Two subroutines were implemented to reconfigure the sleep period and preamble length for high traffic and low traffic respectively. These subroutines, as well as an override flag were made accessible to the network layer.

As mentioned previously, the sampling period was increased to 10 ms, hence the power savings were remodelled with this modified parameter. The optimal sleep intervals for a very sparse network (n = 3) were found to be 55.6 ms for high traffic (f = 1), and 2.7746 seconds for low traffic (f = 1/1800).

The first experiment conducted was to validate the assumption that average power consumption of the X2010 transceiver is linearly related to the duty cycle. An ammeter was connected in series with the radio's power supply, and the average current consumption measured for various duty cycles with the sampling period fixed at 10 milliseconds. The results presented in Figure 4.8 confirm the assumption that power consumption is linearly related to the duty cycle. Note that there is some random variation for duty cycles between 1% and 5%, however this can be attributed to the precision of the multimeter used for the experiment.

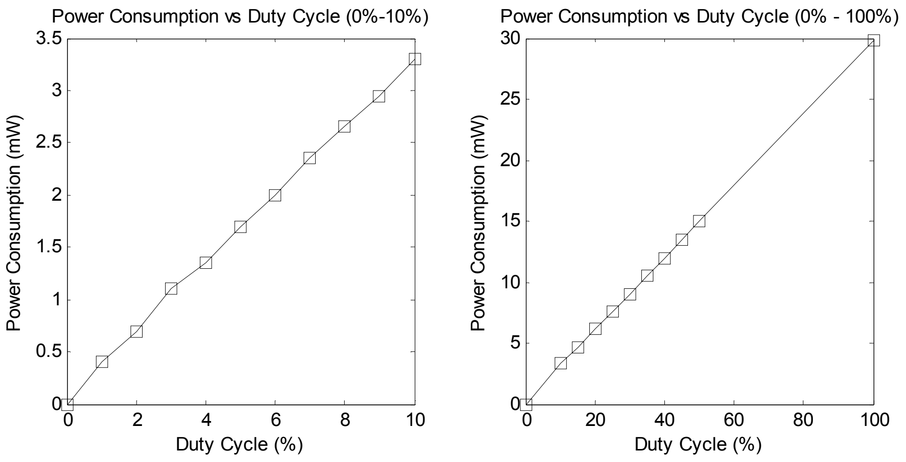


Figure 4.8 - Power Consumption vs. Duty Cycle for X2010 Transceiver

The second experiment was to test if the preamble detection mechanism was working correctly. A receiving node was set up with a sleep interval of five seconds, and a transmitting node was set up to transmit a 2.5 kHz square wave. When the transmitting node was switched on, the receiving node's RXEN (receive enable) light turned on within five seconds. The RXEN light remained on, indicating that the MAC layer had detected a valid preamble and was waiting for the start of frame sequence. When the transmitting node was turned off, the RXEN light immediately went out, indicating that the MAC layer had ceased to detect valid data, and hence switched off its receiver for the remainder of the sleep interval.

The third experiment was to verify that the preamble sampling algorithm does not affect link reliability. One node was set up to transmit a packet upon reset, with a preamble length of 200ms. A Second node was set up to receive these packets, with a sampling interval of 200ms. An indicator LED on the receiver was programmed to flash every time a packet was successfully received. Of twenty packets transmitted, every one was received successfully, indicating that preamble sampling does not affect link reliability.

4.3.3 Scheduled Reconfiguration

The DS1305 real time clock chip provides two time-of-day alarms, with corresponding IRQ lines. For the purpose of testing, the alarms were configured to go off every thirty seconds, and switch the node to high traffic mode for ten seconds. The receive enable LED was observed to visibly change frequency for ten seconds every thirty seconds, indicating that this simple schedule was working properly.

4.3.4 Dynamic Reconfiguration

In order to test the basic concept, a reconfiguration protocol similar to BECA [11] was implemented, which does not require any interaction with the network layer. When a node transmits a packet, it switches to high traffic mode until a timeout expires, assuming that neighbouring nodes which received the packet will all be in high traffic mode for the same period. When a node successfully receives a packet, it samples at the high traffic rate until a timeout expires. However, *it does not change its preamble length* until it transmits a packet itself, as not all of its neighbours may be in high traffic mode.

In order to allow this protocol to operate alongside the scheduled reconfiguration protocol, neither protocol was allowed to directly reconfigure the preamble sampling protocol. Instead, both protocols use flags to indicate their state, and additional logic was added to set the configuration based on the state of these flags.

This protocol was tested with two nodes as proof of concept. Two nodes were set up to bounce a packet between each other ten times with a one second delay, and one node was set up to transmit the initial packet on reset. The high traffic mode timeout was set at five seconds. The initial packet was observed to take several seconds to transmit because of the long preamble, whilst the remaining transmissions resulted in only a brief flash of the transmit LED. Both nodes were observed to remain in high traffic mode for five seconds after the last transmission, as indicated by rapid flashing of the receive enable LED.

This protocol is fairly effective, as it allows all the nodes responding to a broadcast request to save power by using the shorter preamble. However, it lacks intelligence, as nodes within a one hop radius of any network activity will unnecessarily switch to high traffic mode, and nodes will remain in high traffic even if they are not expecting any further activity. It can be observed that this simple protocol would only be effective if transmission consumes much more power than listening.

This protocol was then adapted to be use information from the network layer, with the packet transmission subroutine modified to allow the preamble length to be overridden on a per-packet basis. Unfortunately, this could not be tested, as the network layer implementation was not completed.

4.4 ADDRESS ALLOCATION PROTOCOL

4.4.1 Simulation

Sloots noted that testing and evaluating his routing protocol with only two physical nodes was a time consuming procedure, as it required that the nodes' state was manually changed before each hop [2]. This was sufficient to demonstrate that the protocol was functional; however it did not allow the performance of the protocol to be qualitatively measured.

The adaptive preamble sampling protocol and synchronisation mechanisms can be tested and evaluated with only two physical nodes, as these protocols only operate at the data link layer. However, this does not provide any indication of how well the protocol performs in a real network running a real application.

The addressing protocol could be tested by the same procedure used by Sloots, however this would provide no means of determining whether the modifications made to [23] actually offer power savings in a particular scenario.

The only real solution to this problem is software simulation, and in fact all of the reviewed research into new protocols included simulation results. The ubiquitous simulation tool for this area of research is ns-2 [40], although some researchers have used GloMoSim [41]. Ns-2 was chosen for this project, purely because it is more popular.

4.4.2 NS-2 Simulation Environment

NS-2 [40] is an object-oriented, discrete event driven network simulator developed written in C++ and OTcl. It is currently developed by the VINT project at the University of California, Berkeley. It was originally only capable of simulating wired IP networks. However, the Rice University Monarch (**Mo**bile **N**etwork **Arch**itecture) Extensions [42] add wireless capabilities, and have since been assimilated into the main ns-2 build.

The simulator consists of an event scheduler, and a library of network component objects, both of which are written in C++. Therefore, any new network component objects such as new protocols must be implemented in C++. Adding new modules requires that the ns-2 binary be recompiled from source. OTcl (object oriented Tcl) is used to build a simulation scenario using the precompiled network component objects.

The ns-2 network simulator is designed to run under a UNIX/LINUX environment, thus the CYGWIN environment is required to compile and run the simulator under Microsoft Windows. The simulator could not be compiled until it was realised that some additional CYGWIN packages were required. Some of the other modules refused to compile for unknown reasons, however binary executables were easily obtained for these modules.

4.4.3 Design

There are two main entities of concern in the address allocation protocol: nodes and packets. The following fragments of C++ code define the properties of these entities:

Table 4.5 - Definition of Packet

```
/* Packet Types:
 * AREQ = Address Request
 * AREP = Address Reply (Offer)
 * NREP = Negative Reply
 * AACK = Address Accept
 * PACK = Proxy Address Accept
 * ASRCH = Initiate Address Search
 */
enum {AREQ, AREP, NREP, AACK, AREJ, ASRCH};

struct packet {
    int daddr;              // destination address
    int saddr;              // source address
    int type;               // message type
    int seq;                // number of times this message has been sent
    int uid;                // unique ID of the requesting node
    int addrs[2];           // address space being allocated
    int route[10];          // route back to the requesting node
    int route_size;         // number of addresses in the route
}
```

Table 4.6 - Definition of Node

```
/* States:
 * INIT = Initialising
 * IDLE = Idle
 * ALLOC = Allocating addresses
 * PROXY = Acting as an allocation proxy
 */
enum {UNINIT, INIT, IDLE, ALLOC, PROXY};

class node {
    int state;                          // Node State
    int uid;                            // Unique identifier;
    int my_addr[2];                     // My Address Space

    // Variables for Initialisation
    int areq_retry_count;               // AREQ retry counter
    static const int AREQ_LIMIT;        // AREQ retry limit
    static const int AREQ_TIMEOUT;      // AREQ retry timeout
    int arep_recv_count;                // AREP received counter
    int nrep_recv_count;                // NREP received counter
    Packet* best_offer;                 // Best offer
    int best_offer_size;                // Size of the best offer

    // Variables for Allocation
    int arep_retry_count                // AREP retry counter
    static const int AREP_RETRY_LIMIT;  // AREP retry limit
    static const int AREP_TIMEOUT;      // AREP retry timeout;
    int under_alloc[2];                 // Allocated Address Space
    Packet* request_source;             // the AREQ/ASRCH we are replying to
}
```

The address allocation protocol is entirely event driven. Events may be asynchronous, e.g. the arrival of a packet, or scheduled, e.g. a timeout. Before the protocol was implemented in C++, it was designed using flowcharts, showing the sequence of actions which occur in response to each event. This approach is advantageous, as it provides a detailed design which can be implemented on any platform, in any programming language. The flowcharts thoroughly describe the control and data flow of each event, yet their graphical nature makes them easy to understand.

The important events for the protocol are:

1. Node startup.
2. Initialisation timeout.
3. Allocation timeout.
4. AREQ message received.
5. AREP message received.
6. NREP message received.
7. AACK message received.
8. AREJ message received.
9. ASRCH message received.

Appendix A contains the flow charts for each of these events.

4.4.4 Implementation

Both Mr LM Patnaik and PA Tayal [23] were contacted to try and obtain the source code they used for their simulation, as the new protocol is merely an adaptation of theirs. However Mr Patnaik could not be contacted, and Mr Tayal no longer had the source code in his possession.

The addressing protocol was partially implemented in C++ within the ns-2 framework. However, despite weeks of endeavouring, the protocol could not be made to work in the simulator. Unfortunately, development had to be terminated due to time restrictions.

The C++ code as written is listed in Appendix D.

Chapter 5

DISCUSSION OF RESULTS

5.1.1 Hardware Architecture

For this thesis project and Steven Sloots' thesis project, the Microchip ICE2000 in-circuit emulator was used as the microcontroller development platform. It was observed that the RF noise generated by the ICE2000 interfered with the RSSI and CD readings of the radio transceiver. There were also more practical problems, such as requiring a dedicated ICE2000 and PC for each node being tested. None of the ICE2000 specific features were used during development, hence a better solution would be to use a Microchip ICD2 in-circuit debugger with a DIP-40 PIC18F452 device installed on the PCB. This would allow the node to be run independently of the development hardware; however the node could still be used with the ICE2000 via a PLCC-44 to DIP-40 adaptor board.

The ADG918/919 wideband CMOS switch performed excellently, with vast power savings over the previously used bypass relay. However, the off separation could be improved by better RF design. In particular, all RF components (or their carrier boards) should be soldered directly to the PCB.

5.1.2 Manchester Encoding and Decoding

The old Manchester encoding and decoding algorithms were only able to operate at 2400bps [2]. In contrast, the new algorithms perform the same tasks using 75% less instructions cycles, and thus operate comfortably at 9600bps. Encoding and decoding are facilitated by compare and capture interrupts, hence the time at which an event occurs is not dependent on the time it is processed, and vice versa. Therefore, there is some slack in the system, which allows data link layer processing tasks to also be executed under interrupt alongside the encoding and decoding tasks.

The decoding algorithm synchronises to the encoded clock in the Manchester signal, which provides much faster and more reliable synchronisation than using start and stop bits, and eliminates the associated 25% bandwidth overhead. In fact, as shown in Figure 4.6, the clock can drift by about 8% in either direction before synchronisation is lost. In theory, synchronisation should still be possible with a much larger drift, but the clock detect mechanism restricts the allowable drift to ensure the integrity of the data.

In addition to the 75% saving in instruction cycles, every time the data bit changes, the processor is freed up for an entire chip period, effectively halving the instruction cycles required to calculate that bit. Assuming that for random data, there is a 50% chance of this occurring on each bit, the average number of instruction cycles used will reduced by another 25%, resulting in a total saving of 81.25% for the encoding algorithm. This could be exploited by increasing the transmission rate, which reduces channel utilization and hence the probability of collisions. Alternatively, the processor can use the spare instruction cycles to concurrently perform data processing and communication, which is made possible by the portable data buffer structures developed. During any remaining instruction cycles, the microprocessor can be powered down, resulting in significant power savings. To enable this, the chip rate generator (TIMER1) must be clocked from an internal RC oscillator, so it can continue to run whilst the microprocessor is powered down. Whilst this reduces the accuracy of the baud rate, this is not a problem since the decoder can comfortably tolerate a clock drift of up to 8%.

Another advantage of using Manchester encoding is that receiver can determine the data rate being used by measuring the frequency of the preamble. This allows the data rate to by dynamically varied by the transmitter, perhaps to adapt to changes in the link quality.

5.1.3 MAC Protocol

The adaptive preamble sampling protocol was implemented and demonstrated to be functional, and provides significant power savings for idle nodes' radios. It was demonstrated through modelling that these power savings are up to 99% for a very low traffic network. However, the model used to obtain these results does not include the adaptive reconfiguration, nor does it consider the length of each packet or the energy wasted as a result of collisions. Software simulation is required to more accurately determine the power savings that can be provided by this technique.

5.1.4 Address Allocation Protocol

An innovative addressing protocol was developed and based on [23, 31], with modifications predicted to reduce power consumption. Some of the modifications were to eliminate redundant transmissions, which will save a finite amount of power with no impact on the operation of the protocol. However, the other modifications were to promote more even distribution of addresses throughout the network, and to reduce unnecessary flooding.

This protocol was partially implemented in ns-2, however unfortunately could not be tested. However, a logical analysis can be used to observe the following points:

1. Most address allocation operations should take place without the need for a global address search, thus the majority of operations will occur locally. Ideally, the number of messages required to complete the operation is 1 + 2n, where n is the number of neighbouring nodes. However, by eliminating the rejection messages, the number of messages is reduced to 2 + n.
2. Promoting a more even distribution of addresses reduces the probability of address depletion. When a global address search does occur, waiting for the largest offer instead of accepting the first offer brings more addresses to the area of the network where the depletion occurred.

Clearly, a software simulation is required to better understand the dynamics of this protocol.

Chapter 6

CONCLUSION

The original goals of this thesis project were to improve the hardware platform, develop and implement a dynamic address allocation protocol, and to develop and implement a power saving media access protocol.

The improvements to the hardware platform were very successful. The bypass relay was replaced with a solid state switch which consumes virtually no power, which significantly reduces the energy cost of transmission. Improved Manchester encoding and decoding techniques were implemented, which provide much more reliable synchronisation, zero clock drift, eliminate the need for start and stop bits, and require 75% less instruction cycles. An efficient clock detection technique was also developed, which adds another mechanism to detect whether a valid signal is present.

An extensive literature review was carried out into dynamic address allocation techniques, and it was decided that the address space concept was most suitable for this project. Two existing protocols were logically analysed, combined, and improved upon to create an energy efficient, low maintenance addressing protocol. It was quickly realised that the only way to quantitatively analyse this protocol was through software simulation, thus it was decided to implement and test the protocol in the ns-2 simulator. Unfortunately, this was only partially completed due to technical problems and time constraints.

Existing power saving techniques were also researched, and it was decided that a scheduled rendezvous several times a day would be best suited to the likely traffic patterns for a data logging application. Several techniques were developed and implemented to ensure that the schedule is synchronised across the entire network. A preamble sampling technique is employed between the rendezvous periods to preserve network connectivity. This technique was modelled, which showed that there is an optimum sleep interval for a given network density and traffic rate, thus adaptive techniques are used to dynamically optimise the sleep interval for the anticipated traffic level. The protocol was successfully implemented on the microcontroller hardware platform, and its operation was verified. Unfortunately, one of the more advanced adaptive techniques relied on having a network layer present, and thus could not be tested.

Chapter 7

CONTINUATIONS AND EXTENSIONS

7.1 SOFTWARE SIMULATION

The power saving protocol developed in this thesis has been implemented and tested on the microcontroller hardware, and sufficient results have been obtained to conclude that the protocol is capable of providing significant power savings. However, there are no results which quantify the power savings for realistic network scenarios. Similarly, the new dynamic addressing protocol definitely provides power savings over its predecessors; however the exact improvement has not been quantified.

For this work to have any real credibility in the scientific community, these protocols must be simulated to provide quantitative results. Ns-2 seems to be the de facto standard for simulation of ad hoc networks, however experience has shown that very little software support is available, and a lot of the documentation currently available on the web is not fully up to date.

A more useful alternative may be to implement the protocols on Berkeley Motes and TinyOS, as the same code can be run on the actual mote, and in a simulator. This will also make the work more useful to Simon Willis' ad hoc networking research project, which is currently using Motes as the hardware platform.

7.2 LOCATION DISCOVERY

Because the nodes do not have MAC addresses and network addresses are dynamically assigned, the nodes have no hard coded identification number. Consequently, once the nodes have been deployed, the only way to tell which node is which is with application layer support.

Other ad hoc networks face a similar problem where the nodes are not deployed to known locations, for example if an area is blanketed with nodes for surveillance. An obvious solution is to attach a GPS module to each node, however this is costly, bulky, and power consuming. An alternative is to use a *distributed location discovery algorithm* [43-50], which uses triangulation to calculate the relative positions of each node, given there are three nodes whose absolute positions are known.

The triangulation algorithm typically uses the received signal strength indicator (RSSI) as a measure of the distance between nodes. Unfortunately, in a sparse outdoor ad hoc network, problems such as localised geography, multi-path propagation, and weather conditions will likely make this measurement unsuitable. However, the long distances between nodes make it viable to use propagation delay as a measure of distance. Assuming that a 20 MHz PIC microcontroller can measure propagation delay to a precision of 200 nanoseconds [2], distance can be calculated to a precision of 60m.

This is clearly only possible in ad hoc networks with large distances between nodes, thus it is a unique possibility for this project, and should be investigated further in a subsequent thesis.

[2] A 20 MHz PIC microcontroller executes instructions at 5 MHz, or 200 ns per instruction.

REFERENCES

[1] N. Sim, "Ad Hoc Sensor Network For Reef Monitoring System." Undergraduate Thesis, School of Engineering, James Cook University, Townsville, October 2003, pp. 116.

[2] S. Sloots, "Ad Hoc Radio Networks." Undergraduate Thesis, School of Engineering, James Cook University, Townsville, October 2004, pp. 216.

[3] L. M. Feeney and M. Nilsson, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment," in *Proc. IEEE INFOCOM*, April 2001, pp. 1548-1557.

[4] *Data Sheet: X2010 High Integrity FM Transceiver*: MK Radios, Revision 4.

[5] R. Zheng, J. C. Hou, and L. Sha, "Asynchronous wakeup for ad hoc networks ," in *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking \& computing* Annapolis, Maryland, USA ACM Press, 2003 pp. 35-45.

[6] T. D. Todd and M. Nosovic, "Low Power Rendezvous and RFID Wakeup for Embedded Wireless Networks," presented at 15th Annual IEEE Computer Communications Workshop (CCW 2000), Captiva Island, Florida, October 2000.

[7] E. Shih, S.-H. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, "Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking* Rome, Italy ACM Press, 2001 pp. 272-287.

[8] "IEEE Computer Society LAN MAN Standards Committee, IEEE 802.11 Standard: Wireless LAN Medium Access Control and Physical Layer Specifications," Aug. 1999.

[9] L. M. Feeney, "A QoS Aware Power Save Protocol for Wireless Ad Hoc Networks," in *First Mediterranean Workshop on Ad Hoc Networks(Med-Hoc Net 2002)*. Sardenga, Italy, September 2002.

[10] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," *Wirel. Netw.* , vol. 8 pp. 481-494 2002.

[11] Y. Xu, J. Heidemann, and D. Estrin, "Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks," USC/Information Sciences Institute 527, October 2000.

[12] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems* Baltimore, MD, USA ACM Press, 2004 pp. 95-107.

[13] A. El-Hoiydi, "Aloha with preamble sampling for sporadic traffic in ad hoc wireless sensor networks," P. o. I. I. C. o. Communcations, Ed., April 2002.

[14] A. El-Hoiydi, J.-D. Decotignie, C. Enz, and E. L. Roux, "wiseMAC, an ultra low power MAC protocol for the wiseNET wireless sensor network," in *Proceedings of the 1st international conference on Embedded networked sensor systems* Los Angeles, California, USA ACM Press, 2003 pp. 302-303.

[15] J. L. Hill and D. E. Culler, "Mica: A Wireless Platform for Deeply Embedded Networks," *IEEE Micro* vol. 22 pp. 12-24 2002.

[16] N. Abramson, "The Aloha System - Another Alternative for Computer Communications," in *AFIPS Conference Proceedings*, vol. 36, 1970, pp. 295-298.

[17] Y. Sun and E. M. Belding-Royer, "A Study of Dynamic Addressing Techniques in Mobile Ad hoc Networks," *Wireless Communications and Mobile Computing*, vol. 4, pp. 315-329, 2004.

[18] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, "Protocols for self-organization of a wireless sensor network," *IEEE Personal Comm. Magazine*, vol. 7, pp. 16 - 27, 2000.

[19] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Commun. ACM* vol. 43 pp. 51-58 2000.

[20] J. Boleng, "Efficient network layer addressing for mobile ad hoc networks," The Colorado School of Mines, Technical Report MCS-00-09, June 2000.

[21] S. Cheshire, B. Aboba, and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses." IETF Internet Draft, http://www.ietf.org/rfc/rfc3927.txt, March 2005.

[22] C. E. Perkins, J. T. Malinen, R. Wakikawa, E. M. Royer, and Y. Sun, "Ad Hoc Address Autoconfiguration." IETF Internet Draft, draft-ietf-manet-autoconf-01.txt (Work In Progress), November 2001.

[23] A. P. Tayal and L. M. Patnaik, "An address assignment for the automatic configuration of mobile ad hoc networks," *Personal Ubiquitous Comput.* , vol. 8 pp. 47-54 2004.

[24] S. Nesargi and R. Prakash, "MANETconf: Configuration of Hosts in a Mobile Ad Hoc Network," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. New York, NY, June 2002.

[25] M. D. Yarvis, W. S. Conner, L. Krishnamurthy, J. Chhabra, B. Elliott, and A. Mainwaring, "Real-World Experiences with an Interactive Ad Hoc Sensor Network," in *Proceedings of the International Workshop on Ad Hoc Networking (IWAHN 2002)*. Vancouver, British Columbia, Canada, 2002.

[26] N. H. Vaidya, "Weak duplicate address detection in mobile ad hoc networks," in *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*. Lausanne, Switzerland: ACM Press, June 2002, pp. 206-216.

[27] Y. Sun and E. M. Belding-Royer, "Dynamic Address Configuration in Mobile Ad hoc Networks," University of California, Santa Barbara, UCSB Technical Report 2003-11 March 2003.

[28] R. Droms, "Dynamic Host Configuration Protocol." IETF Internet Draft, http://www.ietf.org/rfc/rfc2131.txt, March 1997.

[29] S. Toner and D. O'Mahony, "Self-Organising Node Address Management in Ad Hoc Networks," in *Personal Wireless Communications, IFIP-TC6 8th International Conference (PWC 2003) Proceedings*. Venice, Italy, 2003, pp. 476-483.

[30] P. Patchipulusu, "Dynamic Address Allocation Protocols for Mobile Ad Hoc Networks." Masters Thesis, Department of Computer Science: Texas A&M University, August 2001.

[31] Z. Hu and B. Li, "ZAL: Zero-Maintenance Address Allocation in Mobile Wireless Ad Hoc Networks," in *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*. Columbia, Ohio, 2005.

[32] *Data Sheet: Miniature Signal Relay ED2/EF2 Series*: NEC / TOKIN, 2004.

[33] *Data Sheet: ER400TRS "Easy Radio" Transceiever Module*. Low Power Radio Solutions, 2004.

[34] *Data Sheet: ADG918 - Wideband, 43 dB Isolation @ 1 GHz, CMOS 1.65 V to 2.75 V, 2:1 Mux/SPDT Switches*: Analog Devices, Revision A.

[35] *Data Sheet: DS1305 Serial Alarm Real-Time Clock*: Dallas Semiconductor, December 2002.

[36] Wikipedia, "OSI Model," http://en.wikipedia.org/wiki/OSI_model, Accessed 23 September 2005.

[37] *MPASM User's Guide with MPLINK and MPLIB*: Microchip Technology, 1999.

[38] *Data Sheet: AD8369 - 600 MHz, 45 dB Digitally Controlled Variable Gain Amplifier*: Analog Devices, Revision 0.

[39] *Data Sheet:PIC18CXX2 High Performance Microcontroller with 10-bit A/D*: Microchip, 2001.

[40] "The Network Simulator - ns-2," http://www.isi.edu/nsnam/ns/, Accessed 22 September 2005.

[41] "GloMoSim - Global Mobile Information Systems Simulation Library," http://pcl.cs.ucla.edu/projects/glomosim/, Accessed 22 September 2005.

[42] "The Rice University Monarch Project," http://www.monarch.cs.rice.edu/, Accessed 22 September 2005.

[43] P. Biswas and Y. Ye, "Semidefinite programming for ad hoc wireless sensor network localization," in *Proceedings of the third international symposium on Information processing in sensor networks* Berkeley, California, USA ACM Press, 2004 pp. 46-54.

[44] N. Bulusu, J. Heidemann, and D. Estrin, "GPS-less low cost outdoor localization for very small devices," *IEEE Personal Communications Magazine*, vol. 7, pp. 28-34, 2000.

[45] J. Hightower and G. Borriello, "Location Systems for Ubiquitous Computing," *Computer* vol. 34 pp. 57-66 2001.

[46] S. Meguerdichian, S. Slijepcevic, V. Karayan, and M. Potkonjak, "Localized algorithms in wireless ad-hoc networks: location discovery and sensor exposure," in *MOBIHOC 2001*, 2001, pp. 106-116.

[47] A. Nasipuri and K. Li, "A directionality based location discovery scheme for wireless sensor networks," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. Atlanta, Georgia, USA: ACM Press, 2002, pp. 105-111.

[48] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket location-support system," in *Proceedings of the 6th annual international conference on Mobile computing and networking* Boston, Massachusetts, United States ACM Press, 2000 pp. 32-43.

[49] C. Savarese, J. M. Rabaey, and J. Beutel, "Locationing in distributed ad hoc wireless sensor networks," in *Proc. 2001 Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP 2001)*: IEEE, Piscataway, NJ, May 2001.

[50] A. Savvides, C.-C. Han, and M. B. Strivastava, "Dynamic fine-grained localization in Ad-Hoc networks of sensors," in *Proceedings of the 7th annual international conference on Mobile computing and networking* Rome, Italy ACM Press, 2001 pp. 166-179.

Appendix A

SOFTWARE FLOW CHARTS

A.1 Encoding and Decoding

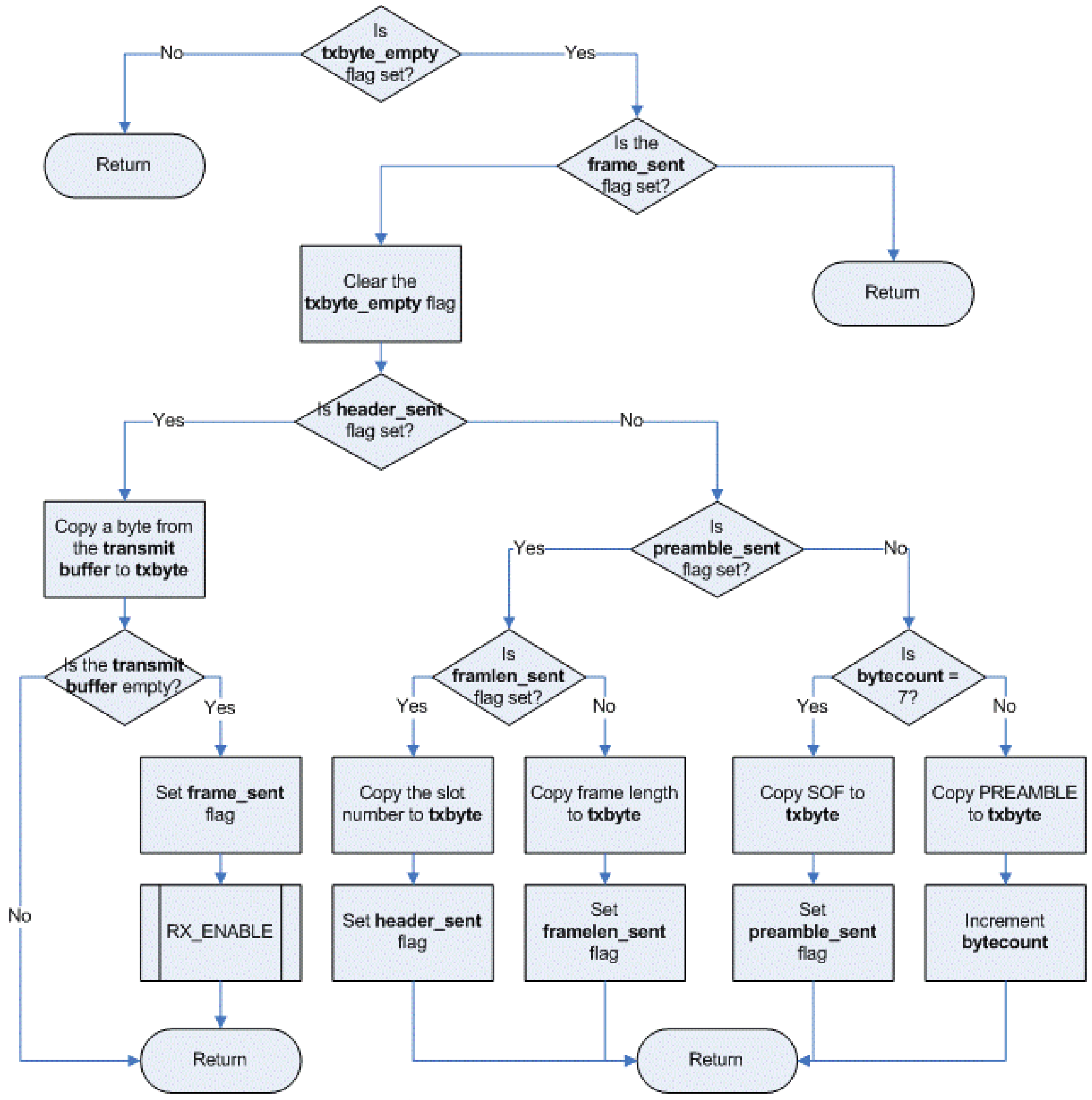

Is txbyte_empty flag set?
No
Yes
Return
Is the frame_sent flag set?
Clear the txbyte_empty flag
Return
Is header_sent flag set?
Yes
No
Copy a byte from the transmit buffer to txbyte
Is preamble_sent flag set?
Yes
No
Is the transmit buffer empty?
Yes
No
Is framlen_sent flag set?
Yes
No
Is bytecount = 7?
Yes
No
Set frame_sent flag
Copy the slot number to txbyte
Copy frame length to txbyte
Copy SOF to txbyte
Copy PREAMBLE to txbyte
RX_ENABLE
Set header_sent flag
Set framelen_sent flag
Set preamble_sent flag
Increment bytecount
Return
Return

Figure A.1 – Frame Encoder

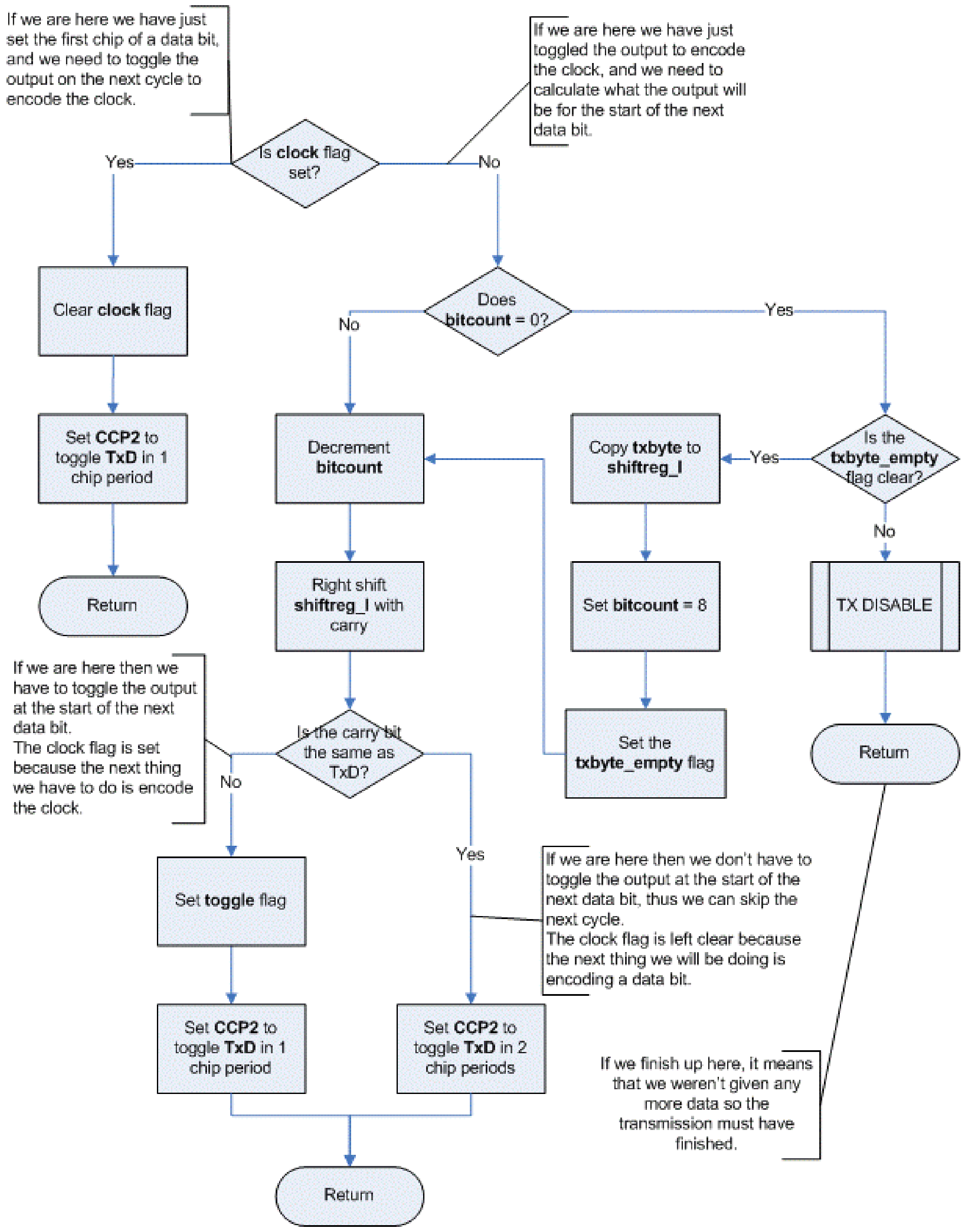

If we are here we have just set the first chip of a data bit, and we need to toggle the output on the next cycle to encode the clock.
If we are here we have just toggled the output to encode the clock, and we need to calculate what the output will be for the start of the next data bit.
Is clock flag set?
Yes
No
Clear clock flag
Does bitcount = 0?
No
Yes
Set CCP2 to toggle TxD in 1 chip period
Decrement bitcount
Copy txbyte to shiftreg_l
Yes
Is the txbyte_empty flag clear?
No
Return
Right shift shiftreg_l with carry
Set bitcount = 8
TX DISABLE
If we are here then we have to toggle the output at the start of the next data bit.
The clock flag is set because the next thing we have to do is encode the clock.
Is the carry bit the same as TxD?
No
Set the txbyte_empty flag
Return
Yes
If we are here then we don't have to toggle the output at the start of the next data bit, thus we can skip the next cycle.
The clock flag is left clear because the next thing we will be doing is encoding a data bit.
Set toggle flag
Set CCP2 to toggle TxD in 1 chip period
Set CCP2 to toggle TxD in 2 chip periods
If we finish up here, it means that we weren't given any more data so the transmission must have finished.
Return

Figure A.2 - Manchester Encoder

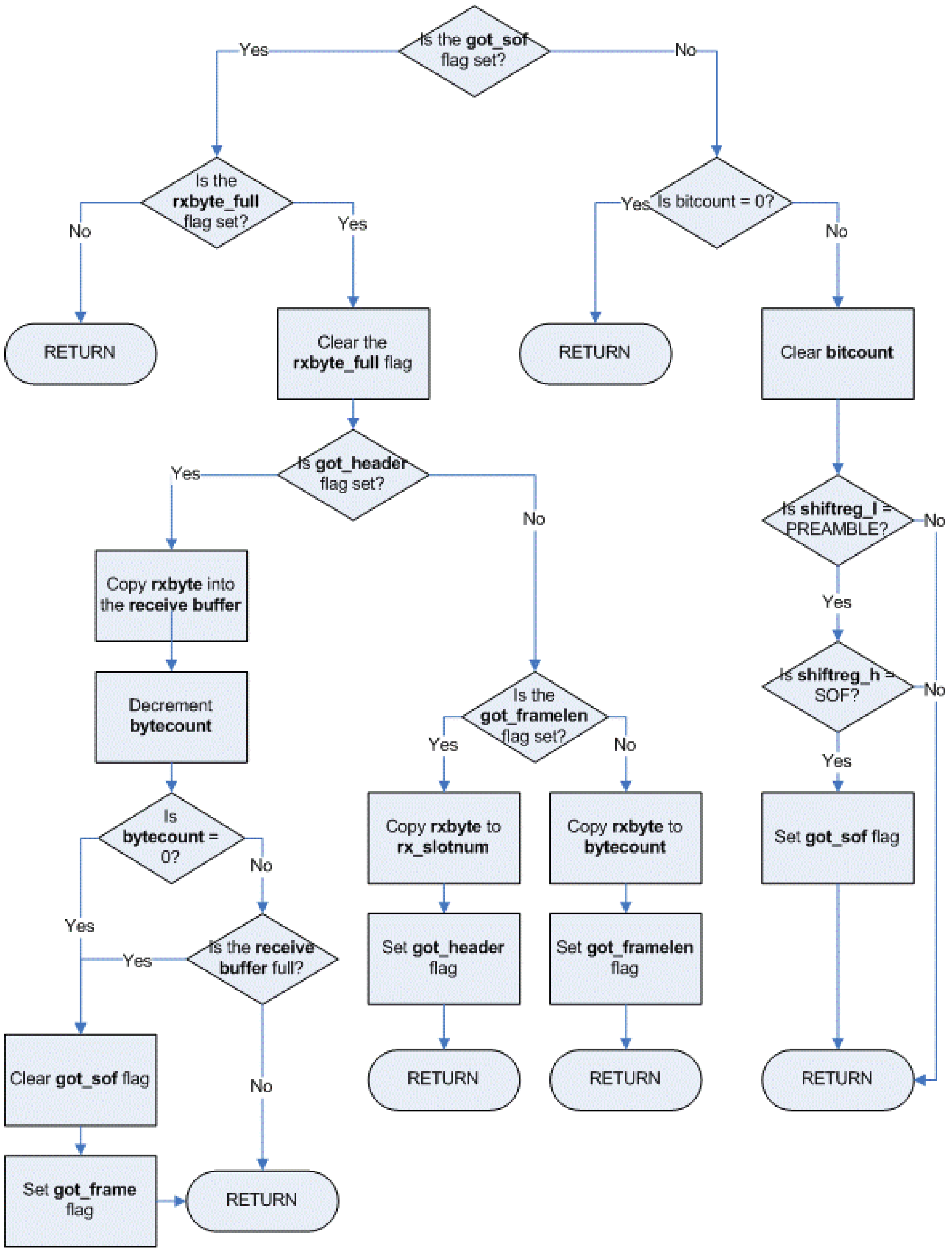


Figure A.3 - Frame Decoder

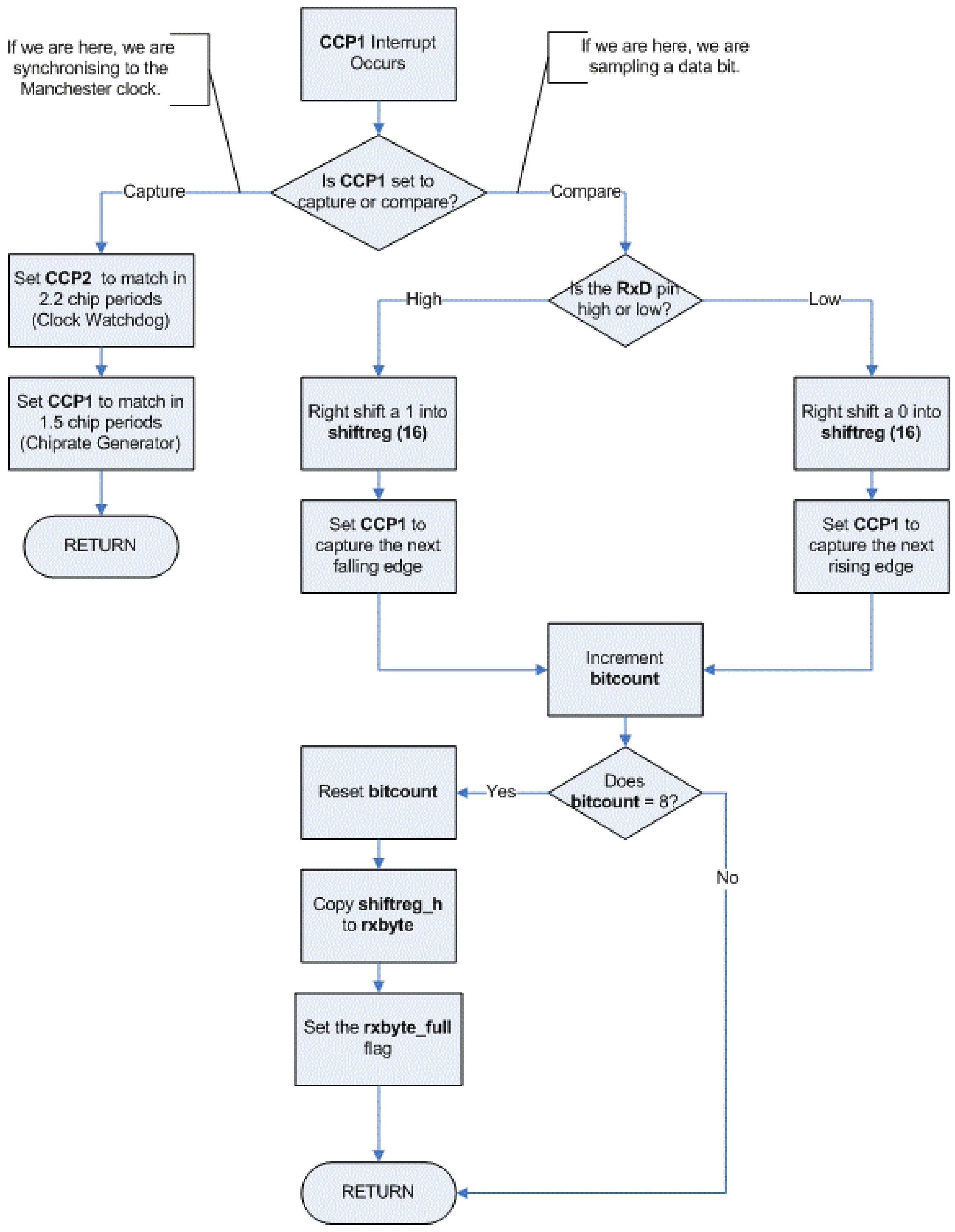

If we are here, we are synchronising to the Manchester clock.
CCP1 Interrupt Occurs
If we are here, we are sampling a data bit.
Capture
Is CCP1 set to capture or compare?
Compare
Set CCP2 to match in 2.2 chip periods (Clock Watchdog)
Is the RxD pin high or low?
High
Low
Set CCP1 to match in 1.5 chip periods (Chiprate Generator)
Right shift a 1 into shiftreg (16)
Right shift a 0 into shiftreg (16)
RETURN
Set CCP1 to capture the next falling edge
Set CCP1 to capture the next rising edge
Increment bitcount
Reset bitcount
Yes
Does bitcount = 8?
No
Copy shiftreg_h to rxbyte
Set the rxbyte_full flag
RETURN

Figure A.4 - Manchester Decoder

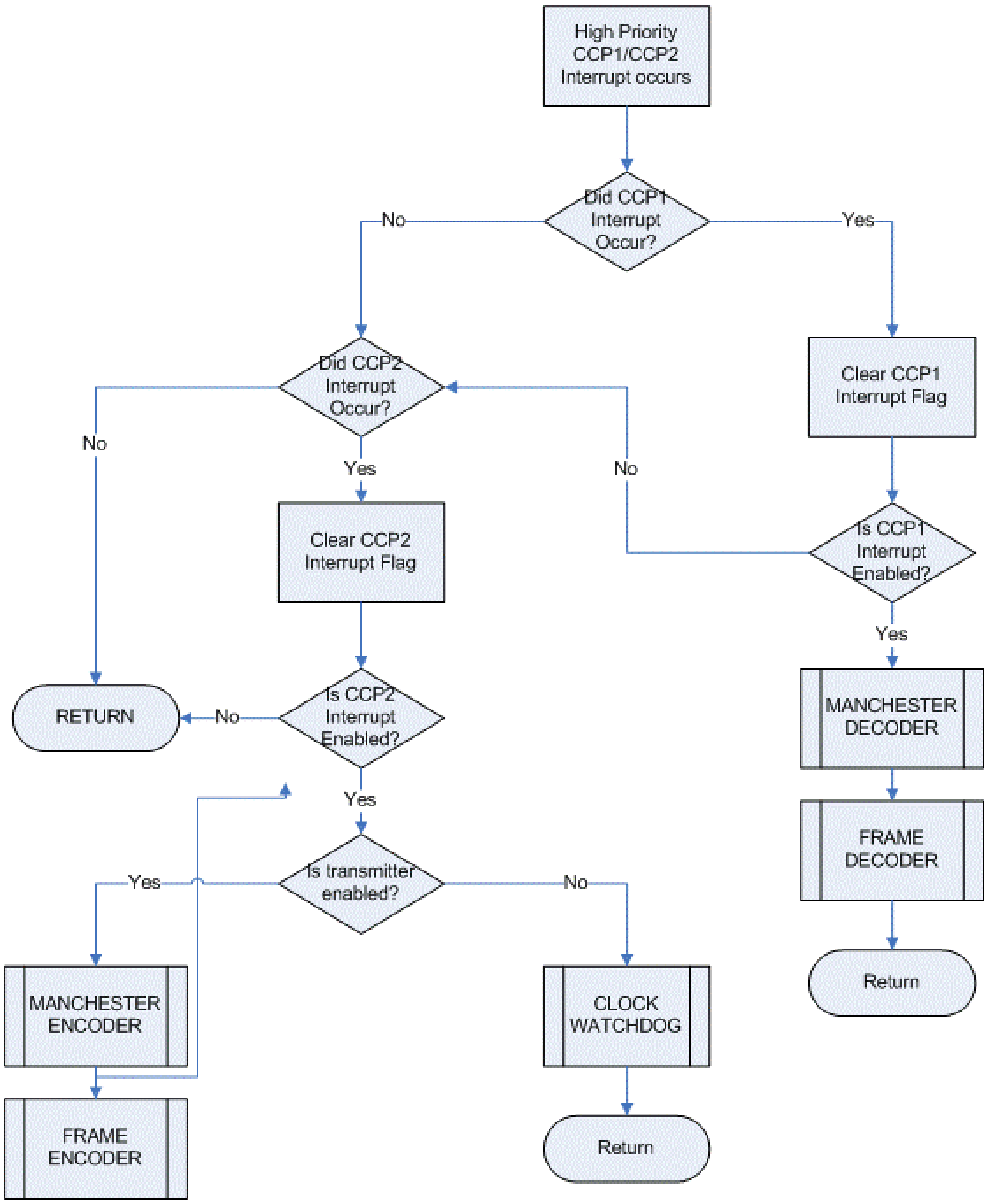

High Priority CCP1/CCP2 Interrupt occurs
Did CCP1 Interrupt Occur?
No
Yes
Clear CCP1 Interrupt Flag
Did CCP2 Interrupt Occur?
No
Yes
No
Clear CCP2 Interrupt Flag
Is CCP1 Interrupt Enabled?
Yes
RETURN
No
Is CCP2 Interrupt Enabled?
MANCHESTER DECODER
Yes
FRAME DECODER
Yes
Is transmitter enabled?
No
Return
MANCHESTER ENCODER
CLOCK WATCHDOG
FRAME ENCODER
Return

Figure A.5 - High Priority Interrupt Service Routine

A.2 ADDRESS ALLOCATION

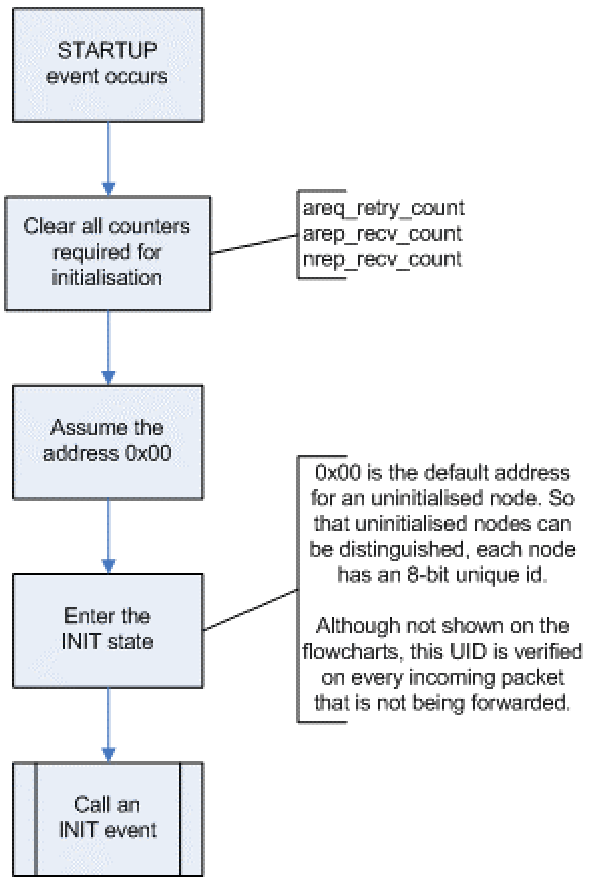

STARTUP event occurs
Clear all counters required for initialisation
areq_retry_count
arep_recv_count
nrep_recv_count
Assume the address 0x00
0x00 is the default address for an uninitialised node. So that uninitialised nodes can be distinguished, each node has an 8-bit unique id.
Although not shown on the flowcharts, this UID is verified on every incoming packet that is not being forwarded.
Enter the INIT state
Call an INIT event

Figure A.6 - Node Startup Event

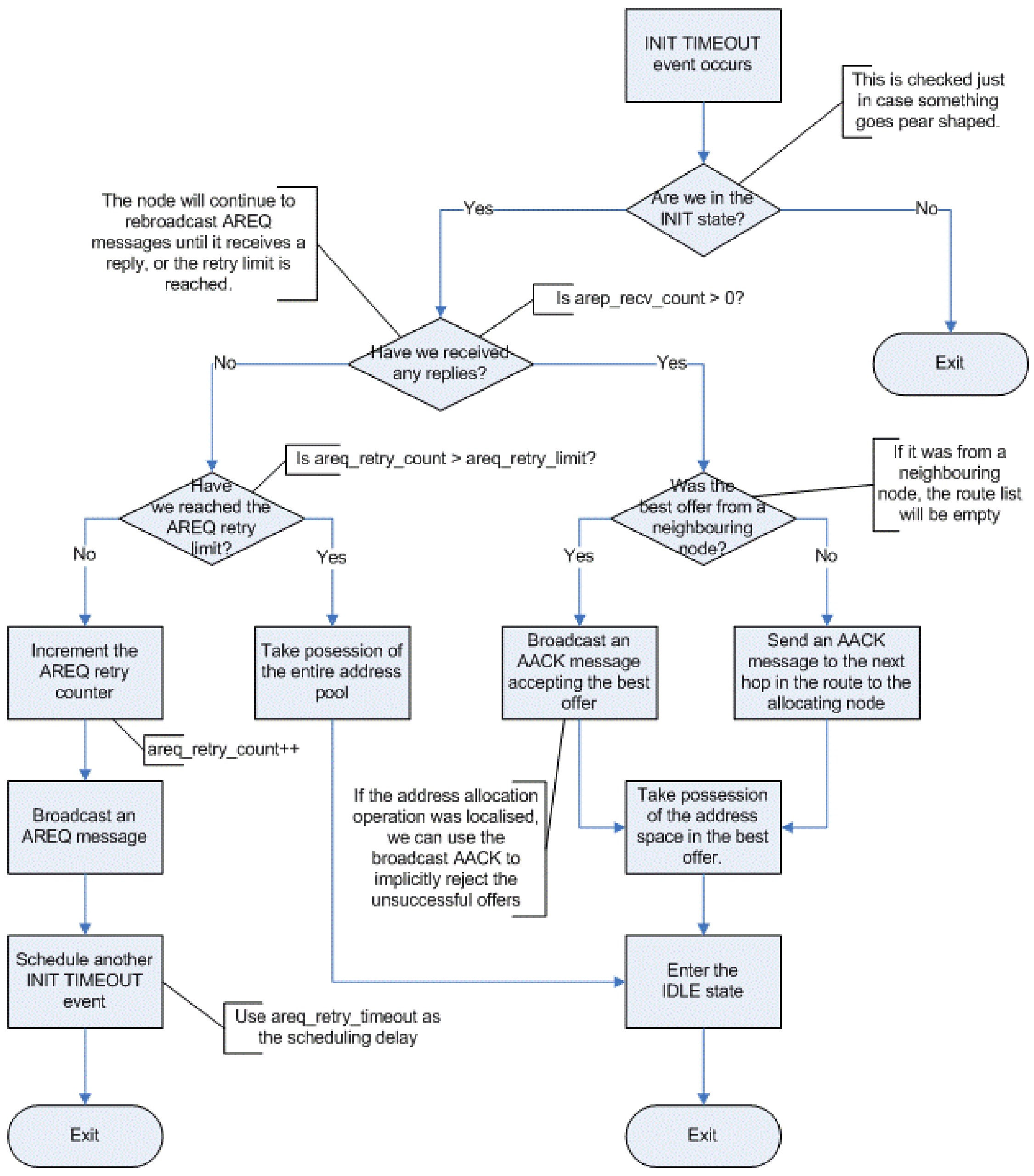

INIT TIMEOUT event occurs
This is checked just in case something goes pear shaped.
Are we in the INIT state?
Yes
No
Exit
The node will continue to rebroadcast AREQ messages until it receives a reply, or the retry limit is reached.
Is arep_recv_count > 0?
Have we received any replies?
No
Yes
Is areq_retry_count > areq_retry_limit?
Have we reached the AREQ retry limit?
No
Yes
Was the best offer from a neighbouring node?
If it was from a neighbouring node, the route list will be empty
Yes
No
Increment the AREQ retry counter
Take posession of the entire address pool
Broadcast an AACK message accepting the best offer
Send an AACK message to the next hop in the route to the allocating node
areq_retry_count++
Broadcast an AREQ message
If the address allocation operation was localised, we can use the broadcast AACK to implicitly reject the unsuccessful offers
Take possession of the address space in the best offer.
Schedule another INIT TIMEOUT event
Use areq_retry_timeout as the scheduling delay
Enter the IDLE state
Exit
Exit

Figure A.7 - Initialisation Timeout Event

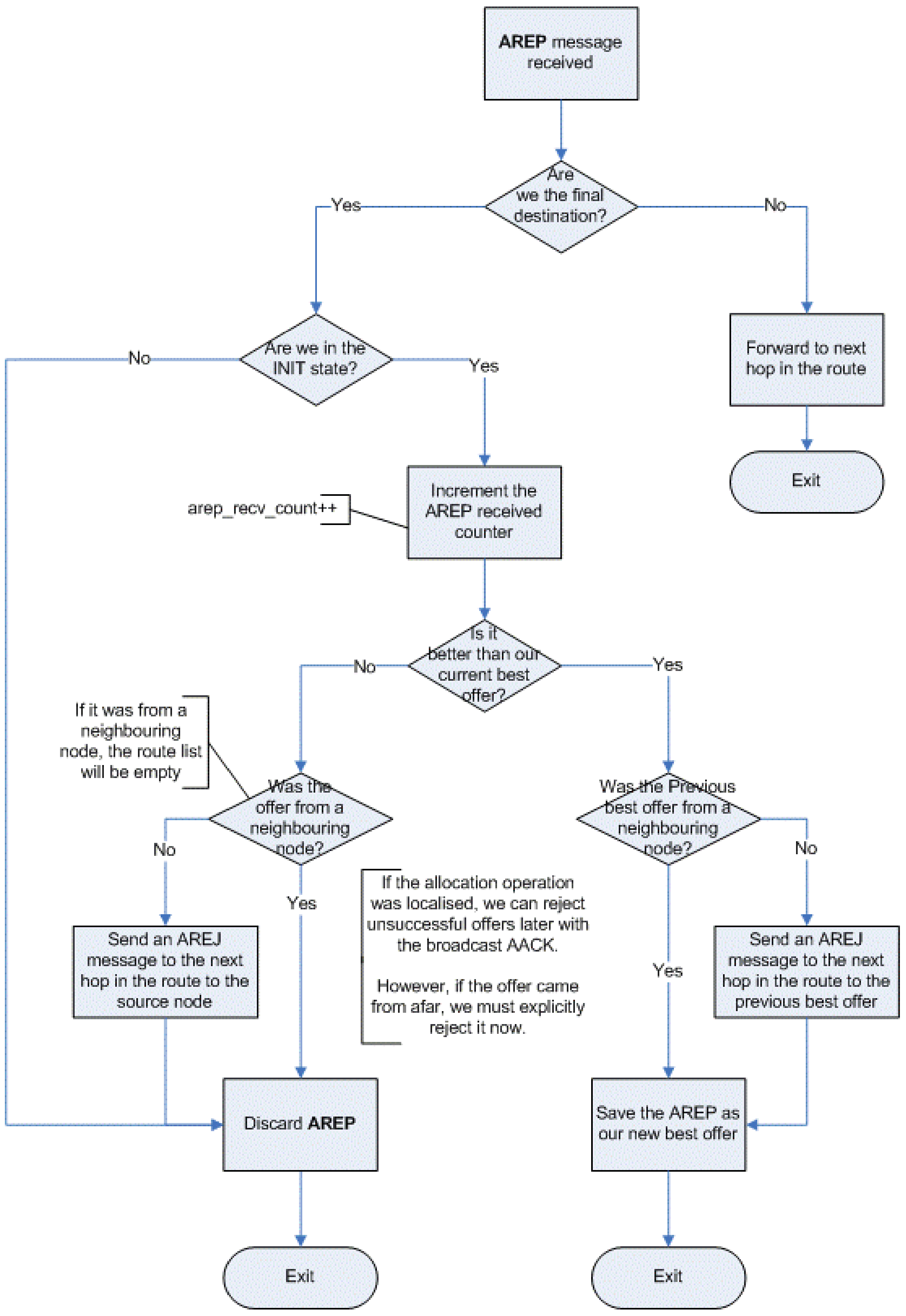

AREP message received
Are we the final destination?
Yes
No
Forward to next hop in the route
Exit
Are we in the INIT state?
No
Yes
Increment the AREP received counter
arep_recv_count++
Is it better than our current best offer?
No
Yes
If it was from a neighbouring node, the route list will be empty
Was the offer from a neighbouring node?
Was the Previous best offer from a neighbouring node?
No
Yes
Send an AREJ message to the next hop in the route to the source node
If the allocation operation was localised, we can reject unsuccessful offers later with the broadcast AACK.
However, if the offer came from afar, we must explicitly reject it now.
Yes
No
Send an AREJ message to the next hop in the route to the previous best offer
Discard AREP
Save the AREP as our new best offer
Exit
Exit

Figure A.8 - AREP Message Received Event

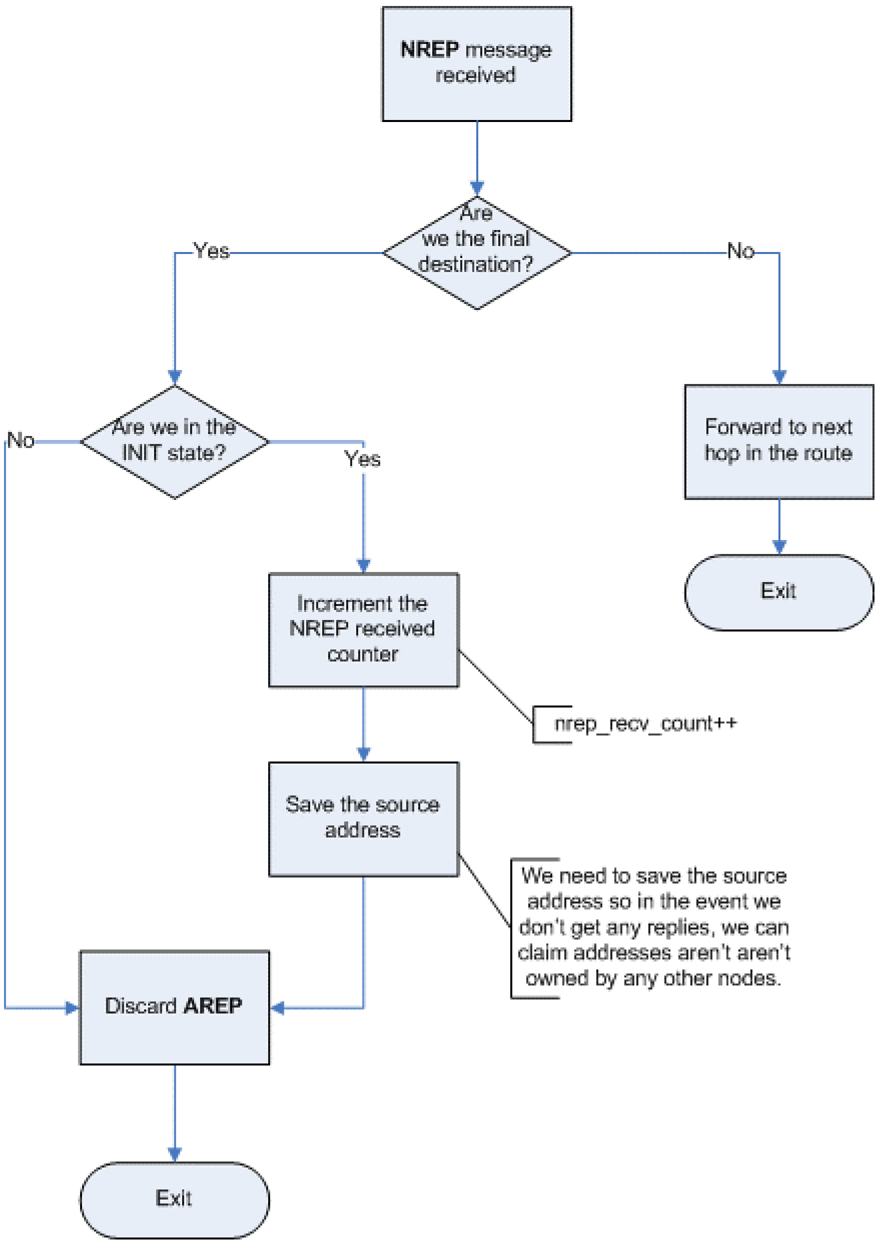

NREP message received
Are we the final destination?
Yes
No
Are we in the INIT state?
No
Yes
Forward to next hop in the route
Exit
Increment the NREP received counter
nrep_recv_count++
Save the source address
We need to save the source address so in the event we don't get any replies, we can claim addresses aren't aren't owned by any other nodes.
Discard AREP
Exit

Figure A.9 - NREP Message Received Event

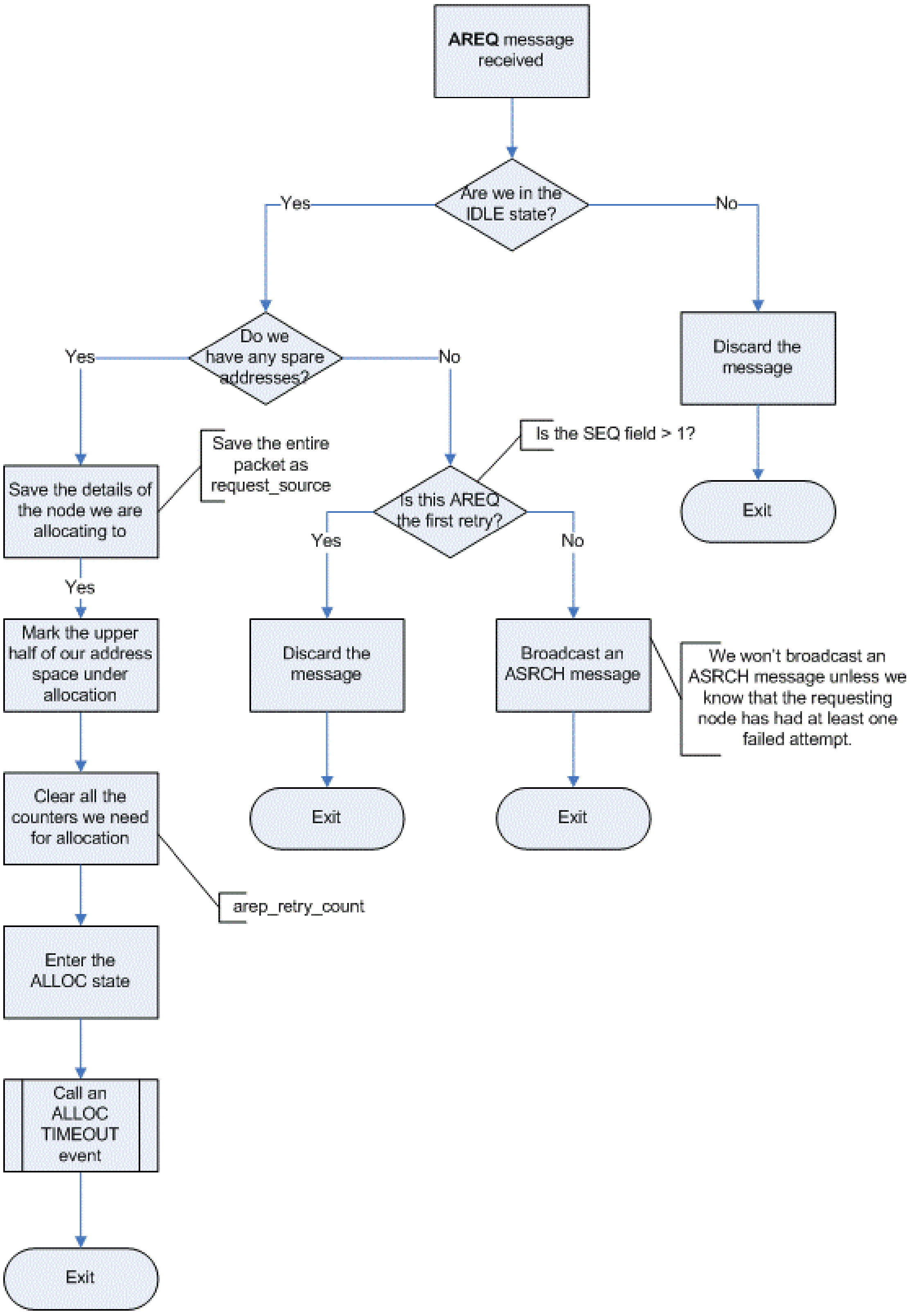

AREQ message received
Are we in the IDLE state?
Yes
No
Do we have any spare addresses?
Yes
No
Discard the message
Exit
Is the SEQ field > 1?
Save the entire packet as request_source
Save the details of the node we are allocating to
Is this AREQ the first retry?
Yes
No
Yes
Mark the upper half of our address space under allocation
Discard the message
Broadcast an ASRCH message
We won't broadcast an ASRCH message unless we know that the requesting node has had at least one failed attempt.
Clear all the counters we need for allocation
Exit
Exit
arep_retry_count
Enter the ALLOC state
Call an ALLOC TIMEOUT event
Exit

Figure A.10 - AREQ Message Received Event

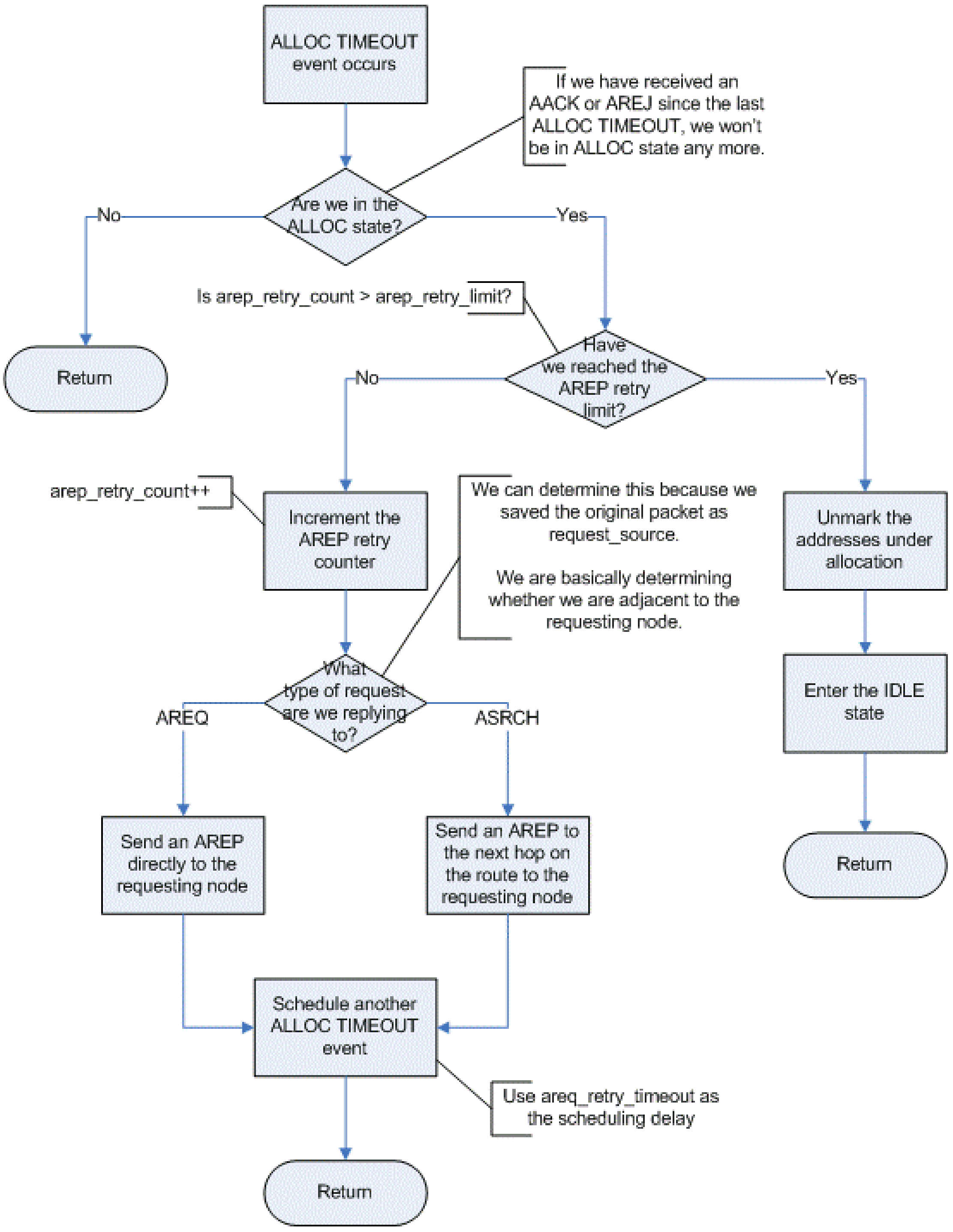

ALLOC TIMEOUT event occurs
If we have received an AACK or AREJ since the last ALLOC TIMEOUT, we won't be in ALLOC state any more.
Are we in the ALLOC state?
No
Yes
Return
Is arep_retry_count > arep_retry_limit?
Have we reached the AREP retry limit?
No
Yes
arep_retry_count++
Increment the AREP retry counter
We can determine this because we saved the original packet as request_source.
We are basically determining whether we are adjacent to the requesting node.
Unmark the addresses under allocation
Enter the IDLE state
What type of request are we replying to?
AREQ
ASRCH
Send an AREP directly to the requesting node
Send an AREP to the next hop on the route to the requesting node
Return
Schedule another ALLOC TIMEOUT event
Use areq_retry_timeout as the scheduling delay
Return

Figure A.11 - Allocation Timeout Event

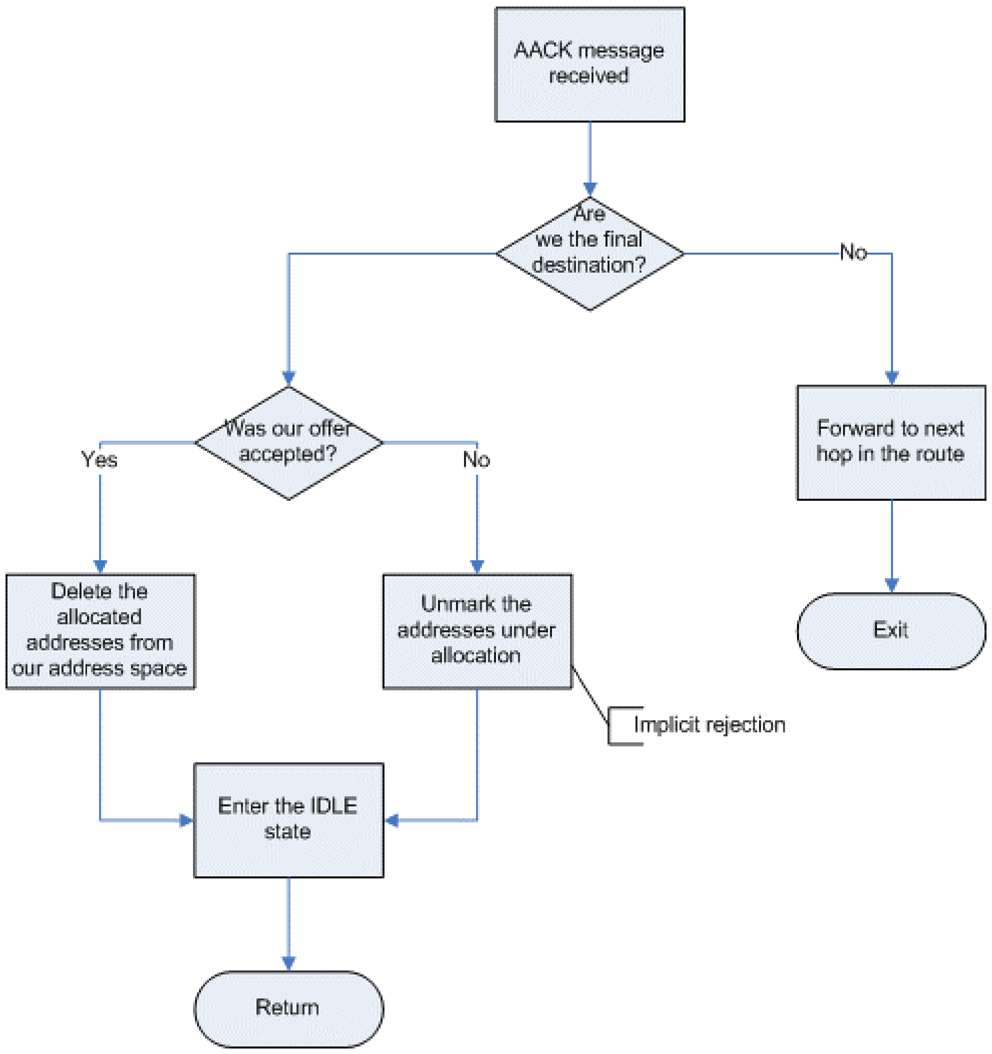

AACK message received
Are we the final destination?
No
Was our offer accepted?
Yes
No
Forward to next hop in the route
Delete the allocated addresses from our address space
Unmark the addresses under allocation
Exit
Implicit rejection
Enter the IDLE state
Return

Figure A.12 - AACK Message Received Event

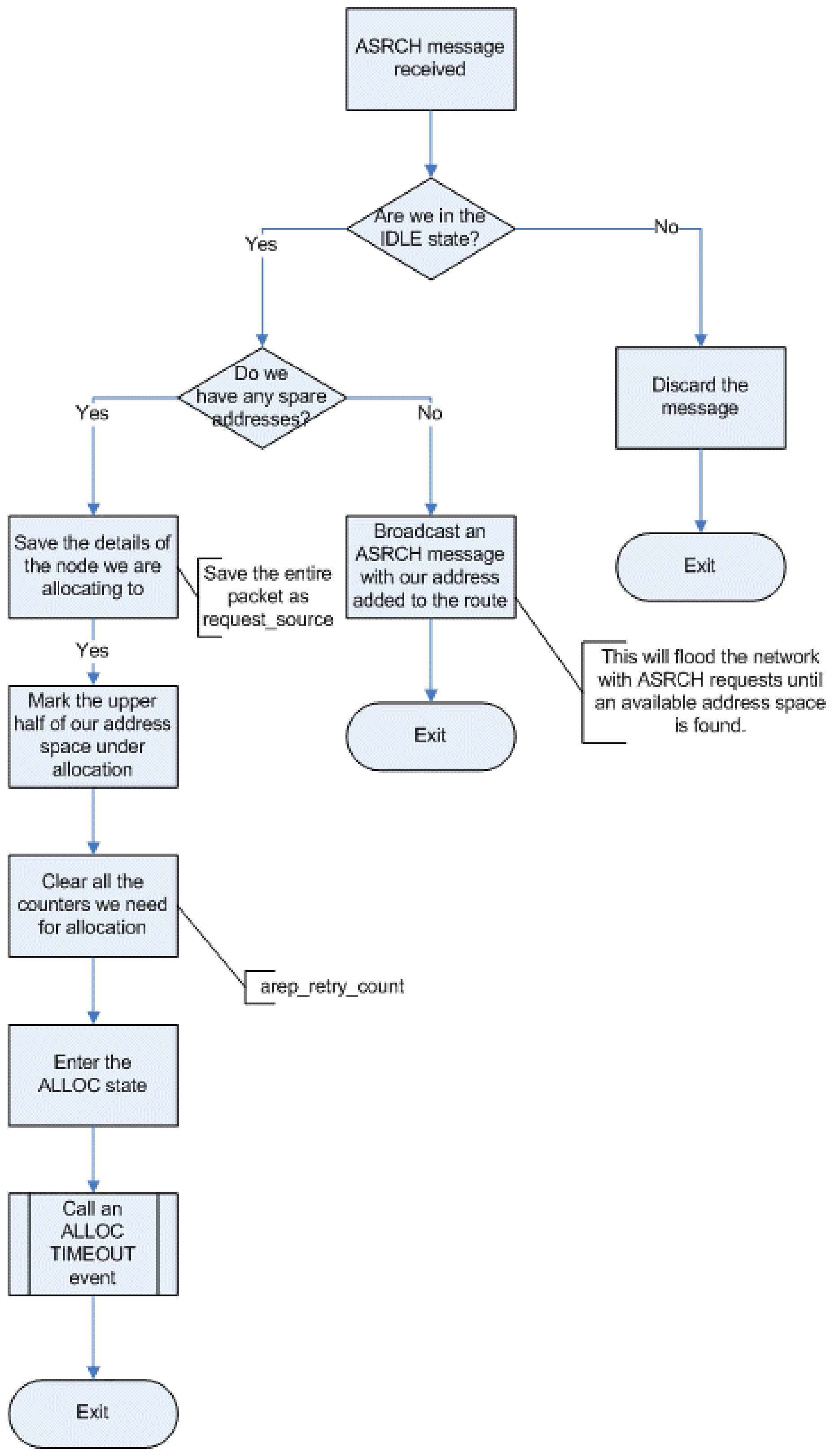

ASRCH message received
Are we in the IDLE state?
Yes
No
Do we have any spare addresses?
Discard the message
Yes
No
Save the details of the node we are allocating to
Save the entire packet as request_source
Broadcast an ASRCH message with our address added to the route
Exit
This will flood the network with ASRCH requests until an available address space is found.
Yes
Mark the upper half of our address space under allocation
Exit
Clear all the counters we need for allocation
arep_retry_count
Enter the ALLOC state
Call an ALLOC TIMEOUT event
Exit

Figure A.13 - ASRCH Message Received Event

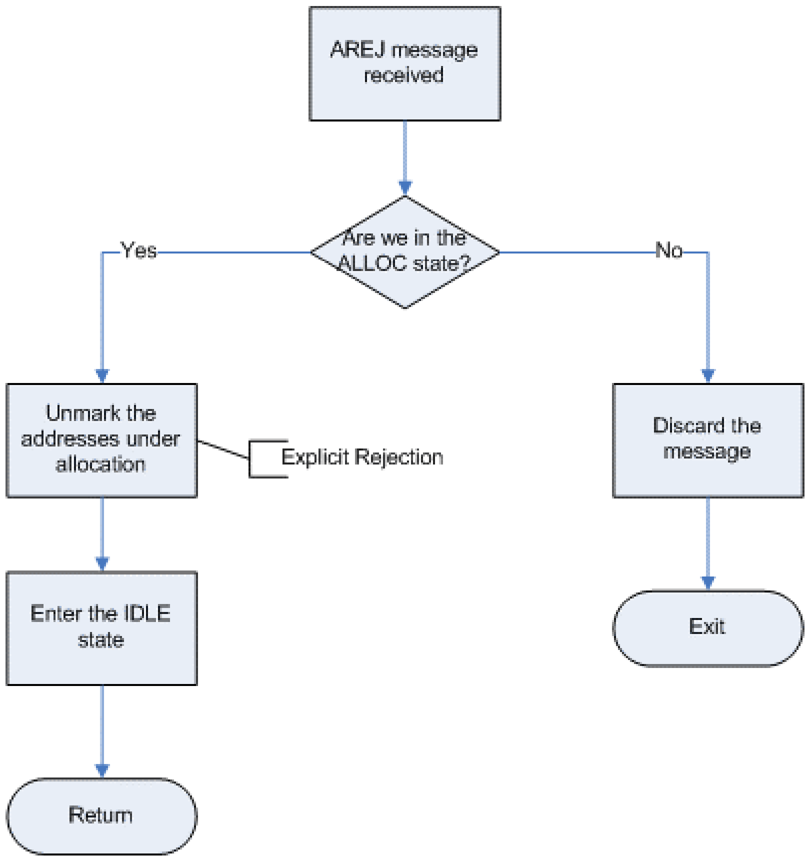


Figure A.14 - AREJ Message Received Event

Appendix B

SCHEMATIC DIAGRAMS

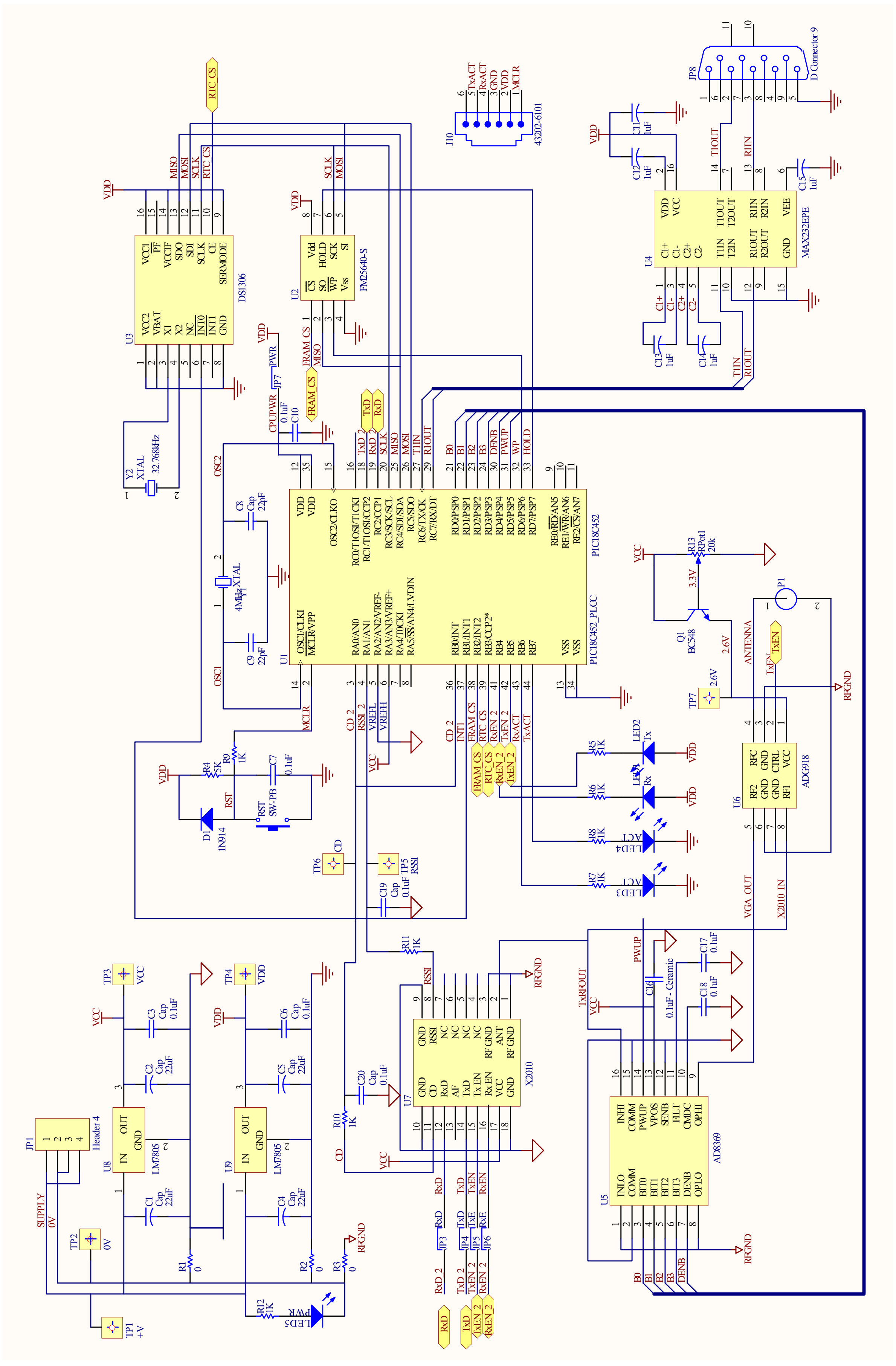


Figure B.1 - Node Rev 3 Schematic Diagram

Appendix C

RAW DATA

Table C.1 - X2010 Transceiver Power Consumption vs Duty Cycle

Duty Cycle	T_{off}	Current	Power
1%	990.0 ms	0.08 mA	0.40 mW
2%	490.0 ms	0.14 mA	0.70 mW
3%	323.3 ms	0.22 mA	1.10 mW
4%	240.0 ms	0.27 mA	1.35 mW
5%	190.0 ms	0.34 mA	1.70 mW
6%	156.7 ms	0.40 mA	2.00 mW
7%	132.9 ms	0.47 mA	2.35 mW
8%	115.0 ms	0.53 mA	2.65 mW
9%	101.1 ms	0.59 mA	2.95 mW
10%	90.0 ms	0.66 mA	3.30 mW
15%	56.7 ms	0.94 mA	4.70 mW
20%	40.0 ms	1.24 mA	6.20 mW
25%	30.0 ms	1.51 mA	7.55 mW
30%	23.3 ms	1.80 mA	9.00 mW
35%	18.6 ms	2.10 mA	10.50 mW
40%	15.0 ms	2.39 mA	11.95 mW
45%	12.2 ms	2.69 mA	13.45 mW
50%	10.0 ms	3.01 mA	15.05 mW
100%	-	5.96 mA	29.80 mW

Appendix D

NS-2 SIMULATION SOFTWARE

Table D.1 - DAA.H

```
/*
 * File: Code for a new 'Dynamic Address Allocation' Agent Class for the ns
 *       network simulator
 * Author: Li-Wen Yip (LiWen.Yip@jcu.edu.au), September 2005
 *
 */


#ifndef ns_daa_h
#define ns_daa_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "address.h"
#include "ip.h"


/*
 * Packet Types:
 * AREQ = Address Request
 * AREP = Address Reply (Offer)
 * NREP = Negative Reply
 * AACK = Address Accept
 * PACK = Proxy Address Accept
 * ASRCH = Initiate Address Search
 */

enum {AREQ, AREP, NREP, AACK, PACK, ASRCH};


/*
 * Agent States:
 * UNINIT = Uninitialised
 * INIT = Initialising
 * IDLE = Idle
 * ALLOC = Allocating addresses
 * PROXY = Acting as an allocation proxy
 */

enum {UNINIT, INIT, IDLE, ALLOC, PROXY};


/*
 * The data structure for the Dynamic Address Allocation packet header
 */
struct hdr_daa {
    short type;          // The message type
    int seq;             // The sequence number
    int uid;             // The unique ID of the requesting node
    int alloc_addr[2];  // The address range being allocated
    // Header access methods
    static int offset_; // required by PacketHeaderManager
    inline static int& offset() { return offset_; }
    inline static hdr_daa* access(const Packet* p) {
        return (hdr_daa*) p->access(offset_);
    }
};

/*
 * Define the Dynamic Address Allocation agent as a subclass of "Agent"
 */
class DaaAgent : public Agent {
 public:
    // Default Constructor
    DaaAgent();
    // Execute a command
    int command(int argc, const char*const* argv);
    // Process a packet
    void recv(Packet*, Handler*);


    // Agent Variables
    int state_;                     // Agent State
    int uid_;                       // Unique identifier;
    int my_addr[2];                 // My Address Space
    // Variables for Initialisation
    int areq_retry_ =;              // AREQ retry counter
    static const int AREQ_LIMIT_;   // AREQ retry limit
    static const int AREQ_TIMEOUT_; // AREQ retry timeout
    int arep_counter_;              // AREP retry/received counter
    int nrep_counter_;              // NREP received counter
    Packet* best_offer_;            // Best offer
    int best_offer_size_;           // Size of the best offer
    // Variables for Allocation
    int arep_retry_                 // AREP retry counter
    static const int AREP_LIMIT_;   // AREP retry limit
    static const int AREQ_TIMEOUT_; // AREP retry timeout;
    int alloc_addr[2];              // Allocated Address Space
    Packet* areq_src_;              // the AREQ we are replying to
    // Controller functions
    void init();
    void alloc();
    void recv_areq(Packet*);
    void recv_arep(Packet*);
    void recv_nrep(Packet*);
    void recv_aack(Packet*);

    // Packet creation functions
    static Packet* create_broadcast(int /*type*/, int /*seq*/);
    static Packet* create_reply(int /*type*/, Packet* /*src*/);

};


#endif
```

Table D.1 - DAA.CC

```
/*
 * File: Code for a new 'Dynamic Address Allocation' Agent Class for the ns
 *       network simulator
 * Author: Li-Wen Yip (LiWen.Yip@jcu.edu.au), September 2005
 *
 */


#include "daa.h"


/*
 * The following two static classes link the C++ classes with corresponding Tcl classes.
 */



int hdr_daa::offset_;
static class DaaHeaderClass : public PacketHeaderClass {
public:
    DaaHeaderClass() : PacketHeaderClass("PacketHeader/Daa",
                           sizeof(hdr_daa)) {
        bind_offset(&hdr_daa::offset_);
    }
} class_daahdr;


static class DaaClass : public TclClass {
public:
    DaaClass() : TclClass("Agent/Daa") {}
    TclObject* create(int, const char*const*) {
        return (new DaaAgent());
    }
} class_daa;




/*
 * The constructor for the class 'DaaAgent'.
 * It binds the variables which have to be accessed both in Tcl and C++.
 */
DaaAgent::DaaAgent() : Agent(PT_DAA)
{
  bind("packetSize_", &size_);
}

/*
 * The function 'command()' is called when a
 * Tcl command for the class 'DaaAgent' is executed.
 */
int DaaAgent::command(int argc, const char*const* argv)
{
    if (argc == 2) {
        if (strcmp(argv[1], "init") == 0) {
            // Run the initialisation procedure
            state_ = INIT;
            my_addr[0] = 0;
            my_addr[1] = 0;
            areq_retry_ = 0;
            arep_counter_ = 0;
            nrep_counter_ = 0;
            best_offer_ = NULL;
            init();
            return(TCL_OK);
        }
  }
  // If the command hasn't been processed by DaaAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}


void DaaAgent::recv(Packet* pkt, Handler*)
{
    // Access the IP header for the received packet:
    hdr_ip* hdrip = hdr_ip::access(pkt);
    // Access the DAA header for the received packet:
    hdr_daa* hdr = hdr_daa::access(pkt);

    // Check the packet type and pass it to the appropriate function.
    if      (hdr->type == AREQ)     recv_areq(pkt);
    else if (hdr->type == AREP)     recv_arep(pkt);
    else if (hdr->type == NREP)     recv_nrep(pkt);
    else if (hdr->type == AACK)     recv_aack(pkt);
//  else if (hdr->type == PACK)     recv_pack(pkt);
//  else if (hdr->type == ASRCH)    recv_asrch(pkt);
    // Discard the packet once it's been processed
    Packet::free(pkt);
}

////////////////////////////////////////////////////////////////////////////////
// CONTROLLER FUNCTIONS
////////////////////////////////////////////////////////////////////////////////
//
// Controls the initialisation procedure
//
void DaaAgent::init()
{
    // If we aren't in the INIT state, then bugger off
    if (state_ != INIT) return;

    // Check if we have received any replies.
    // If we have, then accept the best offer and enter the IDLE state.
    if (arep_counter_ > 0)
    {
        // Create a reply to the best offer and send it
        Packet* reply = create_reply(AACK, best_offer_);
        send(reply, 0);

        // Take posession of the offered addresses
        hdr_daa* hdr = hdr_daa::access(best_offer_);
        my_addr[0] = hdr->alloc_addr[0];
        my_addr[1] = hdr->alloc_addr[1];

        // Dispose of the packet
        Packet::free(best_offer_);

        // Enter the idle state
        state_ = IDLE;
        return;
    }

    // We haven't received any replies, so check how many times we have broadcast AREQ.
    // If it is less than the limit, (re)broadcast the AREP message, schedule a retry.
    else if (areq_retry_ <= AREQ_LIMIT_)
    {
        // Send an AREQ broadcast packet, and include a sequence number
        Packet* request = create_broadcast(AREQ, areq_retry_++);
        send(request, 0);

        // Schedule a timeout
        // ### TO DO ###
        return;
    }

    // We have exceeded the retry limit, so assume we are not in range of an
    // existing network and take posession of the entire address space.
    else
    {
        // Take posession of the entire address space
        my_addr[0] = 1;
        my_addr[1] = 254;

        // Enter the idle state
        state_ = IDLE;
        return;
    }


}

//
// Controls the allocation procedure
//
void DaaAgent::alloc()
{
    // If we aren't in the ALLOC state, bugger off.
    if (state_ != ALLOC) return;

    // If we are still in the ALLOC state then we haven't had a reply.
    // If we haven't reached the retry limit, (re)transmit an AREP message.
    else if (arep_retry_ <= AREP_LIMIT_)
    {
        // Send an AREP reply to the AREQ source packet
        Packet* reply = create_reply(AREP, areq_src_);
        send(reply, 0);

        // Increase the retry counter
        arep_retry_++;

        // Schedule a timeout
        // ## TO DO ##
    }

    // If we have reached the retry limit, give up and go back to the IDLE state.
    else
    {
        Packet::free(areq_src_);
        state_ = IDLE;
    }

}

//////////////////////////////////////////////////////////////////////////////
// PACKET RECEIVING FUNCTIONS
//////////////////////////////////////////////////////////////////////////////
void DaaAgent::recv_areq(Packet* pkt)
{
    // If we are not in the IDLE state, discard the packet
    if (state_ != IDLE)
    {
        Packet::free(pkt);
        return;
    }

    // Check if we have any available addresses
    else if (my_addr[1] > my_addr[0])
    {
        // We have available addresses: allocate the upper half.
        alloc_addr[1] = my_addr[1];
        alloc_addr[0] = (my_addr[0] + my_addr[1]) / 2;
        // Start the allocation procedure
        arep_retry_ = 0;        // Reset the retry counter
        areq_src_ = pkt;        // Save the AREQ packet
        alloc();
        return;
    }

    // We don't have available addresses, if this is only the first request then
    // discard the packet.
    else if (hdr_daa::access(pkt)->seq == 0)
    {
        Packet::free(pkt);
    }

    // If this is not the first request, then send a negative reply and go into
    // the proxy state.
    else
    {

        // Send the NREP
        Packet* reply = create_reply(NREP, pkt);
        send(reply, 0);

        // Enter the proxy state
        Packet::free(pkt);

        return;
    }
}

// Process an AREP packet
void DaaAgent::recv_arep(Packet* pkt)
{
    // If we are not in initialisation mode, discard the packet.
    if (state_ != INIT)
    {
        Packet::free(pkt);
    }

    // If the uid doesn't match, discard the packet.
    if (hdr_daa::access(pkt)->uid != uid)
    {
        Packet:free(pkt);
    }

    // If it is better than our previous best offer, discard the
    // previous best offer and save the new one
    else
    {
        arep_counter_++;
        hdr_daa* hdr = hdr_daa::access(pkt);
        int new_offer_size = hdr->alloc_addr[1] - hdr->alloc_addr[0];
        if (new_offer_size > best_offer_size_ || best_offer_ == NULL)
        {
            Packet::free(best_offer_);
            best_offer_ = pkt;
            best_offer_size_ = new_offer_size;
        }
        else
        {
            Packet::free(pkt);
        }
        return;
    }

}
```

```
// Process an NREP packet
void DaaAgent::recv_nrep(Packet* pkt)
{
    // Discard if we are not in initialisation mode.
    if (state_ != INIT)
    {
        Packet::free(pkt);
    }
    else
    {
        nrep_counter_++;
        // save the address so we can track which addresses are in use.
        // ## TO DO ##
        Packet::free(pkt);
    }
}

// Process an AACK packet
void DaaAgent::recv_aack(Packet* pkt)
{
    // Discard if we are not in allocation mode.
    if (state_ != ALLLOC)
    {
        Packet::free(pkt);
    }
    else
    {
        // Check the UID on the packet:
        hdr_daa* hdr = hdr_daa::access(pkt);
        if (hdr->uid ==
    }
}



//////////////////////////////////////////////////////////////////////////////
// PACKET FUNCTIONS
//////////////////////////////////////////////////////////////////////////////
//
// Creates a new broadcast with the specified type and sequence number.
//
Packet* DaaAgent::create_broadcast(int type, int seq)
{
    // Create a new packet
    Packet* pkt = allocpkt();

    // Populate the header with type, sequence number, and UID
    hdr_daa* hdr = hdr_daa::access(pkt);
    hdr->type = type;
    hdr->seq = seq;
    hdr->uid = uid_;

    // Set the destination address to broadcast in the IP header
    hdr_ip* iphdr = hdr_ip::access(pkt);
    iphdr->daddr() = IP_BROADCAST;
    iphdr->dport() = iphdr->sport();

    // Give it back
    return pkt;
}

//
// Creates a reply to the specified packet of the specified type
//
Packet* DaaAgent:: create_reply(int type, Packet* src)
{
    // Create a new packet
    Packet* pkt = allocpkt();

    // Copy the UID from the old packet into the new packet, and set the type
    hdr_daa* dest_hdr = hdr_daa::access(pkt);
    hdr_daa* src_hdr = hdr_daa::access(src);
    dest_hdr->uid = src_hdr->uid;
    dest_hdr->type = type;

    // Copy saddr from the old packet into daddr of the new packet
    hdr_ip* dest_iphdr = hdr_ip::access(pkt);
    hdr_ip* src_iphdr = hdr_ip::access(src);
    dest_iphdr->daddr() = src_iphdr->saddr();
    dest_iphdr->dport() = src_iphdr->sport();

    // Give it back
    return pkt;
}
```

Appendix E

MICROCONTROLLER SOFTWARE LISTING

E.1 HEADER FILES

```
;*************************************************************************
; Master Header File
; Version 1.00
; 16/09/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;*************************************************************************
 LIST P=18C452, F=INHX32        ; directive to define processor and file format
 #include <P18C452.INC>         ; processor specific variable definitions
 #include "macrolib.inc"        ; common macros
 #include "pinconnections.inc"  ; Pin connections
 #include "swstack.inc"         ; Software Stack
```

Table E.1 - MACROLIB.INC

```
;**************************************************************************
; Macro Library
; Version 1.00
; 16/09/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;**************************************************************************
;**********************************************************************************
; Macro:        SERVICE_IRP
;
; Description:  Service an interrupt.
;               MUST BE CALLED FROM WITHIN A SUBROUTINE.
;
; If the flag was set, it clears it.
; If the flag wasn't set, or the interrupt wasn't enabled,
; it returns from the subroutine.
; Regs Used:
;**********************************************************************************
SERVICE_IRP macro int_flag_reg, int_flag, int_en_reg, int_en_flag

; Check the interrupt enable flag.
    btfss   int_en_reg, int_en_flag     ; Was the interrupt enabled?
    return                              ; NO - exit the routine.
; Check the interrupt flag.
    btfss   int_flag_reg, int_flag      ; Did this device's interrupt event occur?
    return                              ; NO - return.
    bcf     int_flag_reg, int_flag      ; YES - clear the interrupt flag.

 endm

;**********************************************************************************
; Macro:        ADDLF16
;
; Description:  Add a 16 bit literal to a 16 bit field
; Regs Used:    WREG
;**********************************************************************************
ADDLF16 macro   k, d
    movlw   low k       ; load up the low byte of the literal
    addwf   d           ; add it to the low byte of the destination
    movlw   high k      ; load up the high byte of the literal
    addwfc  d + 1       ; add it and the carry bit to the high byte of the destination
 endm

;**********************************************************************************
; Macro:        SUBLF16
;
; Description:  Subtract a 16 bit literal from a 16 bit field
; Regs Used:    WREG
;**********************************************************************************
SUBLF16 macro   k, d
    bsf     STATUS, C   ; Make sure the low byte subtraction doesnt use the borrow  bit
    movlw   low k       ; load up the low byte of the literal
    subwf   d, f        ; subtract it from the low byte of the destination
    movlw   high k      ; load up the high byte of the literal
    subwfb  d + 1, f    ; subtract it and the borrow bit from the high byte of the dest
 endm
```

Table E.2 - PINCONNECTIONS.INC

```
;**************************************************************************
```

```
; Pin Connections Header File
;
; Version 1.00
; 14/08/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;*************************************************************************
; To get to the TRIS Register, add 0x12.
 #define    TRIS        0x12
; To get to the LAT Register, add 0x09
 #define    LAT         0x09


; -------------------------------------------------------
; Transceiver Connections
; -------------------------------------------------------
; Inputs
 #define    RSSI        PORTA, AN1      ; Pin 3
 #define    NCD         PORTB, RB0      ; Pin 36
 #define    NCD2        PORTA, RA0      ; Pin 4
 #define    RXD         PORTC, CCP1     ; Pin 19
; Outputs
 #define    TXD         PORTC, CCP2     ; Pin 18
 #define    NTXEN       PORTB, RB5      ; Pin 41
 #define    NRXEN       PORTB, RB4      ; Pin 42
 #define    TXLED       PORTB, RB7      ; Pin 44
 #define    RXLED       PORTB, RB6      ; Pin 43
CONFIG_TRANSCIEVER_PINS macro
    ; Inputs
    bsf     TRIS + RSSI
    bsf     TRIS + NCD
    bsf     TRIS + NCD2
    bsf     TRIS + RXD
    bcf     TRIS + TXD
    ; Set Outputs' Initial State
    bsf     LAT + NTXEN     ; High
    bsf     LAT + NRXEN     ; High
    bcf     LAT + TXLED     ; Low
    bcf     LAT + RXLED     ; Low
    ; Outputs
    bcf     TRIS + NTXEN
    bcf     TRIS + NRXEN
    bcf     TRIS + TXLED
    bcf     TRIS + RXLED
 endm


; -------------------------------------------------------
; VGA Connections
; -------------------------------------------------------
; Inputs
 #define    B0          PORTD, RD0      ; Pin 21
 #define    B1          PORTD, RD1      ; Pin 22
 #define    B2          PORTD, RD2      ; Pin 23
 #define    B3          PORTD, RD3      ; Pin 24
 #define    PWUP        PORTD, RD4      ; Pin 30
 #define    DENB        PORTD, RD5      ; Pin 31
CONFIG_VGA_PINS macro
    ; Inputs
    bsf     TRIS + B0
    bsf     TRIS + B1
    bsf     TRIS + B2
    bsf     TRIS + B3
    bsf     TRIS + PWUP
    bsf     TRIS + DENB
 endm

; -------------------------------------------------------
; SPI Connections
; -------------------------------------------------------
CONFIG_SPI_PINS macro
    ; Inputs
    bsf     TRISC, SDI
    ; Outputs
    bcf     TRISC, SDO
    bcf     TRISC, SCK
 endm


; -------------------------------------------------------
; FRAM Connections
; -------------------------------------------------------
; Outputs
 #define    FRAM_NCS    PORTB, RB2      ; Pin 18
 #define    NWP         PORTD, RD6      ; Pin 32
 #define    NHOLD       PORTD, RD7      ; Pin 33
CONFIG_FRAM_PINS macro
    ; Set Outputs' Initial State
    bsf     LAT + FRAM_NCS  ; High
    bcf     LAT + NWP       ; Low
    bsf     LAT + NHOLD     ; High
    ; Outputs
    bcf     TRIS + FRAM_NCS
    bcf     TRIS + NWP
    bcf     TRIS + NHOLD
 endm

; -------------------------------------------------------
; RTC Connections
; -------------------------------------------------------
; Outputs
 #define    RTC_CS      PORTB, RB3      ; Pin 16
CONFIG_RTC_PINS macro
    ; Inputs
    bsf     TRISB, INT1     ; ALARM IRQ
    ; Set Outputs' Initial State
    bcf     LAT + RTC_CS    ; Low
    ; Outputs
    bcf     TRIS + RTC_CS
 endm
```

Table E.3 - SWSTACK.INC

```
;**************************************************************************
; Software Stack
;
; Version 0.10
; 11/08/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;**************************************************************************
;
; Description:
; Dependencies:
; Resources Used:
; Things you must do to use this module:
; What you need to understand to work with this code:
; Notes:
;****************************************************************
; MACRO:    SWPUSH
;
; Description:  Push a file to the software stack.
; Arguments:    file - The file to be pushed to the stack.
; Regs Used:    FSR2 (Exclusive)
;****************************************************************
SWPUSH macro file
;   if file == WREG
;       movwf   PREINC2
;   else
        movff   file, PREINC2
;   endif
    endm

;****************************************************************
; MACRO:    SWPOP
;
; Description:  Pop a file from the software stack.
; Arguments:    f - The file to be popped to.
; Regs Used:    FSR2 (Exclusive)
;****************************************************************
SWPOP macro file
;   if file == WREG
;       movf    POSTDEC2, W
;   else
        movff   POSTDEC2, file
;   endif
    endm

;****************************************************************
; MACRO:    STACKINIT
;
; Description:  Initialise the software stack
; Arguments:    The address of the stack
; Regs Used:    FSR2 (Exclusive)
;****************************************************************
STACKINIT macro addr
    lfsr 2, addr
    endm
```

Table E.4 - BUFFERS.INC

```
;***************************************************************************
; Software Buffers Header File
;
; Version 1.00
; 14/08/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;***************************************************************************
  extern        buflen

;****************************************************************
; MACRO:        BUF_SEL
;
; Description:  Select a buffer
; Arguments:    addr - the address of the buffer.
;               size - the size of the buffer.
; Postcond'ns:  - Address of buffer has been loaded to FSR0.
;               - Address of cursor and END has been loaded to FSR1.
;               - Length of buffer has been loaded to buflen.
; Regs Ch'd: WREG
;****************************************************************
; CURSOR(1) : END(1) : DATA(VAR) ;
; ^ FSR1             ^ FSR0
BUF_SEL macro addr, size
    lfsr    0, addr + 2         ; Load address of buffer data area to FSR0.
    lfsr    1, addr             ; Load address of buffer variables to FSR1.
    movlw   size - 2            ; Load the size of the buffer data area.
    movwf   buflen              ; Store it in buflen.
 endm

;****************************************************************
; MACRO:        BUF_GETCURSOR
;
; Description:  Gets the cursor value of the currently selected buffer.
; Arguments:    None.
; Precond'ns:   A buffer has been selected with BUF_SEL
; Postcond'ns:  cursor -> WREG
; Regs Used:    WREG
;****************************************************************
; INDF1 = cursor
BUF_GETCURSOR macro
    movf    INDF1, W
 endm

;****************************************************************
; MACRO:        BUF_SETCURSOR
;
; Description:  Sets the cursor value of the currently selected buffer.
; Arguments:    Cursor value in WREG.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  WREG -> cursor
; Regs Used:    None.
;****************************************************************
; INDF1 = cursor
BUF_SETCURSOR macro
    movwf   INDF1
 endm

;****************************************************************
; MACRO:        BUF_CLEAR
;
```

```
; Description:  Clears the currently selected buffer.
; Arguments:    None.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  0 -> cursor, 0 -> end.
; Regs Used:    None.
;****************************************************************
; POSTINC1 = cursor, POSTDEC1 = end
BUF_CLEAR macro
    clrf    POSTINC1            ; 0 -> Cursor
    clrf    POSTDEC1            ; 0 -> End
 endm


;****************************************************************
; MACRO:        BUF_MARKEND
;
; Description:  Mark the current cursor position as the end of the
;                   currently selected buffer.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  - Cursor -> End
; Regs Ch'd:    WREG
;****************************************************************
; POSTINC1 = cursor, POSTDEC1 = end
BUF_MARKEND macro
    movf    POSTINC1, W         ; Cursor -> WREG
    movwf   POSTDEC1            ; WREG -> End
 endm

;****************************************************************
; MACRO:        BUF_GETEND
;
; Description:  Get the end index of the buffer (i.e. the length of the
;                   data contained in the buffer).
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  End -> WREG
; Regs Ch'd:    WREG
;****************************************************************
; POSTINC1 = cursor, POSTDEC1 = end
BUF_GETEND macro
    movf    POSTINC1, F         ; Increment FSR1
    movf    POSTDEC1, W         ; End -> WREG
 endm


;****************************************************************
; MACRO:        BUF_PUT
;
; Description:  Write a byte to the buffer at the current cursor location
;               and increment the cursor by one.
; Arguments:    Data byte in WREG.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  WREG -> Buffer[Cursor++]
; Regs Ch'd:    WREG
;****************************************************************
; INDF1 = cursor, PLUSW0 = Buffer[Cursor]
BUF_PUT macro
    movwf   PREINC2             ; Push our data byte onto the stack.
    movf    INDF1, W            ; Load the cursor to WREG.
    movff   POSTDEC2, PLUSW0    ; Pop our data byte on the buffer.
    incf    INDF1               ; Increment the cursor.
 endm

;****************************************************************
; MACRO:    BUF_GET
;
; Description:  Get a byte from the buffer at the current cursor
;                   location and increment the cursor by one.
; Arguments:    Data byte in WREG.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  Buffer[Cursor++] -> WREG
; Regs Ch'd:    WREG
;****************************************************************
; INDF1 = cursor, PLUSW0 = Buffer[Cursor]
BUF_GET macro
    ; Read a byte from the buffer.
    movf    INDF1, W            ; Load the cursor to WREG.
    movf    PLUSW0, W           ; Read a byte from WREG.
    ; Increment the cursor.
    incf    INDF1, F
 endm

;****************************************************************
; MACRO:    BUF_FREE
;
; Description:  Calculate the amount of free space in the buffer.
; Arguments:    None.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:  buflen - end -> WREG
; Regs Ch'd:    WREG
;****************************************************************
BUF_FREE macro
    BUF_GETEND                  ; Load the end position.
    subwf       buflen, W       ; buflen 0 end -> WREG
 endm

;****************************************************************
; MACRO:    BUF_SNEOF (Skip if cursor is not at end of data)
;
; Description:  Skip if cursor < end
; Arguments:    None.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:
; Regs Ch'd:    WREG.
;****************************************************************
; POSTINC1 = cursor, POSTDEC1 = end
BUF_SNEOF macro
    movf    POSTINC1, W         ; Load the cursor to WREG.
    cpfsgt  POSTDEC1            ; Skip if end > cursor.
 endm

;****************************************************************
; MACRO:    BUF_SNFULL (Skip if buffer is not full)
;
; Description:  Skip if cursor < buflen
; Arguments:    None.
; Precond'ns:   A buffer has been selected with BUF_SEL.
; Postcond'ns:
; Regs Ch'd:    WREG.
;****************************************************************
; INDF1 = cursor
BUF_SNFULL macro
    movf    INDF1, W            ; Load the cursor to WREG.
    cpfsgt  buflen              ; Skip if buflen > cursor.
 Endm
```

E.2 MODULES

Table E.5 - MAIN.ASM

```
;******************************************************************************
;   This file is a basic template for creating relocatable assembly code for  *
;   a PIC18C452. Copy this file into your project directory and modify or     *
;   add to it as needed. Create a project with MPLINK as the language tool    *
;   for the hex file. Add this file and the 18C452.LKR file to the project.   *
;                                                                             *
;   The PIC18CXXX architecture allows two interrupt configurations. This      *
;   template code is written for priority interrupt levels and the IPEN bit   *
;   in the RCON register must be set to enable priority levels. If IPEN is    *
;   left in its default zero state, only the interrupt vector at 0x008 will   *
;   be used and the WREG_TEMP, BSR_TEMP and STATUS_TEMP variables will not    *
;   be needed.                                                                *
;                                                                             *
;   Refer to the MPASM User's Guide for additional information on the         *
;   features of the assembler and linker.                                     *
;                                                                             *
;   Refer to the PIC18CXX2 Data Sheet for additional information on the       *
;   architecture and instruction set.                                         *
;                                                                             *
;******************************************************************************
;                                                                             *
;    Filename:      Main.asm                                                  *
;    Date:          25/09/2005                                                *
;    File Version:  1.00                                                      *
;                                                                             *
;    Author:        Li-Wen Yip                                                *
;    Company:       James Cook University                                     *
;                                                                             *
;******************************************************************************
;                                                                             *
;    Files required:         P18C452.INC                                      *
;                            18C452.LKR                                       *
;                                                                             *
;******************************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"


;********************************************************************************
; EXTERNAL LABELS
 #include "PHY.inc"
 #include "MAC.inc"

 #include "RTC.inc"
 #include "buffers.inc"     ; Labels for buffers.asm
; From <SoftwareTimers.asm>
 extern SWTIMERS_INIT, SWTIMERS_ISR

; From <SPI.asm>
 extern SPI_INIT

; From <RouteCache.asm>
 extern FRAM_INIT

;******************************************************************************
;Configuration bits
; The __CONFIG directive defines configuration data within the .ASM file.
; The labels following the directive are defined in the P18C452.INC file.
; The PIC18CXX2 Data Sheet explains the functions of the configuration bits.
; Change the following lines to suit your application.
;   __CONFIG    _CONFIG0, _CP_OFF_0
;   __CONFIG    _CONFIG1, _OSCS_OFF_1 & _RCIO_OSC_1
;   __CONFIG    _CONFIG2, _BOR_ON_2 & _BORV_25_2 & _PWRT_OFF_2
;   __CONFIG    _CONFIG3, _WDT_ON_3 & _WDTPS_128_3
;   __CONFIG    _CONFIG5, _CCP2MX_ON_5
;   __CONFIG    _CONFIG6, _STVR_ON_6
;******************************************************************************
;Variable definitions
; These variables are only needed if low priority interrupts are used.
; More variables may be needed to store other special function registers used
; in the interrupt routines.
MAIN_UDATA  UDATA

WREG_TEMP   RES 1   ;variable in RAM for context saving
STATUS_TEMP RES 1   ;variable in RAM for context saving
BSR_TEMP    RES 1   ;variable in RAM for context saving

 UDATA 0x500
swstack     RES 64

;******************************************************************************
;Reset vector
; This code will start executing when a reset occurs.
RESET_VECTOR    CODE 0x0000
    goto    Main        ;go to start of main code
;******************************************************************************
;High priority interrupt vector
; This code will start executing when a high priority interrupt occurs or
; when any interrupt occurs if interrupt priorities are not enabled.
HI_INT_VECTOR   CODE 0x0008
    bra     HighInt     ;go to high priority interrupt routine
;******************************************************************************
;Low priority interrupt vector
; This code will start executing when a low priority interrupt occurs.
; This code can be removed if low priority interrupts are not used.
LOW_INT_VECTOR  CODE 0x0018
    bra     LowInt      ;go to low priority interrupt routine


 CODE

;******************************************************************************
;High priority interrupt routine
; The high priority interrupt code is placed here.
HighInt:

; Check if CCP1 (Receive Clock) caused an interrupt.
_CCP1_ISR:
    btfss   PIR1, CCP1IF        ; Is the interrupt flag set?
    bra     _CCP2_ISR           ; NO - skip to checking CCP2.
    bcf     PIR1, CCP1IF        ; YES - clear it.
    btfss   PIE1, CCP1IE        ; Was the interrupt enabled?
    bra     _CCP2_ISR           ; NO - skip to checking CCP2.
    ; Let's do some decoding. Yeah Baby!
    call    MANCHESTER_DECODER  ; Call the manchester decoding routine. <PHY.asm>
    call    FRAME_DECODER       ; Call the Frame decoding routine. <MAC.asm>
    retfie  FAST                ; Done.
; Check if CCP2 (Transmit Clock / Receive Clock Watchdog) caused an interrupt.
_CCP2_ISR:
    btfss   PIR2, CCP2IF        ; Is the interrupt flag set?
    retfie  FAST                ; NO - return.
    bcf     PIR2, CCP2IF        ; YES - clear it.
    btfss   PIE2, CCP2IE        ; Was the interrupt enabled?
    retfie  FAST                ; NO - return.
    ; Check if we should run the encoder routine or the watchdog routine.
    btfss   NTXEN               ; Are we in transmit mode?
    bra     _RUN_ENCODER        ; YES
    call    CLOCK_WATCHDOG      ; NO - run the watchdog routine. <PHY.asm>
    call    RESET_DECODER       ; Notify the MAC layer clock has been lost. <MAC.asm>
    retfie  FAST                ; Done.
_RUN_ENCODER:
    ; Let's do some encoding.
    call    MANCHESTER_ENCODER  ; Call the manchester encoding routine. <PHY.asm>
    call    FRAME_ENCODER       ; Call the frame encoding routine. <MAC.asm>
    retfie  FAST                ; Done.
;******************************************************************************
;Low priority interrupt routine
; The low priority interrupt code is placed here.
; This code can be removed if low priority interrupts are not used.
LowInt:
    movff   STATUS, PREINC2     ; Save STATUS register
    movwf   PREINC2             ; Save WREG register
    movff   BSR, PREINC2        ; Save BSR register
    call    SWTIMERS_ISR        ; <SWTimers.asm>
    call    TMR0_ISR            ; <MAC.asm>
    call    TRAFFIC_CONTROL     ; <MAC.asm>
    call    INT1_ISR            ; <MAC.asm>
    movff   POSTDEC2, BSR       ; Restore BSR register
    movf    POSTDEC2, W         ; Restore WREG register
    movff   POSTDEC2, STATUS    ; Restore STATUS register
    retfie

;******************************************************************************
;Start of main program
; The main program code is placed here.
Main:

    ; Disable interrupts during initialisation
    bcf     INTCON, GIEL        ; disable low priority interrupt.
    bcf     INTCON, GIEH        ; disable high priority interrupt.
    ; Initialisation Routines
    lfsr    2, 0x500            ; Initialise the stack
    call    SWTIMERS_INIT       ; Initialise software timers
    call    PHY_INIT            ; Initialise physical layer
    call    MAC_INIT            ; Initialise mac layer
    call    SPI_INIT            ; Initialise SPI Bus
    call    FRAM_INIT           ; Initialise FRAM chip
    call    RTC_INIT            ; Initialise RTC chip
    ; Reenable Interrupts
    bsf     RCON, IPEN          ; enable interrupt priorities
    bsf     INTCON, GIEL        ; enable low priority interrupt.
    bsf     INTCON, GIEH        ; enable high priority interrupt.

;   call    RX_ENABLE
;   bcf     INTCON3, INT1IE
    extern  MAC_TEST
    bra     MAC_TEST


Loopy:
    clrwdt
    bra Loopy

;******************************************************************************
;End of program
    reset   ; If we ever get here do a reset.
 END
```

Table E.6 - PHY.INC

```
;**************************************************************************
; Physical Layer Header File
; Version 1.00
; 16/09/2005
; Initialisation Routine
 extern     PHY_INIT

; Manchester Encoding
 extern     MANCHESTER_ENCODER

; Manchester Decoding
 extern     MANCHESTER_DECODER, CLOCK_WATCHDOG

; Hardware Control
 extern     RX_DISABLE, TX_DISABLE, RX_ENABLE, TX_ENABLE

; Variables
 extern phy_status, txbyte, rxbyte, bitcount, shiftreg_l, shiftreg_h

 #define txbyte_empty   phy_status, 0 ; Set when a byte has been transmitted.
 #define rxbyte_full    phy_status, 1 ; Set when a byte has arrived.
 #define clock_detect   phy_status, 2 ; Set when there is a valid clock.
```

Table E.7 - PHY.ASM

```
;**************************************************************************
; PHYSICAL LAYER MODULE
;
; Version 1.00
; 16/09/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;**************************************************************************
;
; Description:
;
; Functions:
;
; Dependencies:
;
; Resources Used:
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;**************************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;**************************************************************************
; CONFIGURATION CONSTANTS
; Chip Length: The number of periods of TMR3 that equals one chip.
 #define    CHIPTIME    d'100'      ; 100 us
; Power on Delay: The amount of time to wait between powering up the transmitter
; hardware and starting to transmit data. Defined in periods of TMR3.
 #define    TX_DELAY    d'5000'     ; 5 ms
;**************************************************************************
; CONSTANTS
; CCP Compare Modes:
CCP_IRP         equ b'1010'     ; Just generate an interrupt on match
CCP_RAISE       equ b'1000'     ; Raise the output on match
CCP_CLEAR       equ b'1001'     ; Clear the output on match
CCP_TOGGLE      equ b'0010'     ; Toggle the output on match - doesn't seem to work
; CCP Capture Modes:
CCP_FALLING     equ b'0100'     ; Capture every falling edge
CCP_RISING      equ b'0101'     ; Capture every rising edge
CCP_RISING4     equ b'0110'     ; Capture every 4th rising edge
CCP_RISING16    equ b'0111'     ; Capture every 16th rising edge
 #define rx_get_sample  CCP1CON, 3      ; Have we been waiting to get a sample?
;*******************************************************************************
; GLOBAL VARIABLES
.phy_global_vars udata_acs

; Status register - used to communicate state information to and from the MAC layer.
phy_status      res 1
 #define txbyte_empty   phy_status, 0 ; Set when a byte has been transmitted.
 #define rxbyte_full    phy_status, 1 ; Set when a byte has arrived.
 #define clock_detect   phy_status, 2 ; Set when there is a valid clock.
; Tx/Rx Registers - data is exchanged with the physical layer via these registers.
txbyte          res 1
rxbyte          res 1

; Bit counter - used to synchronise to the start of a byte.
bitcount        res 1

; 16 bit shift register - directly accessed by the MAC layer to detect the SOF.
shiftreg_l      res 1
shiftreg_h      res 1

 global phy_status, txbyte, rxbyte, bitcount, shiftreg_l, shiftreg_h


;*******************************************************************************
; LOCAL VARIABLES
.phy_local_vars udata_acs

phy_flags           res 1
 #define    tx_clock    phy_flags, 0    ; Set when we are encoding a clock chip.

;*******************************************************************************
; IMPORTED SUBROUTINES

;*******************************************************************************
; EXPORTED SUBROUTINES
; Initialisation Routine
 global     PHY_INIT

; Manchester Encoding
 global     MANCHESTER_ENCODER

; Manchester Decoding
 global     MANCHESTER_DECODER, CLOCK_WATCHDOG

; Harware Control
 global     RX_DISABLE, TX_DISABLE, RX_ENABLE, TX_ENABLE


;*******************************************************************************
; START OF CODE
 CODE
```

```
;***************************************************************
; SUBROUTINE:   PHY_INIT
;
; Description:  Initialisation for physical layer.
; Precond'ns:   -
; Postcond'ns:
; Regs Used:    WREG, CCP1 Registers, CCP2 Registers, TMR3 Registers
;***************************************************************
PHY_INIT:

    ; Configure Transceiver pins.
    CONFIG_TRANSCIEVER_PINS

    ; Configure TIMER3 (Chiprate Generator)
    movlw   b'11000001'         ; Load config byte for TIMER3.
    ;        '1-------'         ; Enable 16-bit Read/Write.
    ;        '-1--X---'         ; Use TIMER3 for both CCP modules.
    ;        '--00----'         ; 1:1 Prescale.
    ;        '------0-'         ; Use internal clock (Fosc/4)
    ;        '-------1'         ; Turn Timer3 on
    movwf   T3CON

    ; Set up interrupts.
    bcf     PIE1, CCP1IE        ; Disable the CCP1 interrupt.
    bcf     PIE2, CCP2IE        ; Disable the CCP2 interrupt.
    bcf     PIE2, TMR3IE        ; Disable the TMR3 interrupt.
    bsf     IPR1, CCP1IP        ; CCP1 (Receive) is high priority.
    bsf     IPR2, CCP2IP        ; CCP2 (Transmit) is high priority.
    return

;***************************************************************
; SUBROUTINE:   RX_ENABLE
;
; Description:  Enable the receiver.
; Precond'ns:
; Postcond'ns:  - Transmitter is disabled.
;               - /RXEN is activated (set low).
;               - CCP1 is set to interrupt in 5ms.
;               - CCP1 Interrupt is enabled (PIE1<CCP1IE> is set).
;               - TMR1 Interrupt is enabled (PIE1<TMR1IE> is set).
; Regs Used:
;***************************************************************
RX_ENABLE:
    call    TX_DISABLE          ; Disable the Transmitter.
    bcf     NRXEN               ; Activate /RXEN.
    ; Reset the state of the manchester decoder.
    clrf    bitcount            ; Reset the bit counter.
    bcf     clock_detect

    ; Set CCP1 to match in about 5ms.
    ; This gives the transmitter enough time to power up.
    bcf     PIE1, CCP1IE        ; Disable the interrupt to stop it going off.
    movlw   low TX_DELAY        ; Load the low byte of 5ms.
    addwf   TMR3L, W            ; Add low byte of TMR3 value.
    movwf   CCPR1L              ; Store it in low byte of CCPR1
    movlw   high TX_DELAY       ; Load the high byte of 5ms.
    addwfc  TMR3H, W            ; Add the high byte of TMR3 value with carry.
    movwf   CCPR1H              ; Store it in high byte of CCPR1
    ; Enable CCP1 Interrupt.
    movlw   CCP_IRP             ; Set CCP1 to generate software interrupt on match.
    movwf   CCP1CON
    bcf     PIR1, CCP1IF        ; Clear the flag so we don't get false interrupt.
    bsf     PIE1, CCP1IE        ; Enable the interrupt.
    ; Enable CCP2 Interrupt.
    movlw   CCP_IRP             ; Set CCP2 to generate software interrupt on match.
    movwf   CCP2CON             ; ...
    bcf     PIR2, CCP2IF        ; Clear the flag so we don't get false interrupt.
    bsf     PIE2, CCP2IE        ; Enable the CCP2 interrupt.
    return

;****************************************************************
; SUBROUTINES:  TX_ENABLE
;
; Description:  Enable the transmitter.
; Precond'ns:
; Postcond'ns:  - Receiver is disabled.
;               - /TXEN is activated (set low).
;               - VGA is powered up.
;               - CCP2 is set to interrupt in 5ms.
;               - CCP2 Interrupt is enabled (PIE2<CCP2IE> is set).
; Regs Used:
;****************************************************************
TX_ENABLE:
    ; Enable the transmitter hardware.
    call    RX_DISABLE          ; Disable the Receiver.
    bsf     PWUP                ; Power up the VGA.
    bcf     NTXEN               ; Activate /TXEN.
    ; Reset the state of the encoder.
    clrf    phy_flags
    clrf    bitcount

    ; Set CCP2 to match in about 5ms.
    ; This gives the transmitter enough time to power up.
    bcf     PIE2, CCP2IE        ; Disable the interrupt to stop it going off.
    movlw   low TX_DELAY        ; Load the low byte of 5ms.
    addwf   TMR3L, W            ; Add low byte of TMR3 value.
    movwf   CCPR2L              ; Store it in low byte of CCPR2.
    movlw   high TX_DELAY       ; Load the high byte of 5ms.
    addwfc  TMR3H, W            ; Add the high byte of TMR3 value with carry.
    movwf   CCPR2H              ; Store it in high byte of CCPR2.
    ; Enable CCP2 Interrupt.
    movlw   CCP_TOGGLE          ; Set CCP2 to toggle TxD on a match
    movwf   CCP2CON             ; ...
    bcf     PIR2, CCP2IF        ; Clear the flag so we don't get false interrupt.
    bsf     PIE2, CCP2IE        ; Enable the CCP2 interrupt.
    return

;****************************************************************
; SUBROUTINES:  RX_DISABLE
;
; Description:  Disable the receiver.
; Precond'ns:
; Postcond'ns:  - TMR1 Interrupt is disabled.
;               - CCP1 Interrupt is disabled.
;               - /TXEN is deactivated (set high).
; Regs Used:
;****************************************************************
RX_DISABLE:
    bcf     PIE1, CCP1IE        ; Disable CCP1 Interrupt.
    bsf     NRXEN               ; Deactivate /RXEN.
    return

;****************************************************************
; SUBROUTINES:  TX_DISABLE
;
; Description:  Disable the transmitter.
; Precond'ns:
; Postcond'ns:  - CCP2 Interrupt is disabled.
;               - VGA is powered down.
;               - /TXEN is deactivated (set high).
; Regs Used:
;****************************************************************
TX_DISABLE:
    bcf     PIE2, CCP2IE        ; Disable CCP2 Interrupt.
    movlw   CCP_IRP             ; ...
    movwf   CCP2CON             ; ...
    bcf     PWUP                ; Power down the VGA.
    bsf     NTXEN               ; Deactivate /TXEN.
    return




;***************************************************************
; SUBROUTINE:   MANCHESTER_ENCODER (ISR, CCP2)
;
; Description:  - Manchester encodes bytes from txbyte.
; Precond'ns:   - CCP2 Interrupt must be serviced.
; Postcond'ns:
; Regs Used:
;***************************************************************
MANCHESTER_ENCODER:

; ----------------------------------------------------------------
; DECISION: Check if we are encoding a clock chip or a data chip.
    btfsc   tx_clock            ; Are we encoding a clock chip?
    bra     _TX_CLOCKCHIP       ; Yes - encode a clock chip.
; ----------------------------------------------------------------
; DECISION: Check if we need to get another data byte.
    tstfsz  bitcount            ; Does bitcount = 0?
    bra     _TX_DATACHIP        ; NO - don't get another byte.
; ----------------------------------------------------------------
; DECISION: Check if we have another data byte to transmit.
    btfsc   txbyte_empty        ; Is there a transmit byte?
    bra     TX_DISABLE          ; NO - disable the transmitter.
; ----------------------------------------------------------------
; PROCESS: Fetch the next data byte.
    movff   txbyte, shiftreg_l  ; Fetch the next byte from the buffer.
    movlw   d'08'               ; Reset the bit counter.
    movwf   bitcount            ; ...
    bsf     txbyte_empty        ; Set a flag to say we are ready for the next byte.
; ----------------------------------------------------------------
; PROCESS: Encode a data chip on the next cycle.
_TX_DATACHIP:
    decf    bitcount            ; Decrement the bit counter.
    rrcf    shiftreg_l          ; Pop a the LSB off the end of the data byte.
    bc      _DATACHIP_HIGH      ; Data bit was high.
;   bnc     _DATACHIP_LOW       ; Data bit was low.
_DATACHIP_LOW:
    btfss   PORTC, CCP2         ; Is TxD also low?
    bra     _TX_TOGGLE_IN_TWO   ; Yes - we can skip the next cycle.
    bsf     tx_clock            ; NO - set the clock flag ...
    bra     _TX_TOGGLE_IN_ONE   ; ... and Toggle TxD in one chip period.
_DATACHIP_HIGH:
    btfsc   PORTC, CCP2         ; Is TxD also high?
    bra     _TX_TOGGLE_IN_TWO   ; Yes - we can skip the next cycle.
    bsf     tx_clock            ; NO - set the clock flag ...
    bra     _TX_TOGGLE_IN_ONE   ; ... and Toggle TxD in one chip period.
; ----------------------------------------------------------------
; PROCESS: Set CCP2 (Chiprate Generator) to toggle TxD in one chip period.
_TX_TOGGLE_IN_ONE:
    movlw   low CHIPTIME        ; Load up the low byte of the chip time.
    addwf   CCPR2L              ; Add it to the low byte of CCPR2
    movlw   high CHIPTIME       ; Load up the high byte of the chip time.
    addwfc  CCPR2H              ; Add it to the high byte of CCPR2 with carry.
    return

; ----------------------------------------------------------------
; PROCESS: Set CCP2 (Chiprate Generator) to toggle TxD in two chip periods.
_TX_TOGGLE_IN_TWO:
    movlw   low CHIPTIME*2      ; Load up the low byte of the chip time * 2.
    addwf   CCPR2L              ; Add it to the low byte of CCPR2
    movlw   high CHIPTIME*2     ; Load up the high byte of the chip time * 2.
    addwfc  CCPR2H              ; Add it to the high byte of CCPR2 with carry.
    return

; ----------------------------------------------------------------
; PROCESS: Encode a clock chip.
_TX_CLOCKCHIP:

    ; Clear the clock flag.
    bcf     tx_clock

    ; Toggle TxD in one chip period.
    movlw   low CHIPTIME        ; Load up the low byte of the chip time.
    addwf   CCPR2L              ; Add it to the low byte of CCPR2
    movlw   high CHIPTIME       ; Load up the high byte of the chip time.
    addwfc  CCPR2H              ; Add it to the high byte of CCPR2 with carry.
    return


;**************************************************************
; SUBROUTINE:   MANCHESTER_DECODER (ISR, CCP1)
;
; Description:  - Decodes manchester encoded bits and shifts them
;                   into shiftreg_h and shiftreg_l.
; Precond'ns:   - CCP1 interrupt must be serviced.
; Postcond'ns:
; Regs Used:
;**************************************************************
MANCHESTER_DECODER:

; -------------------------------------------------------------------------------
; DECISION: Check if CCP1 was set to capture or compare mode.
; Compare mode (CCP1CON<3> is set) - Go to the sampling routine.
; Capture mode (CCP1CON<3> is clear) - Go to the clock detect routine.
    btfsc   rx_get_sample       ; Were we waiting to get a sample?
    bra     _SAMPLE             ; YES - Take a sample.
    ;bra    _CLOCK_DETECT       ; NO - Synchronise to the clock edge.
; -------------------------------------------------------------------------------
; PROCESS: Sync to the manchester clock (the edges in the middle of each data bit)
; Even if the edge we just captured is not the correct clock edge, it will sort itself
; out as soon as the data bit changes.
; a) Set CCP2 (Clock Watchdog) to match in 2.2 chip periods
; b) Set CCP1 (Chiprate Generator) to match in 1.5 chip periods
_CLOCK_DETECT:

    ; a) Set CCP2 (Clock Watchdog) to match in 2.2 chip periods.
    ; If don't detect a clock edge and run this routine ever two chip periods,
    ; CCP2 will generate an interrupt, indicating we have lost the clock.
    movlw   low CHIPTIME*22/10  ; Load the low byte of 2.2 chiptimes.
    addwf   CCPR1L, W           ; Add the low byte of the current time.
    movwf   CCPR2L              ; Store it to CCPR2.
    movlw   high CHIPTIME*22/10 ; Load the high byte of 2.2 chiptimes.
    addwfc  CCPR1H, W           ; Add the high byte of the current time with carry.
    movwf   CCPR2H              ; Store it to CCPR2.
    ; This is done in RX_ENABLE
    movlw   CCP_IRP             ; Set CCP2 to generate a software interrupt on match.
    movwf   CCP2CON             ; ...
    bcf     PIR2, CCP2IF        ; Prevent a false interrupt after changing modes.
    ; b) Set CCP1 (Chiprate Generator) to match in 1.5 chip periods.
    ; CCP1 should generate an interrupt in the middle of the first chip of the
    ; next bit... the perfect place to sample.
    movlw   low CHIPTIME*3/2    ; Load low byte of 1.5 chiptimes.
    addwf   CCPR1L              ; Add it to the low byte of CCPR1.
    movlw   high CHIPTIME*3/2   ; Load the high byte of the chiptime.
    addwfc  CCPR1H              ; Add it to the high byte of CCPR1 with carry.
    movlw   CCP_IRP             ; Set CCP1 to generate a software interrupt on match.
    movwf   CCP1CON             ; ...
    bcf     PIR1, CCP1IF        ; Prevent a false interrupt after changing modes.
    return                      ; Done.
; ------------------------------------------------------------------------------
; PROCESS: Take a sample and place it into the shift register.
; a) Test the value of RxD.
;       High: set STATUS<C>, set CCP1 to capture the next falling edge.
;       Low: clear STATUS<C>, set CCP1 to capture the next rising edge.
; b) Rotate shiftreg_h and shiftreg_l right through the carry bit.
; c) Increment the bit counter.
;
; NB - RRCF shifts the carry bit into the register -
; this is why we are writing to STATUS<C>.
; Make sure any instructions do not change STATUS<C> until the RRCF is done.
_SAMPLE:
    btfss   PORTC, CCP1         ; Is the TxD high or low?
    bra     _SAMPLE_LOW
_SAMPLE_HIGH:
    bsf     STATUS, C           ; Set the carry bit.
    movlw   CCP_FALLING         ; Next edge will be a falling edge.
    bra     _SAMPLE_1           ; Continue.
_SAMPLE_LOW:
    bcf     STATUS, C           ; Clear the carry bit.
    movlw   CCP_RISING          ; Next edge will be a rising edge.
_SAMPLE_1:
    movwf   CCP1CON             ; Configure CCP1.
    bcf     PIR2, CCP1IF        ; Prevent a false interrupt after changing modes.
    rrcf    shiftreg_h          ; Right shift the carry bit into the shiftreg.
    rrcf    shiftreg_l          ; ...
    incf    bitcount            ; Increment the bit counter to say we received a bit.
; ------------------------------------------------------------------------------
; DECISION: Check if we have received 8 bits.
; YES - Reset the bit counter, copy the byte into rxbyte, and set rxbyte_full.
; NO - increment the bit counter and return.
    movlw   d'07'               ;
    cpfsgt  bitcount            ; Is bitcount >= 8?
    return                      ; NO - return.
    ; Process the byte we just received.
    clrf    bitcount            ; Clear the bit counter.
    movff   shiftreg_h, rxbyte  ; Copy the received byte into rxbyte.
    bsf     rxbyte_full         ; Set a flag to say we have received a byte.
    bsf     clock_detect        ; Set the clock detect flag.
    bsf     RXLED
    return                      ; Bugga orf.



;***************************************************************
; Subroutine:   CLOCK_WATCHDOG (ISR, CCP2)
;
; Description:  Called when CCP2 matches in receive mode, indicating that
;               clock synchronisation has been lost in the receiver.
; Precond'ns:   CCP2 Interrupt must be serviced.
; Postcond'ns:
; Regs Used:
;***************************************************************
CLOCK_WATCHDOG:
    ; We have lost the clock, so reset the decoder.
    clrf    bitcount            ; Reset the bit counter.
    clrf    shiftreg_l          ; Clear the shift registers
    clrf    shiftreg_h          ; ...
    bcf     clock_detect        ; Reset the clock detect flag.
    bcf     RXLED
    return

 end
```

Table E.8 - MAC.INC

```
;***********************************************************************
; MAC Layer Header File
; Version 1.00
; 16/09/2005
 extern     MAC_INIT, MAC_TEST                               ; Initialisation
 extern     FRAME_ENCODER, FRAME_DECODER, RESET_DECODER     ; High Prioirity ISR's
 extern     CSMA, TMR0_ISR, INT1_ISR, TRAFFIC_CONTROL       ; Low Priority ISR's
 extern     rxbuf, txbuf, mac_status
 #define    packet_received     mac_status, 0  ;Indicates we have received a packet.
 #define    packet_transmitted  mac_status, 1  ;Indicates we are ready to send a packet.
 #define    retry_limit_reached mac_status, 2  ;Indicates we reached the retry limit.
```

Table E.9 - MAC.INC

```
;***********************************************************************
; MEDIA ACCESS CONTROL LAYER MODULE
;
; Version 1.00
; 16/9/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;***********************************************************************
;
; Description:
;
; Functions:
;
; Dependencies:
;
; Resources Used:
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;***********************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;***********************************************************************
; CONFIGURATION CONSTANTS
; Preamble (Synchronization Header)
; These variables define the preamble that will be sent at the start of
; every packet.
; The encoder will sent (PREAMBLE_LEN) bytes of PREAMBLE_BYTE,
; followed by one byte of SOF_BYTE.
; The decoder will detect one byte of PREAMBLE_BYTE immediately
; followed by one byte of SOF_BYTE.
 #define    PREAMBLE_BYTE   0x55    ; 10101010 LSB first
 #define    SOF_BYTE        0x00    ; 00000000 LSB first
; CSMA: number of times to retry before giving up.
 #define    RETRY_LIMIT     d'10'   ; retry 10 times.
; Transmit/Recieve buffer sizes
RXBUFLEN        equ 0x40            ; 64 bytes
TXBUFLEN        equ 0x40            ; 64 bytes
```

```
; MAC Constants - in TMR0 periods. TMR0 is running on a 1:256 prescaler.
; Amount of time to stay awake for.
WAKETIME        equ .39             ; 10 milliseconds
; Amount of time to sleep for during low traffic.
SLEEPTIME_L     equ .3906           ; 1 second
;SLEEPTIME_L        equ .10838      ; 2.7746 seconds
; Amount of time to sleep for during high traffic.
SLEEPTIME_H     equ .217            ; 55.6 milliseconds
; Preamble length during low traffic.
PREAMBLELEN     equ .125            ; 125 bytes = 200ms @ 5000 baud
; Preamble length during high traffic.
PREAMBLELEN2    equ .32             ; 32 bytes = 50ms @ 5000 baud
; Define this cosnstant to use BAPSA (BASIC ADAPTIVE PREAMBLE SAMPLING ALGORITHM)
#define BAPSA

;**************************************************************************
; CONSTANTS
; Frame types
NORMAL          equ 0x00
CLOCKSYNC       equ 0x01

;*******************************************************************************
; IMPORTED VARIABLES
 ; Physical Layer
 #include   "PHY.inc"

 ; CRC Checking
 #include   "CRC.inc"

 ; Software Timer 4 - See <SWTimers.asm>
 extern     TMRF4, TMRL4, TMRH4

;*******************************************************************************
; GLOBAL VARIABLES
; Transmit/Receive Buffers
.mac_buffers udata
rxbuf           res RXBUFLEN    ; Receive Buffer
txbuf           res TXBUFLEN    ; Transmit Buffer
rxbuf_b         res RXBUFLEN    ; Receive Buffer
txbuf_b         res TXBUFLEN    ; Transmit Buffer
.mac_globals udata_acs
; Status flags.
mac_status          res 1
 #define    packet_received     mac_status, 0  ;Indicates we have received a packet.
 #define    packet_transmitted  mac_status, 1  ;Indicates we are ready to send a packet.
 #define    retry_limit_reached mac_status, 2  ;Indicates we reached the retry limit.
 global rxbuf, txbuf, mac_status

;*******************************************************************************
; LOCAL VARIABLES
.mac_locals udata_acs

; Internal Flags
mac_flags       res 1
 #define mac_csma_active   mac_flags, 0    ; The CSMA algorithm is running.
 #define mac_sampling      mac_flags, 1    ; We are in a sampling period.
 #define mac_schedule_htp  mac_flags, 2    ; We are in a scheduled high traffic period.
 #define mac_dynamic_htp   mac_flags, 3    ; We are in a dynamic high traffic period.

tx_flags        res 1
 #define tx_clock       tx_flags, 0 ; Do we have to do a toggle?
 #define preamble_sent  tx_flags, 1 ; Set when we have sent the start of frame.
 #define framelen_sent  tx_flags, 2 ; Set when we have sent the frame length.
 #define frame_sent     tx_flags, 3 ; Set when we have sent all the data bytes.
rx_flags        res 1
 #define got_sof        rx_flags, 0 ; Set when we have detected the start of frame.
 #define got_framelen   rx_flags, 1 ; Set when we have received frame length.
 #define got_frame      rx_flags, 2 ; Set when we have received an entire frame.
; Frame Encoding
preamble_len_l  res 1       ; Preamble length
preamble_len_h  res 1       ; ...
bytecount_l     res 1       ; Byte Counter
bytecount_h     res 1       ; ...
; CSMA
retry_counter   res 1       ; Count how many times we have tried to transmit.
timeout_l       res 1       ; Retry timeout.
timeout_h       res 1
timeout_mask_l  res 1       ; Retry timeout mask.
timeout_mask_h  res 1


; Preamble sampling
sleeptime_l     res 1       ; Sleep value in use.
sleeptime_h     res 1       ; ...
hitraf_timer    res 1       ; High traffic mode timer.
schedule_timer  res 1       ; Timer for scheduled high traffic periods
dynamic_timer   res 1       ; Timer for dynamic high traffic periods
sleeping_con    res 1       ; Control register for sleep interval.
preamble_con    res 1       ; Control register for preamble length.
scheduled equ .0
dynamic equ .1



; Popcorn buffering
popcorn_flags   res 1
 #define switch_rxbuf   popcorn_flags, 0  ; rxbuf_a and rxbuf_b are switched.
 #define switch_txbuf   popcorn_flags, 1  ; rxbuf_b and rxbuf_a are switched.
;*****************************************************************************
; IMPORTED SUBROUTINES
 #include "Buffers.inc"         ; Software Buffers Header File
 #include "RTC.inc"

;*****************************************************************************
; EXPORTED SUBROUTINES
 global     MAC_INIT, MAC_TEST                          ; Initialisation
 global     FRAME_ENCODER, FRAME_DECODER, RESET_DECODER    ; High Prioirity ISR's
 global     CSMA, TMR0_ISR, INT1_ISR, TRAFFIC_CONTROL      ; Low Priority ISR's
;*****************************************************************************
; START OF CODE
 CODE

MAC_TEST:

    BUF_SEL txbuf, TXBUFLEN
    BUF_CLEAR
    movlw   41
    BUF_PUT
    movlw   42
    BUF_PUT
    movlw   43
    BUF_PUT
    BUF_MARKEND
    call    TX_PACKET

Loopy:
    clrwdt
    bra Loopy



;****************************************************************
; SUBROUTINE:   MAC_INIT
```

```
;
; Description:  Initialisation for mac layer.
; Precond'ns:   -
; Postcond'ns:
; Regs Used:    WREG, CCP1 Registers, CCP2 Registers, TMR3 Registers
;*************************************************************
MAC_INIT:

    ; Clear all variables.
    clrf    mac_flags
    clrf    retry_counter
    clrf    schedule_timer
    clrf    dynamic_timer
    clrf    sleeping_con
    clrf    preamble_con

    ; Set Sleep interval and preamble length for low traffic
    call    SET_LTP_SLEEP
    call    SET_LTP_PREAMBLE

    ; Configure TIMER0 (Sampling Timer)
    bcf     INTCON, TMR0IE      ; Disable the TMR0 interrupt.
    bcf     INTCON2, TMR0IP     ; TMR0 (Sampling Timer) is low priority.
    movlw   b'10000111'         ; Load config byte for TIMER0.
    ;        '1-------'         ; Enable Timer0.
    ;        '-0------'         ; Configure as 16-bit counter.
    ;        '--0-----'         ; Use internal clock (Fosc/4).
    ;        '----0---'         ; Use Prescaler.
    ;        '-----111'         ; 1:256 Prescale.
    movwf   T0CON

    ; Configure TIMER1 (High Traffic Mode Timer)
    bcf     PIE1, TMR1IE        ; Disable the TMR1 interrupt.
    bcf     IPR1, TMR1IP        ; TMR0 (High Traffic Timer) is low priority.
    movlw   b'10110000'         ; Load config byte for TIMER1.
    ;        '1-------'         ; Enable 16-bit read/writes.
    ;        '--11----'         ; 1:8 Prescale.
    ;        '----0---'         ; Disable Timer1 RC Oscillator.
    ;        '------0-'         ; Use internal clock (Fosc/4).
    ;        '-------0'         ; Disable TIMER1
    movwf   T1CON
    bsf     PIE1, TMR1IE        ; Enable the TMR1 interrupt.

    ; Make sure the TIMER0 Interrupt runs straight away to initialise TMR0
    bsf     INTCON, TMR0IF      ; SET!!! the TIMER0 Interrupt flag.
    bsf     INTCON, TMR0IE      ; Enable the TIMER0 Interrupt.
    return

; ####################################################################################
; ####################################################################################
; ####################################################################################
; ####################################################################################
;
;                        MEDIA ACCESS CONTROL
;
; ####################################################################################
; ####################################################################################
; ####################################################################################
; ####################################################################################
;*************************************************************
; SUBROUTINE:   TX_PACKET
;
; Description:  Transmit a packet.
; Precond'ns:   An outgoing packet has been placed in Buffer 1.
; Postcond'ns:  CRC is appended to buffer and packet transmission is initiated.
; Regs Used:
;*************************************************************
TX_PACKET:
    ; Calculate the CRC on the packet.
    BUF_SEL txbuf, TXBUFLEN     ; Select the transmit buffer
    call    CALC_CRC            ; Calculate the CRC on the buffer.
    movf    crclow, W           ; Load the crc low byte...
    BUF_PUT                     ; And append it to the buffer. <BUFPUT in buffers.asm>
    movf    crchigh, W          ; Load the crc high byte...
    BUF_PUT                     ; And append it to the buffer. <BUFPUT in buffers.asm>
    BUF_MARKEND

_TX_PACKET_TEST:
    ; Reset the retry count and timeout mask and run the CSMA routine.
    movlw   RETRY_LIMIT         ; Set the retry counter.
    movwf   retry_counter       ; ...
    movlw   b'00111111'         ; Set the retry timeout mask.
    movwf   timeout_mask_l      ; ...
    clrf    timeout_mask_h      ; ...
    bsf     mac_csma_active     ; Set the flag that lets the CSMA subroutine run.
    btfss   mac_sampling        ; Are we currently sampling?
    call    _START_SAMPLING     ; NO - start sampling now.
    return


;**************************************************************
; SUBROUTINE:   TMR0_ISR (ISR)
;
; Description:
;
; Precond'ns:
; Postcond'ns:
; Regs Used:
;**************************************************************
TMR0_ISR:

    ; Service the TMR0 Interrupt.
    SERVICE_IRP INTCON, TMR0IF, INTCON, TMR0IE

    ; Check whether CSMA or Preamble Sampling has control
    btfsc   mac_csma_active     ; Is CSMA active?
    bra     CSMA                ; YES - run the CSMA routine.
    bra     PREAMBLE_SAMPLING   ; NO - run the Preamble Sampling routine.

;**************************************************************
; SUBROUTINE:   PREAMBLE_SAMPLING (ISR)
;
; Description:  Use TIMER0 to control preamble sampling.
;               Should be configured as a low priority interrupt.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;**************************************************************
PREAMBLE_SAMPLING:

    ; Check if we are currently transmitting.
    btfss   NTXEN               ; Are we transmitting?
    bra     _DONTTOUCH          ; Don't touch anything.
    ; Check if we are currently sampling.
    btfss   mac_sampling        ; Are we sampling?
    bra     _START_SAMPLING     ; NO - start sampling.
; -------------------------------------------------------------
; Perform a clear channel assesment.
_PREAMBLE_SAMPLING_CCA:
    bcf     mac_sampling        ; Clear the sampling flag.
    btfss   clock_detect        ; Do we have a valid signal?
    call    RX_DISABLE          ; NO - Stop sampling.
    ; Set TMR0 to overflow in SLEEPTIME.
    ; When TMR0 overflows, we will start sampling again.
```

```
_DONTTOUCH:
    bcf     T0CON, TMR0ON       ; Stop the timer
    nop                         ; Wait for it to stop.
    movff   sleeptime_h, TMR0H
    movff   sleeptime_l, TMR0L
    bsf     T0CON, TMR0ON       ; Start the timer
    return

; ---------------------------------------------------------------
; Start sampling.
_START_SAMPLING:
    bsf     mac_sampling        ; Set the sampling flag.
    btfsc   NRXEN               ; Is the receiver already enabled?
    call    RX_ENABLE           ; NO - Enable the receiver.
    ; Set TMR0 to overflow in WAKETIME.
    ; When TMR0 overflows we will stop sampling.
    bcf     T0CON, TMR0ON       ; Stop the timer
    nop                         ; Wait for it to stop.
    movlw   high (0xFFFF - WAKETIME)
    movwf   TMR0H
    movlw   low (0xFFFF - WAKETIME)
    movwf   TMR0L
    bsf     T0CON, TMR0ON       ; Start the timer
    return



;****************************************************************
; SUBROUTINE:   CSMA
;
; Description:  - Checks if the medium is free.
;               - If the medium is free, transmission is initiated.
;               - If it is not free, a random timeout is set.
;               - If after the timeout the medium is still not free,
;                   the timeout is doubled and reset.
;
;               - This routine is designed to be called once from normal
;                 code, and subsequently from interrupt on TIMER2.
;
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
CSMA:

; ----------------------------------------------------------------
; Perform a clear channel assesment.
_CSMA_CCA:
;   btfss   NCD                 ; Not used becase of noise problems.
    btfsc   clock_detect        ; Is there a valid clock?
    bra     _CSMA_WAIT          ; YES - set a timeout.
; -----------------------------------------------------------
; We are Clear To Send, initiate transmission.
_CSMA_CTS:


    ; Initialise the frame encoder
    call    RX_DISABLE          ; Turn off the decoder so it can't
                                ; change things while we are initialising.
    movlw   low preamble_len_l  ; Load up the preamble counter
    movwf   bytecount_l         ; ...
    movlw   low preamble_len_h  ; ...
    movwf   bytecount_h         ; ...
    clrf    tx_flags            ; Clear all flags.
    call    FRAME_ENCODER       ; Run the frame encoder once so the first byte is loaded.
    call    TX_ENABLE           ; Enable the transmitter.
    bcf     mac_csma_active     ; Hand control back to the preamble sampler.
    ; Set a new timeout, so it won't take 16.777 seconds for the preamble
    ; sampler routine to run again.
    bcf     T0CON, TMR0ON       ; Stop the timer
    nop                         ; Wait for it to stop.
    movff   sleeptime_h, TMR0H
    movff   sleeptime_l, TMR0L
    bsf     T0CON, TMR0ON       ; Start the timer
    return

; ---------------------------------------------------------
; Media is busy, set a random timeout with exponential backoff.
; Max wait Time = 2^retry_count milliseconds
_CSMA_WAIT:
    btg     TXLED

    ; Check how many times we have tried to transmit.
    movf    retry_counter,f     ; Has the retry counter reached zero?
    bz      _CSMA_GIVEUP        ; YES - give up.
    ; Decrement the retry counter, and multiply the timeout mask by two.
    decf    retry_counter       ; We are about to burn another retry.
    bsf     STATUS, C           ; Set the carry bit
    rlcf    timeout_mask_l      ; Left shift the mask.
    rlcf    timeout_mask_h      ; ...
    ; Set a random timeout
    movf    crclow, w           ; Get the crc low byte
    xorwf   TMR3H, w            ; Multiply it with the high byte of TMR3
    andwf   timeout_mask_l, w   ; Mask it.
    comf    WREG                ; Compliment it.
    movwf   timeout_l           ; Save it
    movf    crchigh, w          ; Get the crc high byte.
    xorwf   TMR3L, w            ; Multiply it with the low byte of TMR3.
    andwf   timeout_mask_h, w   ; Mask it.
    comf    WREG                ; Negate it.
    movwf   timeout_h           ; Save it.
    ; ####################################################################
    ; TO DO - set interim samples if the timeout is greater than the sleep
    ; interval required to maintain connectivity.
    ; ####################################################################
    bcf     T0CON, TMR0ON       ; Stop the timer
    nop                         ; Wait for it to stop.
    movff   timeout_h, TMR0H
    movff   timeout_l, TMR0L
    bsf     T0CON, TMR0ON       ; Start the timer
    return

; ---------------------------------------------------------
; We have reached the retry limit; Give up already.
_CSMA_GIVEUP:
    return

; ####################################################################################
; ####################################################################################
; ####################################################################################
; ####################################################################################
;
;                         MAC LAYER RECONFIGURATION ROUTINES
;
; ####################################################################################
; ####################################################################################
; ####################################################################################
; ####################################################################################
;**************************************************************
; SUBROUTINE:   SET_HTP_SLEEP
;
; Description:  Sets the sleep interval for high traffic (more frequent sampling).
; Precond'ns:
; Postcond'ns:
```

```
; Regs Used:
;****************************************************************
SET_HTP_SLEEP:
    ; Set the sleep interval for high traffic.
    movlw   low (0xFFFF - SLEEPTIME_H)
    movwf   sleeptime_l
    movlw   high (0xFFFF - SLEEPTIME_H)
    movwf   sleeptime_h
    return

;****************************************************************
; SUBROUTINE:   SET_HTP_PREAMBLE
;
; Description:  Sets the preamble length for high traffic (shorter preamble).
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
SET_HTP_PREAMBLE:
    ; Set the preamble length for high traffic.
    movlw   high PREAMBLELEN2
    movwf   preamble_len_h
    movlw   low PREAMBLELEN2
    movwf   preamble_len_l
    return

;****************************************************************
; SUBROUTINE:   SET_LTP_SLEEP
;
; Description:  Sets the sleep interval for low traffic (less frequent sampling).
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
SET_LTP_SLEEP:
    ; Set the sleep interval for low traffic.
    movlw   low (0xFFFF - SLEEPTIME_L)
    movwf   sleeptime_l
    movlw   high (0xFFFF - SLEEPTIME_L)
    movwf   sleeptime_h
    return

;****************************************************************
; SUBROUTINE:   SET_LTP_PREAMBLE
;
; Description:  Sets the preamble length for low traffic (longer preamble).
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
SET_LTP_PREAMBLE:
    ; Set the preamble length for low traffic.
    movlw   high PREAMBLELEN
    movwf   preamble_len_h
    movlw   low PREAMBLELEN
    movwf   preamble_len_l
    return

;****************************************************************
; SUBROUTINE:   TRAFFIC_CONTROL (ISR)
;
; Description:  Runs every 0.524288 seconds under TIMER1 Low Priority Interrupt.
;               Counts down any high traffic periods which are running, and
;               configures the preamble length and sleep interval accordingly.
;
; Precond'ns:
; Postcond'ns:
; Regs Used:
```

```
;***************************************************************
TRAFFIC_CONTROL:

    ; Service the TMR1 Interrupt.
    SERVICE_IRP PIR1, TMR1IF, PIE1, TMR1IE

    ; Count down the scheduled high traffic period.
    call    _SCHEDULED_HTP

    ; Count down the dynamic high traffic period.
    call    _DYNAMIC_HTP

    ; Check if anything is still forcing the high traffic (shorter) sleep interval.
    call    _CHECK_SLEEP_INTERVAL

    ; Check if anything is still forcing the high traffic (shorter) preamble length.
    call    _CHECK_PREAMBLE_LENGTH

    ; return
    return

; ------------------------------------------------------------------------------
; Count down the scheduled high traffic period.
_SCHEDULED_HTP:
    btfss   mac_schedule_htp        ; Are we in a scheduled high traffic period?
    return                          ; NO - return.
    decfsz  schedule_timer          ; Has the scheduled high traffic timer reached zero?
    return                          ; NO - return.
    ; End the scheduled high traffic period.
    bcf     mac_schedule_htp        ; We are no longer in a scheduled HTP.
    bcf     sleeping_con, scheduled ; Scheduled HTP is not forcing shorter sleeping.
    bcf     preamble_con, scheduled ; Scheduled HTP is not forcing shorter preamble.
    return

; ------------------------------------------------------------------------------
; Count down the dynamic high traffic period.
_DYNAMIC_HTP:
    btfss   mac_dynamic_htp         ; Are we in a dynamic high traffic period?
    return                          ; NO - return.
    decfsz  dynamic_timer           ; Has the dynamic high traffic timer reached zero?
    return                          ; NO - return.
    ; End the dynamic high traffic period.
    bcf     mac_dynamic_htp         ; We are no longer in a dynamic HTP.
    bcf     sleeping_con, dynamic   ; Dynamic HTP is not forcing shorter sleeping.
    bcf     preamble_con, dynamic   ; Dynamic HTP is not forcing shorter preamble.
    return

; ------------------------------------------------------------------------------
; Check if anything is still forcing the high traffic (shorter) sleep interval.
_CHECK_SLEEP_INTERVAL:
    tstfsz  sleeping_con        ; Is sleeping_con zero?
    return                      ; NO - return.
    ; YES - revert back to the low traffic (longer) sleeping interval.
    call    SET_LTP_SLEEP
    return

; ------------------------------------------------------------------------------
; Check if anything is still forcing the high traffic (shorter) preamble length.
_CHECK_PREAMBLE_LENGTH:
    ; Check if anything is still forcing a shorter preamble.
    tstfsz  preamble_con        ; Is preamble_con zero?
    return                      ; NO - return.
    ; YES - revert back to the low traffic (longer) preamble.
    call    SET_LTP_PREAMBLE
    return
```

```
;***************************************************************
; SUBROUTINE:   SET_SCHEDULED_HTP
;
; Description:  Starts a scheduled high traffic period for 9.961 seconds.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;***************************************************************
SET_SCHEDULED_HTP:

    ; Clear TIMER1 (High Traffic Timer) and start it
    clrf        TMR1H
    clrf        TMR1L
    bsf         T1CON, TMR1ON

    ; Load the timer preset (19 * 524288us = 9.96s)
    movlw   .19
    movwf   schedule_timer

    ; Set the flags for a scheduled high traffic period.
    bsf     mac_schedule_htp        ; Clear the flag to say we are in a scheduled HTP.
    bsf     sleeping_con, scheduled ; The scheduled HTP is forcing shorter sleeping.
    bsf     preamble_con, scheduled ; The scheduled HTP is forcing shorter preamble.
    ; Set the sleep interval for high traffic.
    call    SET_HTP_SLEEP

    ; Set the preamble length for high traffic.
    call    SET_HTP_PREAMBLE
    return

;***************************************************************
; SUBROUTINE:   SET_DYNAMIC_HTP
;
; Description:  Starts a scheduled high traffic period for 5.24 seconds.
; Precond'ns:   Set sleeping_con<dynamic> to force faster sampling.
;               Set preamble_con<dynamic> to force shorter preamble.
; Postcond'ns:
; Regs Used:
;***************************************************************
SET_DYNAMIC_HTP:

    ; Clear TIMER1 (High Traffic Timer) and start it
    clrf        TMR1H
    clrf        TMR1L
    bsf         T1CON, TMR1ON

    ; Load the timer preset (10 * 524288us = 5.24s)
    movlw   .10
    movwf   dynamic_timer

    ; Set the flags for a dynamic high traffic period.
    bsf     mac_dynamic_htp         ; Clear the flag to say we are in a dynamic HTP.
    ; Check if the dynamic HTP needs to force shorter sleeping.
    btfsc   sleeping_con, dynamic   ; Well does it?
    call    SET_HTP_SLEEP           ; YES - set shorter sleeping.
    ; Check if the dynamic HTP needs to force shorter sleeping.
    btfsc   preamble_con, dynamic   ; Well does it?
    call    SET_HTP_PREAMBLE        ; YES - set shorter sleeping.
    ; If we aren't in a sampling period, start one right now.
    btfss   mac_sampling
    bsf     INTCON, TMR0IF          ; SET!!! the TIMER0 Interrupt flag.
    return

;***************************************************************
; SUBROUTINE:   INT1_ISR (ISR)
;
; Description:  Starts a scheduled high traffic period for 9.961 seconds.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;***************************************************************
INT1_ISR:

    ; Service the INT1 Interrupt.
    SERVICE_IRP INTCON3, INT1IF, INTCON3, INT1IE

    ; Clear the RTC IRQ flags
    call    RTC_CLEAR_IRQS

    ; Start
    call    SET_SCHEDULED_HTP

    ; return
    return

;***************************************************************
; SUBROUTINE:   TX_SYNC_FRAME:
;
; Description:  Transmit a clock sync frame.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;***************************************************************
TX_SYNC_FRAME:

    ; Select the TX Buffer and clear it.
    BUF_SEL txbuf, TXBUFLEN     ; Select the transmit buffer.
    BUF_CLEAR                   ; Clear it.
    ; Put the frame type into the buffer
    movlw   CLOCKSYNC
    BUF_PUT

    ; Load the current clock value into the buffer
    call    RTC_GET_CLOCK

    ; Transmit the packet
    call    TX_PACKET

    ; Set low traffic mode (make sure our preamble is long enough for all nodes)
;   call    SET_LOW_TRAFFIC
    return



; #####################################################################################
; #####################################################################################
; #####################################################################################
; #####################################################################################
;
;                           FRAME ENCODING / DECODING ROUTINES
;
; #####################################################################################
; #####################################################################################
; #####################################################################################
; #####################################################################################


;***************************************************************
; Subroutine:   FRAME_ENCODER
;
; Description:  Encodes Frames. Should be called each time another
;               bit is encoded.
;
; Precond'ns:
;
```

```
; Postcond'ns:
;
; Regs Used:    WREG, FSR0
;*************************************************************
FRAME_ENCODER:

; -------------------------------------------------------------
; DECISION: Check if the manchester encoder is ready for the next byte,
; and if we have a byte to give.
    btfss   txbyte_empty        ; Is the transmit buffer empty?
    return                      ; NO - return.
    btfsc   frame_sent          ; Do we have any data to transmit?
    bra     _FRAME_SENT         ; NOOOOOOO!
    bcf     txbyte_empty        ; YES - clear the flag and continue.
; -------------------------------------------------------------
; DECISION: Check if we have transmitted the preamble and SOF yet.
    btfsc   framelen_sent       ; Have we sent the frame length yet?
    bra     _TX_DATABYTE        ; YES - transmit a data byte.
                                ; NO - continue.
; -------------------------------------------------------------
; DECISION: Check if we have transmitted the preamble and SOF yet.
    btfsc   preamble_sent       ; Have we sent the SOF yet?
    bra     _TX_FRAMELEN        ; YES - transmit the frame length.
                                ; NO - continue.
; -------------------------------------------------------------
; DECISION: Check if we have transmitted enough preamble bytes yet.
    tstfsz  bytecount_h         ; Is the bytecounter zero?
    bra     _TX_PREAMBLE        ; NO - send a preamble byte.
    tstfsz  bytecount_l         ; Is the bytecounter zero?
    bra     _TX_PREAMBLE        ; NO - send a preamble byte.
    bra     _TX_SOF             ; YES - send a SOF byte.
; -------------------------------------------------------------
; PROCESS: Send a preamble byte.
_TX_PREAMBLE:
    movlw   PREAMBLE_BYTE       ; Copy the preamble byte ...
    movwf   txbyte              ; ... to the transmit reigster.
    decf    bytecount_l         ; Deccrement the counter.
    btfss   STATUS, C           ; ...
    decf    bytecount_h         ; ...
    return

; -------------------------------------------------------------
; PROCESS: Send a SOF byte.
_TX_SOF:
    movlw   SOF_BYTE            ; Copy the SOF byte ...
    movwf   txbyte              ; ... to the transmit reigster.
    bsf     preamble_sent       ; Set the flag to say we have sent the preamble.
    return

; -------------------------------------------------------------
; PROCESS: Send a Frame Length byte.
_TX_FRAMELEN:
    BUF_SEL txbuf, TXBUFLEN     ; Select the transmit buffer.
    BUF_GETEND                  ; Get the size of the buffer.
    movwf   txbyte              ; Place it in the transmit reigster.
    bsf     framelen_sent       ; Set flag to say that we have sent the frame length.
    movlw   0x00                ; Reset the buffer cursor.
    BUF_SETCURSOR               ; ...
    return

; -------------------------------------------------------------
; PROCESS: Send a DATA byte.
_TX_DATABYTE:
    BUF_SEL txbuf, TXBUFLEN     ; Select the transmit buffer.
    BUF_GET                     ; Get a byte from the buffer.
    movwf   txbyte              ; Place it in the transmit reigster.
    BUF_SNEOF                   ; Was that the last byte in the buffer?
    bsf     frame_sent          ; YES - set the frame sent flag.
    return                      ; NO - return.
_FRAME_SENT:

 #ifdef BAPSA
    ; BASIC ADAPTIVE PREAMBLE SAMPLING ALGORITHM TEST:
    ; Switch to high traffic rate for 5 seconds.
    btfss   NTXEN                   ; Has the receiver been disabled yet?
    return                          ; NO - return.
    bsf     sleeping_con, dynamic   ; change the sleeping interval.
    bsf     preamble_con, dynamic   ; change the preamble length.
    call    SET_DYNAMIC_HTP         ; Just do it.
 #endif
    return


;***************************************************************
; SUBROUTINE:   FRAME_DECODER:
;
; Description:  - Decodes frames.
;               - Should be called every time a new bit is decoded.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;***************************************************************
FRAME_DECODER:

; ------------------------------------------------------------------------------
; DECISION: Check if we are currently in the process of receiving a frame.
; got_sof is clear: Run the SOF detect routine.
; got_sof is set: Run the frame receive routine.
    btfss   got_sof             ; Have we already detected the SOF pattern?
    bra     _GET_SOF            ; NO - look for the SOF pattern.
; ------------------------------------------------------------------------------
; DECISION: Check if we have received a byte.
; YES - process the byte.
; NO - return.
    btfss   rxbyte_full         ; Have we decoded a full byte?
    return                      ; NO - wait till we have.
    bcf     rxbyte_full         ; YES - Clear the flag ...
; ------------------------------------------------------------------------------
; DECISION: Check if we have received the frame length byte yet.
; got_framelen is clear: Run the Frame Length byte receive routine.
; got_framelen is set: Run the data byte receive routine.
    btfss   got_framelen        ; Have we received the frame length yet?
    bra     _GET_FRAMELEN       ; NO - receive the frame length byte.
    ;bra    _GET_DATABYTE       ; YES - receive a data byte.
; ------------------------------------------------------------------------------
; PROCESS: Read the data byte into the buffer.
_GET_DATABYTE:
    BUF_SEL rxbuf, RXBUFLEN     ; Select the receive buffer.
    movf    rxbyte, w           ; Load up the byte we just received.
    BUF_PUT                     ; Put it into the buffer. <BUFPUT in buffers.asm>
    BUF_SNFULL                  ; Is the buffer full?
    bra     _GOT_FRAME          ; YES: we have received as much of the frame as possible.
    dcfsnz  bytecount_l         ; Have we received all the bytes in the frame?
    bra     _GOT_FRAME          ; YES - we have received the entire frame.
    return                      ; NO - keep receiving bytes.
; ------------------------------------------------------------------------------
; PROCESS: Check if we have received the start of frame pattern.
_GET_SOF:

    ; Don't try and detect the SOF until we have received a new bit.
    movlw   0x00                ;
    cpfsgt  bitcount            ; Is bitcount > 0?
    return                      ; NO - return.
    clrf    bitcount            ; YES - clear bitcount and try to detect SOF.
    ; We clear the bitcount to do two things:
```

```
    ; 1) So we can determine if a new bit has arrived by testing if bitcount != zero.
    ; 2) It stops the bit counter getting to 8, therefore the physical layer
    ; cannot assert the clock_detect flag. If we have not yet detected the SOF,
    ; we only want to assert the clock detect if we detect a valid preamble.
    ; Try and detect the preamble byte.
    movlw   PREAMBLE_BYTE       ; Load up the preamble byte.
    cpfseq  shiftreg_l          ; Compare with the low byte of the shiftreg.
    return                      ; It wasn't equal, so just try again next time.
    bsf     clock_detect        ; We have preamble - set the clock detect flag.
    bsf     RXLED

    ; We have the preamble byte in the low byte of the shiftreg,
    ; so try and detect the SOF (start of frame) byte.
    movlw   SOF_BYTE            ; Load up the start byte.
    cpfseq  shiftreg_h          ; Compare with the high byte of the shiftreg.
    return                      ; It wasn't equal, so just try again next time.
    ; We have a winner!
    bsf     got_sof             ; Set the Got Start Of Frame flag.
    clrf    bitcount            ; Reset the bit counter.
    bsf     TXLED               ; Turn on the RED LED to say we got the SOF.
    return


; -------------------------------------------------------------------------------
; PROCESS: Read the frame length, and store it so we know how many bytes to expect.
; Set a flag to say we know how many bytes to receive, and reset the buffer.
_GET_FRAMELEN:
    movff   rxbyte, bytecount_l ; Save the frame length
    bsf     got_framelen        ; Set the flag to say we know the frame length
    BUF_SEL     rxbuf, RXBUFLEN ; Select the Receive Buffer.
    BUF_CLEAR                   ; Reset the buffer.
    return

; -------------------------------------------------------------------------------
; We have received the entire frame - Verify the CRC.
_GOT_FRAME:
    bcf     TXLED

    ; Mark the end of the receive buffer
    ; (Buffer should still be selected).
    BUF_MARKEND

    ; Calculate the CRC on the receive buffer.
    call    CALC_CRC            ; Calculate the CRC on the buffer.
    ; Verify that the CRC for the packet is zero.
    tstfsz  crchigh             ; Is the high byte zero?
    bra     RESET_DECODER       ; NO - CRC is bad, reset the decoder.
    tstfsz  crclow              ; Is the low byte zero?
    bra     RESET_DECODER       ; NO - CRC is bad, reset the decoder.
; The CRC was good.
_CRC_GOOD:

 #ifdef BAPSA
    ; BASIC ADAPTIVE PREAMBLE SAMPLING ALGORITHM:
    ; Sample at the high traffic rate for 5 seconds, but don't change
    ; the preamble.
    bsf     sleeping_con, dynamic   ; Only change the sleeping interval
    call    SET_DYNAMIC_HTP         ; Just do it.
 #endif

    ; Remove the last two bytes (the CRC) from the buffer.
    BUF_GETEND                  ; Get the end pointer.
    sublw   d'2'                ; Shorten the buffer by 2
    BUF_MARKEND                 ; Set the end pointer.
    ; Go back to the beginning of the buffer
    movlw   0x00
    BUF_SETCURSOR
    ; Get the first byte (frame type) and check it
    BUF_GET
    sublw   CLOCKSYNC           ; Is it a clocksync packet?
    bz      _UPDATE_CLOCK       ; YES - sync the clock.
    ; NO - its just a normal packet.
    bsf     packet_received     ; Set the packet received flag.
    return                      ; Go pikachu!
_UPDATE_CLOCK:
    call    RTC_SET_CLOCK
    call    RESET_DECODER
    return

; Reset the decoder - start looking for a SOF again.
RESET_DECODER:
    clrf    rx_flags            ; Clear all flags.
    clrf    bytecount_l         ; Clear byte counter.
    clrf    bytecount_h         ; Clear byte counter.
    ; If we are aren't in a sampling window, turn off the receiver.
    btfss   mac_sampling        ; Are we sampling?
    call    RX_DISABLE          ; NO - disable the receiver.
    return                      ; Turns the node into a toadstool.
;*********************************************************************************
; END OF CODE
 END
```

E.3 PERIPHERALS

Table E.10 - SPI.ASM

```
;**********************************************************************
; SPI MODULE
;
; Version 0.10
; 17/07/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;**********************************************************************
;
; Description:
;   Provides routines for initialization and transfer on SPI.
;
; Functions:
;
; Dependencies:
;
; Resources Used:
;   - MSSP Module.
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;**********************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;**********************************************************************
; CONFIGURATION CONSTANTS
;**********************************************************************
; CONSTANTS
;***************************************************************************
; IMPORTED VARIABLES
 UDATA_ACS
;***************************************************************************
; GLOBAL VARIABLES
;***************************************************************************
; LOCAL VARIABLES
;***************************************************************************
; IMPORTED SUBROUTINES
;***************************************************************************
; EXPORTED SUBROUTINES
 global SPI_INIT, SPIRW

;***************************************************************************
; START OF CODE
 CODE

;****************************************************************
; Subroutine:   SPI_INIT
;
; Description:  Initialises the MSSP in SPI mode 3.
; Regs Used:    WREG, TRISC, SSP Registers.
;****************************************************************
SPI_INIT:
    ; Initialise the MSSP in SPI mode
    movlw   B'00100000'
    ;        '--1-----'     ; Enables SPI and configure SCK, SDO, SDI, and SS
    ;        '---0----'     ; Clock idle low
    ;        '----0000'     ; SPI Master mode, clock = FOSC/4
    movwf   SSPCON1         ; ...
    clrf    SSPCON2         ; This register only used for I2C mode.
    movlw   b'11000000'
    ;        '1-------'     ; Input data sampled at end of data output time
    ;        '-1------'     ; Data transmitted on rising edge of SCK
    movwf   SSPSTAT

    ; Configure SPI pins
    CONFIG_SPI_PINS
    return                  ; Done.
;***************************************************************
; Subroutine:   SPIRW
;
; Description:  Transfers 1 byte on the SPI bus.
; Precond'n:    WREG contains the data to be transmitted.
; Postcond'n:   WREG contains the data that was received.
; Regs Used:    WREG, SSPBUF
;***************************************************************
SPIRW:
    movwf   SSPBUF              ; Move data to be transmitted into SSPBUF.
_SPIRW_LOOP:
    btfss   SSPSTAT, BF         ; Check if transfer is complete.
    bra     _SPIRW_LOOP         ; NO - Loop.
    movf    SSPBUF, W           ; YES - Move the received data into WREG.
    return                      ; Done.

 End
```

Table E.11 - RTC.INC

```
;*************************************************************************
; Real Time Clock Module Header File
; Version 1.00
; 16/09/2005
; Initialisation
 extern RTC_INIT
; Random Crap
 extern RTC_CLEAR_IRQS, RTC_GET_STATUS, RTC_GET_CLOCK, RTC_SET_CLOCK
```

Table E.12 - RTC.ASM

```
;*************************************************************************
; RTC MODULE
;
; Version 0.10
; 17/07/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;*************************************************************************
;
; Description:
;   Provides routines for accessing the Maxim-Dallas DS1305 Real Time Clock Chip..
;
; Functions:
;
; Dependencies:
;   SPI.asm
;   Buffers.inc
;
; Resources Used:
;   - MSSP Module.
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;*************************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;*************************************************************************
; CONFIGURATION CONSTANTS
;*************************************************************************
; CONSTANTS
; Internal Addresses
CLOCK       equ 0x00    ; Address of the Clock Registers
ALARM0      equ 0x07    ; Address of Alarm 0 Registers
ALARM1      equ 0x0B    ; Address of Alarm 1 Registers
CONTROL     equ 0x0F    ; Control Register
RTCSTATUS   equ 0x10    ; Status Register
CHARGER     equ 0x11    ; Trickle Charger Register
USERRAM     equ 0x20    ; User RAM 0x20 - 0x7F
; Offsets
SECONDS     equ 0x00    ; Seconds Offset
```

```
MINUTES     equ 0x01    ; Clock Minutes Offset
HOURS       equ 0x02    ; Clock Hours Offset
DAY         equ 0x03    ; Clock Day Offset
DATE        equ 0x04    ; Clock Date Offset
MONTH       equ 0x05    ; Clock Month Offset
YEAR        equ 0x06    ; Clock Year Offset
WRITE       equ 0x80    ; Write offset


;******************************************************************************
; IMPORTED VARIABLES
 UDATA_ACS
;******************************************************************************
; GLOBAL VARIABLES
;******************************************************************************
; LOCAL VARIABLES
;******************************************************************************
; IMPORTED SUBROUTINES
 #include "Buffers.inc"

; From <SPI.asm>
 extern SPI_INIT, SPIRW

;******************************************************************************
; EXPORTED SUBROUTINES
 global RTC_INIT
 global RTC_CLEAR_IRQS, RTC_GET_STATUS, RTC_GET_CLOCK, RTC_SET_CLOCK

;******************************************************************************
; START OF CODE
 CODE

;****************************************************************
; Subroutine:   RTC_INIT
;
; Description:  Initialises the Real Time Clock.
; Precond'ns:   SPI bus has been initialised.
; Postcond'ns:
; Regs Used:    WREG
;****************************************************************
RTC_INIT:

    ; Set up RTC Pins
    CONFIG_RTC_PINS

    ; Set up INT1
    bcf     INTCON2, INTEDG1    ; Trigger on falling edge.
    bcf     INTCON3, INT1IP     ; Low Priority
    bcf     INTCON3, INT1IF     ; Clear the IRQ flag
    bsf     INTCON3, INT1IE     ; Enable the Interrupt
    ; Enable Writes
    bsf     RTC_CS              ; Start an operation.
    movlw   CONTROL + WRITE     ; Load the CONTROL register write address
    call    SPIRW               ; ...
    movlw   b'00000000'         ; Load a byte to enable writes.
    ;        '-0------'         ; Disable Write Protect.
    call    SPIRW               ; ...
    bcf     RTC_CS              ; End the operation.
    nop


    ; Set up the control register
    bsf     RTC_CS              ; Start an operation.
    movlw   CONTROL + WRITE     ; Load the CONTROL register write address
    call    SPIRW               ; ...
    movlw   b'00000011'
    ;        '0-------'         ; Enable the oscillator.
    ;        '-0------'         ; Disable Write Protect.
    ;        '-----0--'         ; Both Alarm 0 and Alarm 1 activate /INT0.
    ;        '------1-'         ; Enable Alarm 1 Interrupt.
    ;        '-------1'         ; Enable Alarm 0 Interrupt.
    call    SPIRW               ; ...
    bcf     RTC_CS              ; End the operation.
    nop

    ; Set up Alarm 0 to interrupt when seconds = 00
    bsf     RTC_CS              ; Start an operation.
    movlw   ALARM0 + WRITE      ; Load the Alarm 0 write address
    call    SPIRW               ; ...
    movlw   0x00                ; Load 00 seconds (BCD)
    call    SPIRW               ; Write to the seconds register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the minutes register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the hours register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the days register
    bcf     RTC_CS              ; End the operation.
    nop

    ; Set up Alarm 1 to interrupt when seconds = 30
    bsf     RTC_CS              ; Start an operation.
    movlw   ALARM1 + WRITE      ; Load the Alarm 1 write address
    call    SPIRW               ; ...
    movlw   0x30                ; Load 30 seconds (BCD)
    call    SPIRW               ; Write to the seconds register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the minutes register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the hours register
    movlw   b'10000000'         ; Set the bit mask
    call    SPIRW               ; Write to the days register
    bcf     RTC_CS              ; End the operation.
    nop

;   return
; DEBUGGING CODE...
;
;_READCLOCK:
;    ; Do some reads
;    bsf     RTC_CS              ; Start an operation.
;    movlw   CLOCK               ; Load the clock read address
;    call    SPIRW               ; ...
;    call    SPIRW               ; Read Seconds. (CONTROL)
;    call    SPIRW               ; Read Minutes. (STATUS)
;    call    SPIRW               ; Read Hours.
;    call    SPIRW               ; Read Day.
;    call    SPIRW               ; Read Date.
;    call    SPIRW               ; Read Month.
;    call    SPIRW               ; Read Year.
;    bcf     RTC_CS              ; End the operation.
;    nop
;
;    return

;****************************************************************
; Subroutine:   CRTC_CLEAR_IRQS
;
; Description:  Clears both IRQ flags in the STATUS register by reading
;               from the DAY register of ALARM0 and the SECONDS register of ALARM1
; Precond'ns:
; Postcond'ns:
; Regs Used:    WREG
;****************************************************************
RTC_CLEAR_IRQS:
    ; Read 0x0A (ALARM0-DAY) and 0x0B (ALARM1-SECONDS).
    bsf     RTC_CS              ; Start an operation.
    movlw   ALARM0 + DAY        ; Load the Alarm 0 DAY read address
    call    SPIRW               ; ...
    call    SPIRW               ; Read ALARM0-DAY
    call    SPIRW               ; Read ALARM1-SECONDS
    bcf     RTC_CS              ; End the operation.
    nop
    return


;****************************************************************
; Subroutine:   RTC_GET_STATUS
;
; Description:  Get the status register.
; Precond'ns:
; Postcond'ns:
; Regs Used:    WREG
;****************************************************************
RTC_GET_STATUS:
    bsf     RTC_CS              ; Start an operation.
    movlw   RTCSTATUS           ; Load the STATUS register read address
    call    SPIRW               ; ...
    call    SPIRW               ; Read one byte.
    bcf     RTC_CS              ; End the operation.
    return

;****************************************************************
; Subroutine:   RTC_GET_CLOCK
;
; Description:  Read the clock into the currently selected buffer.
; Precond'ns:   There are at least 7 free bytes after the current cursor location.
; Postcond'ns:
; Regs Used:    WREG
;****************************************************************
RTC_GET_CLOCK:
    bsf     RTC_CS              ; Start an operation.
    movlw   CLOCK               ; Load the CLOCK register read address
    call    SPIRW               ; ...
    call    SPIRW               ; Read Seconds and place it in the buffer.
    addlw   d'01'               ; Compensate for the 1 second delay
        BUF_PUT
    call    SPIRW               ; Read Minutes and place it in the buffer.
    BUF_PUT
    call    SPIRW               ; Read Hours and place it in the buffer.
    BUF_PUT
    call    SPIRW               ; Read Day and place it in the buffer.
    BUF_PUT
    call    SPIRW               ; Read Date and place it in the buffer.
    BUF_PUT
    call    SPIRW               ; Read Month and place it in the buffer.
    BUF_PUT
    call    SPIRW               ; Read Year and place it in the buffer.
    BUF_PUT
    BUF_MARKEND                 ; Mark the end of the buffer.
    bcf     RTC_CS              ; End the operation.
    return

;****************************************************************
; Subroutine:   RTC_SET_CLOCK
;
; Description:  Set the clock from the currently selected buffer.
; Precond'ns:   The buffer contains 7 clock bytes at the current cursor location.
; Postcond'ns:
; Regs Used:    WREG
;****************************************************************
RTC_SET_CLOCK:
    bsf     RTC_CS              ; Start an operation.
    movlw   CLOCK + WRITE       ; Load the CLOCK register write address
    call    SPIRW               ; ...
    BUF_GET                     ; Get the Seconds byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Minutes byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Hours byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Day byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Date byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Month byte and write it to the RTC chip.
    call    SPIRW
    BUF_GET                     ; Get the Year byte and write it to the RTC chip.
    call    SPIRW
    bcf     RTC_CS              ; End the operation.
    return

 END
```

Table E.13 - ROUTECACHE.ASM

```
;**************************************************************************
; ROUTE CACHE MODULE
;
; Version 0.10
; 17/07/2005
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;**************************************************************************
;
; Description:
;
;   Use locations in FRAM which are 4 bytes long.
;
; Dependencies:
;   a) SPI.ASM  -   SPI routines and initlialisation.
;
; Resources Used:
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;**************************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;**************************************************************************
; CONFIGURATION CONSTANTS
;**************************************************************************
; CONSTANTS
; FRAM Op-codes
OP_WRSR     equ 0x01        ; Write Status Register
OP_WRITE    equ 0x02        ; Write Memory Data
OP_READ     equ 0x03        ; Read Memory Data
OP_WRDI     equ 0x04        ; Write Disable
OP_RDSR     equ 0x05        ; Read Status Register
OP_WREN     equ 0x06        ; Set Write Enable Latch
;*******************************************************************************
; IMPORTED VARIABLES
 UDATA_ACS
;*******************************************************************************
; GLOBAL VARIABLES
; Fields in each route record
source_addr     res 1
dest_addr       res 1
next_hop        res 1
tx_power        res 1

 global source_addr, dest_addr, next_hop, tx_power

;*******************************************************************************
; LOCAL VARIABLES
cursor_l        res 1
cursor_h        res 1
end_l           res 1
end_h           res 1

Tableptr       res 1       ; Routing table pointer
;*******************************************************************************
```

```
; IMPORTED SUBROUTINES
; From SPI.asm
 extern SPIRW

;********************************************************************************
; EXPORTED SUBROUTINES
 global TEST_ROUTECACHE, FRAM_INIT

;********************************************************************************
; START OF CODE
 CODE

;*****************************************************************
; SUBROUTINE:   FRAM_INIT
;
; Description:  - Initialise the FRAM.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;*****************************************************************
FRAM_INIT:

    ; Set up FRAM pins.
    CONFIG_FRAM_PINS
    return

;*****************************************************************
; MACRO:        FRAM_READ
;
; Description:  Tell the FRAM we want to read from the specified address.
; Arguments:
; Precond'ns:
; Postcond'ns:
; Regs Used:
;*****************************************************************
FRAM_READ macro addr_h, addr_l

    ; Start an FRAM operation (Set /CS low)
    bcf     FRAM_NCS

    ; Send the READ op-code to the FRAM.
    movlw   OP_READ         ; Tell FRAM we want to read.
    call    SPIRW           ; ...
    ; Send the memory address to the FRAM.
    movf    addr_h, W       ; From this high memory location.
    call    SPIRW           ; ...
    movf    addr_l, W       ; And this low memory location.
    call    SPIRW           ; ...
 endm

;*****************************************************************
; MACRO:        FRAM_WRITE
;
; Description:  Tell the FRAM we want to write at the specified address.
; Arguments:
; Precond'ns:
; Postcond'ns:
; Regs Used:
;*****************************************************************
FRAM_WRITE macro addr_h, addr_l

    ; Send the WRITE ENABLE op-code to the FRAM.
    bcf     FRAM_NCS        ; Start an FRAM operation (Set /CS low)
    movlw   OP_WREN         ; Enable write operations
    call    SPIRW           ; ...
    bsf     FRAM_NCS        ; Terminate the FRAM operation (Set /CS high)
    ; Wait a little while before starting the next command.
    nop
    nop
    bcf     FRAM_NCS        ; Start an FRAM operation (Set /CS low)
    ; Send the WRITE op-code to the FRAM.
    movlw   OP_WRITE        ; Tell FRAM we want to write.
    call    SPIRW           ; ...
    ; Send the memory address to the FRAM.
    movf    addr_h, W       ; To this high memory location.
    call    SPIRW           ; ...
    movf    addr_l, W       ; And this low memory location.
    call    SPIRW           ; ...
 endm


;****************************************************************
; SUBROUTINE:   GET_END
;
; Description:  - Get the end pointer.
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
GET_END:

    ; Put the address 0x0000 on the software stack.
    movlw       0x00
    movwf       PREINC2
    movwf       PREINC2

    ; Tell the FRAM we want to start reading from 0x0000
    FRAM_READ   POSTDEC2, POSTDEC2

    ; Read 2 bytes from the FRAM.
    call    SPIRW               ; Read the low byte of the end address.
    movwf   end_l               ; ...
    call    SPIRW               ; Read the high byte of the end address.
    movwf   end_h               ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS
    return

;****************************************************************
; SUBROUTINE:   SET_END
;
; Description:  - Mark the current cursor position as the new end of the table
; Precond'ns:
; Postcond'ns:
; Regs Used:
;****************************************************************
SET_END:

    ; Put the address 0x0000 on the software stack.
    movlw       0x00
    movwf       PREINC2
    movwf       PREINC2

    ; Tell the FRAM we want to start writing at 0x0000
    FRAM_WRITE  POSTDEC2, POSTDEC2

    ; Write 2 bytes of data to the FRAM.
    movf    end_l, W            ; Write the low byte of the end address.
    call    SPIRW               ; ...
    movf    end_h, W            ; Write the high byte of the end address.
    call    SPIRW               ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS
    return

;***************************************************************
; SUBROUTINE:   ROUTE_ADD
;
; Description:  - Adds a route to the routing table.
; Precond'ns:   -
; Postcond'ns:  -
; Regs Used:
;***************************************************************
ROUTE_ADD:

    ; Get end address of the data stored in the FRAM, which is where
    ; we want to store the new record.
    call    GET_END

    ; Because RAM always comes in sizes so that the full address space is used,
    ; we can check if we have gone past the end of the address space by testing
    ; a single bit. In this case, we have a 2048 byte RAM, so the highest address
    ; is 0x07FF, or 0000 0111 1111 1111. When we go one past this address, it will
    ; be 0x0800, or 0000 1000 0000 0000. Therefore, by testing bit 3 of the high
    ; byte of the address, we can check if we have reached the end of the memory space.
    btfsc   end_h, 3            ; Are we at 0x800?
    return                      ; YES - memory is full, exit without adding route.
    ; Tell the FRAM we want to start writing at the end of the table.
    FRAM_WRITE  end_h, end_l

    ; Write the record to the FRAM.
    movf    source_addr, W      ; Load the source address and write it.
    call    SPIRW               ; ...
    movf    dest_addr, W        ; Load the dest address and write it.
    call    SPIRW               ; ...
    movf    next_hop, W         ; Load the next hop and write it.
    call    SPIRW               ; ...
    movf    tx_power, W         ; Load the transmit power and write it.
    call    SPIRW               ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS

    ; Mark the new end address of the data stored in FRAM.
    movlw   0x04                ; Add 4 to the end address.
    addwf   end_l               ; ...
    movlw   0x00                ; ...
    addwfc  end_h               ; ...
    call    SET_END             ; Set the end address in the FRAM chip.
    return



;***************************************************************
; SUBROUTINE:   ROUTE_ERASE
;
; Description:  - Delete the route currently pointed to by the cursor.
; Precond'ns:   -
; Postcond'ns:  -
; Regs Used:
;***************************************************************
ROUTE_ERASE:

    ; We will delete the record, and move the last record into the empty space.
    ; Get the address of the last record in the table, which is at (END - 4)
    call    GET_END             ; Get the end address stored in the FRAM.
    movlw   0x04                ; Subtract 4 from the end address.
    subwf   end_l               ; ...
    movlw   0x00                ; ...
    subwfb  end_h               ; ...
    ; Mark this as the new end of the table.
    call    SET_END
    ; Tell the FRAM we want to start reading the last record in the table.
    FRAM_READ   end_h, end_l

    ; Read the record at the end of the table.
    call    SPIRW               ; Read the source address and save it.
    movwf   source_addr         ; ...
    call    SPIRW               ; Read the destination address and save it.
    movwf   dest_addr           ; ...
    call    SPIRW               ; Read the next hop and save it.
    movwf   next_hop            ; ...
    call    SPIRW               ; Read the transmit power and save it.
    movwf   tx_power            ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS

    ; Tell the FRAM we want to start writing at the current cursor position.
    FRAM_WRITE  cursor_h, cursor_l

    ; Write over the record at the current cursor position.
    movf    source_addr, W      ; Load the source address and write it.
    call    SPIRW               ; ...
    movf    dest_addr, W        ; Load the dest address and write it.
    call    SPIRW               ; ...
    movf    next_hop, W         ; Load the next hop and write it.
    call    SPIRW               ; ...
    movf    tx_power, W         ; Load the transmit power and write it.
    call    SPIRW               ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS
    return



;****************************************************************
; SUBROUTINE:   ROUTE_SEARCH
;
; Description:  - Search for a route in the routing table.
; Precond'ns:   -
; Postcond'ns:  -
; Regs Used:
;****************************************************************
ROUTE_SEARCH:

    ; Make sure we have the correct end pointer for the table.
    call    GET_END

    ; We will do the entire search with the one FRAM command, so we
    ; will use cursor_l and cursor_h as a local memory tracker.
    clrf    cursor_h    ; Reset the cursor to the first record in the table.
    movlw   0x04        ; ...
    movwf   cursor_l    ; ...
    ; Tell the FRAM we want to start reading the first record in the table.
    FRAM_READ   cursor_h, cursor_l

_ROUTE_SEARCH_LOOP:

    ; Check if we have reached the end of the table.
    movf    cursor_h, W         ; Compare the high bytes of the cursor and
    cpfseq  end_h               ; the end pointer. Are they equal?
    bra     _ROUTE_SEARCH_CHK   ; NO - read the current record.
    movf    cursor_l, W         ; Compare the low bytes of the cursor and
    cpfseq  end_l               ;   the end pointer. Are they equal?
    bra     _ROUTE_SEARCH_CHK   ; NO - read the current record.
    ; We didn't find the requested route in the table.
    bsf     FRAM_NCS            ; Terminate the FRAM operation (Set /CS high)
    return
```

```
_ROUTE_SEARCH_CHK:

    ; Check source address of the record.
    call    SPIRW               ; Read the source address.
    cpfseq  source_addr         ; Is it the source address we're looking for?
    bra     _SKIP_3             ; NO - skip the next 3 bytes and try again.
    ; Check destination address of the record
    call    SPIRW               ; Read the destination address.
    cpfseq  dest_addr           ; Is it the destination address we're looking for?
    bra     _SKIP_2             ; NO - skip the next 2 bytes and try again.
    ; We have found a route - read the hop count and tx power
    call    SPIRW               ; Read the next hop and save it.
    movwf   next_hop            ; ...
    call    SPIRW               ; Read the transmit power and save it.
    movwf   tx_power            ; ...
    ; Terminate the FRAM operation (Set /CS high)
    bsf     FRAM_NCS
    return

_SKIP_3:
    ; Skip the next 3 memory locations and check the next record.
    call    SPIRW
_SKIP_2:
    ; Skip the next 2 memory locations and check the next record.
    call    SPIRW
    call    SPIRW

    ; Advance the LOCAL memory tracker to the next record.
    movlw   0x04
    addwf   cursor_l
    movlw   0x00
    addwfc  cursor_h
    bra     _ROUTE_SEARCH_LOOP


;***************************************************************
; SUBROUTINE:   ROUTE_ERASE_ALL
;
; Description:  - Delete all routes.
; Precond'ns:   -
; Postcond'ns:  -
; Regs Used:
;***************************************************************
ROUTE_ERASE_ALL:

    ; Set the beginning of the table as the end.
    clrf    end_h               ; Say that the first record is the end of the table.
    movlw   0x04                ; ...
    movwf   end_l               ;
    ; Write the end address of the table data.
    call    SET_END
    return

 global TEST_ROUTECACHE

TEST_ROUTECACHE:
    call    FRAM_INIT
    call    ROUTE_ERASE_ALL
    movlw   0x11
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x22
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x33
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x44
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x55
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x66
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x77
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x88
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    movlw   0x99
    movwf   source_addr
    movwf   dest_addr
    movwf   next_hop
    movwf   tx_power
    call    ROUTE_ADD
    call    ROUTE_SEARCH
    bra     TEST_ROUTECACHE
    return

 end
```

Table E.14 - SWTIMERS.ASM

```
;***********************************************************************
; SOFTWARE TIMERS MODULE
;
; Version x.xx
; dd/mm/yyyy
;
; Li-Wen Yip
; Ad Hoc Radio Networking Research Project
; School Of Engineering
; James Cook University
;
;***********************************************************************
;
; Description:
; - This module provides one-shot timers which may be used for timeouts.
; - Each timer consists of a 8/16-bit counter, and a boolean flag indicating
;   whether the timer is running.
; - To start the timer, load the desired timeout into the counter, and set the timer
flag.
; - Each time a TIMER2 interrupt occurs, the counter will be decremented if the
;   timer is running. When the counter reaches zero, the flag will be cleared and the
;   timer will stop.
;
; Dependencies:
;
; Resources Used:
;  - TIMER2 (Exclusively)
;
; Things you must do to use this module:
;
; What you need to understand to work with this code:
;
; Notes:
;
;***********************************************************************
; MASTER HEADER FILE
 #include "MasterHeader.inc"

;***********************************************************************
; CONFIGURATION CONSTANTS
; The timebase for all software timers, as a multiple of 4us.
TIMEBASE equ d'250'             ; Timebase = 250 x 4us = 1ms
;***********************************************************************
; CONSTANTS
;****************************************************************************
; IMPORTED VARIABLES
 UDATA_ACS
;****************************************************************************
; GLOBAL VARIABLES
TMRL4   res 1                   ; Timer4 Low Register.
TMRH4   res 1                   ; Timer4 High Register.
TMRF4   res 1                   ; Timer4 Running.
 global TMRL4, TMRH4, TMRF4

TMRL5   res 1                   ; Timer5 Low Register.
TMRH5   res 1                   ; Timer5 High Register.
TMRF5   res 1                   ; Timer5 Running.
 global TMRL5, TMRH5, TMRF5


;****************************************************************************
; LOCAL VARIABLES
;****************************************************************************
; IMPORTED SUBROUTINES
;****************************************************************************
```

```
; EXPORTED SUBROUTINES
 global SWTIMERS_INIT, SWTIMERS_ISR

;******************************************************************************
; START OF CODE
 CODE


;****************************************************************
; SUBROUTINE:   SOFTWARE_TIMERS_INIT
;
; Description:  Initialises TIMER2 as the timebase for the software timers.
; Precond'ns:   None.
; Postcond'ns:  TIMER2 is configured.
; Regs Used:    TIMER2 regs, WREG.
;****************************************************************
SWTIMERS_INIT:
    bcf     PIR1, TMR2IE        ; Disable the interrupt.
    movlw   TIMEBASE            ; Set TMR2 period
    movwf   PR2                 ; ...
    movlw   b'00000101'         ; TIMER2 Config byte.
    ;        '-0000---'         ; 0:0 Postscale
    ;        '-----1--'         ; Timer2 ON
    ;        '------01'         ; 1:4 Prescale
    movwf   T2CON               ; ...
    bcf     IPR1, TMR2IP        ; Set TIMER2 to low priority interrupt.
    bcf     PIR1, TMR2IF        ; Clear the flag so it doesn't irp immediately.
    bsf     PIE1, TMR2IE        ; Enable the interrupt.
    return



;****************************************************************
; MACRO:        TIMER_ISR
;
; Description:  Decrements the timer counter. If the timer and period
;               registers are equal, the interrupt flag is set and the
;               timer registers are cleared.
; Arguments:    n - the timer number.
; Precond'ns:
; Postcond'ns:  Counter is decremented if the timer is running.
; Regs Used:    WREG
;****************************************************************
TIMER_ISR macro n
    btfss   TMRF#v(n), 0        ; Is The timer Running??
    bra     _END_#v(n)          ; NO - nothing to do.
    ; Decrement the counter.
    ; Note: after doing decf, STATUS<C> is set unless the register underflowed.
    decf    TMRL#v(n)           ; Decrement the counter.
    btfss   STATUS, C           ; Did the register underflow? (0x00 -> 0xFF)
    decf    TMRH#v(n)           ; YES - decrement the high byte.
    ; Test if the counter is zero.
    tstfsz  TMRL#v(n)           ; Is the low byte zero?
    bra     _END_#v(n)          ; NO - timer not expired yet.
    tstfsz  TMRH#v(n)           ; Is the high byte zero?
    bra     _END_#v(n)          ; NO - timer not expired yet.
    ; Clear the timer flag to say the timer has stopped.
    clrf    TMRF#v(n)
;   bra     _END_#v(n)
_END_#v(n):

 endm

;****************************************************************
; SUBROUTINE:   SWTIMERS_ISR
;
; Description:  Runs the ISR's for all the timers.
```

```
SWTIMERS_ISR:

    ; Check if TIMER2 caused the interrupt.
    SERVICE_IRP     PIR1, TMR2IF, PIE1, TMR2IE

    TIMER_ISR 4                 ; Run the TIMER4 ISR.
    TIMER_ISR 5                 ; Run the TIMER5 ISR.
    return

 end
```