
Design and Implementation of PicoRadio Data Link Layer

By

Jie Zhou

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Professor Jan M. Rabaey
Research Advisor

Date

* * * * *

Professor Robert W. Brodersen
Second Reader

Date

Table of Contents

Chapter 1 - Introduction	7
1.1 - PicoRadios for Wireless Sensor Networks.....	7
1.2 - Motivations.....	8
1.3 - Overview of the PicoRadio Project.....	9
1.4 - PicoRadio Protocol Stack.....	12
1.4.1 - PicoRadio Application Layer.....	12
1.4.2 - PicoRadio Network Layer.....	12
1.4.3 - PicoRadio Data Link Layer.....	14
1.5 - Related Work.....	17
1.6 - Project Objective.....	18
1.7 - Thesis Overview.....	18
Chapter 2 - History	20
2.1 - Data Link Layer Design Overview	20
2.2 - Data Link Layer System Design Methodology.....	21
2.2.1 - Initialization Process	21
2.2.2 - Maintenance Process.....	22
2.2.3 - Process of Transmit and Receive Network Data	25
2.2.4 - Broadcast and Unicast Session Transmissions	28
2.2.5 - Search for Neighbors	29
2.3 - Comparative Study.....	30
2.4 - Summary.....	31

Chapter 3 - Implementation Tool Set.....	32
3.1 - VCC Feature Introduction and Design Flow.....	32
3.2 - Matlab Stateflow.....	34
3.3 - Matlab Simulink.....	34
3.4 - Xilinx Block Sets in Simulink.....	35
3.5 - BEE/SSHAFT Design Flow.....	35
3.6 - SF2VHD.....	37
3.7 - Summary.....	38
Chapter 4 - System Implementation.....	40
4.1 - Data Link Layer System Architecture.....	40
4.2 - Initialization Manager (InitMngr) Block.....	41
4.3 - Local Address Manager (AddrMngr) Block	45
4.4 - Information Processor (InfoProc) Block	46
4.5 - FIFO Control Block.....	51
4.6 - Transmit Datapath (TXD) Block.....	52
4.7 - Receive Datapath (RXD) Block.....	54
4.8 - Media Access Control (MAC) Block.....	56
4.9 - Summary	61
Chapter 5 - Analysis of Results	63
5.1 - Simulink/Xilinx Simulations.....	63
5.2 - Functional Simulations.....	65
5.2.1 - Single Node Self Loop Test.....	65
5.2.2 - Two Nodes Transmit and Receive Test.....	66

5.3 - Stateflow Simulations.....	69
5.4 - Generating VHDL Files.....	70
5.5 - Summary.....	73
Chapter 6 - Conclusions	73
6.1 - Using Stateflow.....	74
6.2 - Using SSHAFT/BEE.....	75
6.3 - Pros and Cons of Using VCC.....	75
Chapter 7 - Glossary & References	77
Appendix - Block Diagrams of Data Link Layer in Stateflow and Simulink.....	81

List of Figures

Figure 1.1 - Diagram of a Sensor Node.....	9
Figure 1.2 - Conceptual Architecture of a PicoNode.....	10
Figure 1.3 - Various Boards of a PicoNode.....	11
Figure 1.4 - Energy Efficient Multi Hop Network Scenario.....	13
Figure 1.5 - Top Level Diagram of the Location of DLL in PicoRadio Chip.....	17
Figure 2.1 - Sequence Diagram of the Initialization Process.....	23
Figure 2.2 - Sequence Diagram of the Maintenance Process.....	25
Figure 2.3 - Sequence Diagram of the Broadcast and Unicast Session Process.....	28
Figure 3.1 - VCC Design Flow Chart.....	34
Figure 3.2 - BEE Implementation Design Flow.....	37
Figure 4.1 - Top Level Diagram of Data Link Layer in Matlab Simulink.....	40
Figure 4.2 - InitMngr Input and Output Ports.....	42
Figure 4.3 - Diagram of Timers	45
Figure 4.4 - Address Manager Input and Output Ports.....	46
Figure 4.5 - Information Processor Input and Output Ports.....	47
Figure 4.6 - First_In_First_Out Control Block Input and Output Ports.....	52
Figure 4.7 - Transmit Data Path Block Input and Output Ports.....	53
Figure 4.8 - Receive Data Path Block Input and Output Ports.....	55
Figure 4.9 - Media Access Control Block Input and Output Ports.....	57
Figure 5.1 - Top Level Diagram of Simulating the Receive Data Path.....	64
Figure 5.2 - Two Nodes Scenario Diagram	66
Figure 5.3 - Functional Simulation Setup Diagram.....	68
Figure 5.4 - Matlab Results of Two Nodes Scenario Simulation.....	69
Figure 5.5 - MAC Implementation in Stateflow.....	70
Figure 5.6 - Algebraic Timing Loop	72

Acknowledgments

I would like to thank my mother Dr. Jingyi Song and father, Dr. Yulong Zhou for supporting me in every stage of my life with their altruistic love, patience, inspiration, advice, and time. My mother has always been a role model for me, her hardworking ethic and perpetual search for self improvement will continue to propel me to achieve. My father's encouragement was vital in my quest for knowledge. Finally, I would like to mention my grandfather, Professor Jing Ying Song, founder and chair of the mechanical engineering department of Tsinghua University for inspiring me to become who I am.

Sincere gratitude to all my professors, both at UC Berkeley and at my alma mater, Johns Hopkins University for enriching my intellectual experience. In particular, Professor Andreas G. Andreou, who has inspired me to further my graduate research; Professor Borivoje Nikolic for his advice on coursework and enthusiasm in helping me with job search; Professor Andrew R Neureuther for his kindness and understanding during my teaching assistantship; Professor Bernhard Boser for patiently answering all my technical questions and Professor Robert G. Meyer for his teachings in IC for communications and assistance in my employment search. Also, thank Ruth Gjerde, and all members of the graduate student office, who were of invaluable help to me from the first time I entered Cory Hall on the EECS visit day to the last days of filing my thesis.

My time at Berkeley would have been a desolate one if not for my peers who have contributed to my intellectual development and enriched my social life, particularly Hanching Fuh, Richard Lu, Nathan Chan, Eddie Ng and Jon Choy, memories of doing homework, projects, prelim reviews, weekly lunches and BWRC retreat rides will always

remain with me. Special thanks goes to James A. Cataldo for his patience and inspirations. Sincere graditudes to Mounir Bohsali, Mark Chew, Radu Zlatanovici, Bill Tsang, and Ruth Wang for putting up with my endless questions. Lastly, I would like to thank all my colleagues and friends at Berkeley.

Berkeley Wireless Research Center has been a great place to learn and work thanks to its highly motivated staff and students. I am indebted to Kevin Zimmerman for his friendly computer assistance; Elise Mills for her generous help in administrative matters; Fred Burghardt for his expertise in DLL interface design. Lastly, I would like to thank all the members of the NAMP (Network, Application, MAC, and Positioning) group for investigating the design methodology for low-power wireless protocols that this project utilizes. Senior students including Mei Xu, Tufan Karalar, and Michael Sheets were of crucial help in my research. Special thanks to Dr. Johnathan Reason, for his patience, time and teachings throughout my project. His guidance was critical in the writing of this thesis.

Last but not least, this work would not be made possible without my advisor, Professor Jan M. Rabaey for providing me a great opportunity to work on this project. His advice and guidance has been invaluable throughout my graduate career. I am grateful to Professor Bob Brodersen for being this thesis second reader and for aiding me with the Berkeley emulation engine which facilitates the completion of this project. Funding was provided by NSF's "XYZ on a chip", grant number # CMS-0088648 and "Power Aware Sensing, Tracking, and analysis", PASTA (PICO 2): cooperative agreement # 73518 via USC funded via DARPA under grant # F33615-02-2-4005.

Chapter 1

Introduction

1.1 PicoRadios for Wireless Sensor Networks

Recent technology breakthroughs have enabled designers to assemble wireless networks of heterogeneous nodes sensing and collecting wide ranges of environmental data. Some applications of these sensor and monitoring networks include robot control; guidance in automatic manufacturing environments; environmental control in office buildings; integrated patient monitoring and drug administration in hospitals; smart homes with build-in security, identification, and personalization; and interactive toys and museum exhibitions.

The key to these ubiquitous networks is the creation of small, lightweight, low-cost network constituents, named PicoNode. These nodes will occupy less than 0.15 cubic inches, weigh less than 100 grams, and cost less than fifty cents. One criterion to eliminate frequent battery replacement for these nodes is that they must operate under ultra-low power environment. The final version of these nodes will consume below 100 microwatts of power. This level of low power-dissipation calls for an energy-scavenging scheme in which the self-powered nodes consume energy that is extracted from the environment such as wind, vibration, body heat or solar power. To reach this aggressive power dissipation level, the effective range of each PicoNode is limited to a maximum of few meters. Extending the reachable data range requires a scalable network infrastructure that allows distant nodes to communicate with each other. A self-configuring ad hoc

networking approach is crucial to the deployment of such a network that consists of hundreds of nodes. To reduce the PicoNode's energy dissipation, consideration for energy reduction throughout all layers of the design hierarchy (from system to architecture, circuits, and technology) is required. The largest opportunity lies in the protocol stack where a trade-off between communication and computation, as well as elimination of overhead, can lead to significant energy reduction [1].

1.2 Motivations

An untapped opportunity in the world of wireless data lies in small, lightweight, low data-rate of less than one hertz, low-cost wireless transceivers [1]. Such transceivers give birth to sensor nodes that are widely used in distributed networks. This enables applications such as smart buildings and highways, entertainment and factory automations [2]. This kind of distributed network can process large amounts of data, while the individual nodes only contribute a typical data rate of less than 1 Kbit/sec. The protocol stack should be designed to aggressively power-down idle units to avoid large leakage power. A power management scheme powers up functions automatically in response to events at their interfaces shown in Figure 1.1. These events can be from external sources such as sensors or wake-up requests or internal sources such as neighboring blocks. A centralized supervisor block maintains system state, schedules periodic system maintenance, and ensures that blocks do not power down during communication sessions [2].

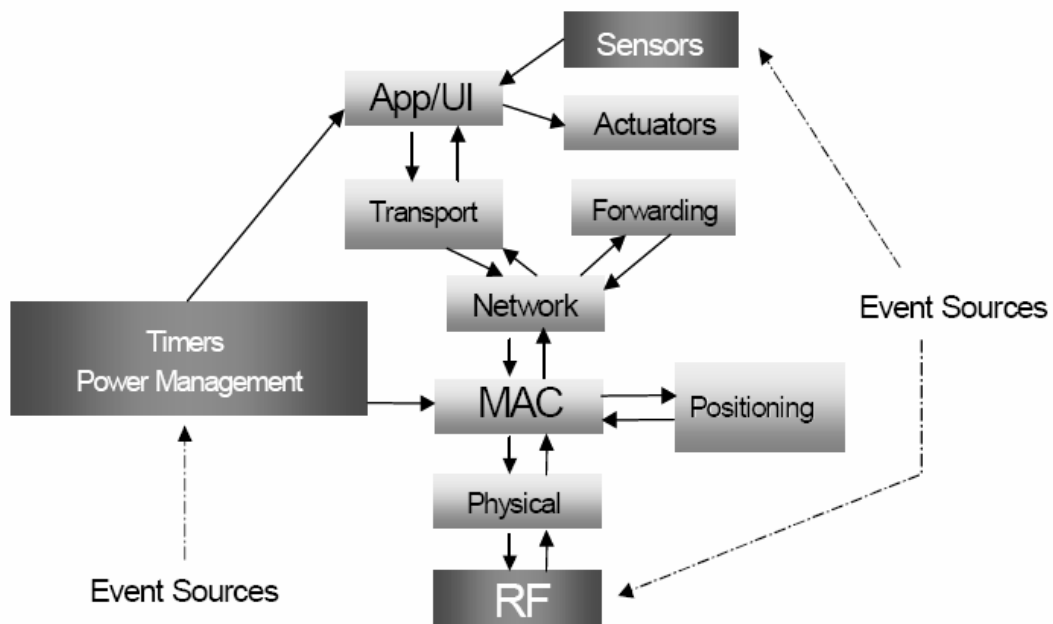


Figure 1.1 Diagram of a Sensor Node

The aim of the PicoRadio project is to create a single chip solution that contains a transceiver, a full protocol stack, IO ports and computing power for sensor and control applications. The PicoRadio network is a multi hop, ad-hoc network. Each of these Pico nodes is consisted of a 2.4GHz Bluetooth radio module, a Xilinx FPGA, and a Strong ARM microprocessor.

A personal motivation is to be able to work with a group of extremely talented, knowledgeable and motivated people in the Berkeley Wireless Research Center. This project also provided me a rare opportunity to work on a research project that is closely tied with industry interests and interests of future wireless consumers.

1.3 Overview of the PicoRadio Project

The goal of the PicoRadio Project is to build an ultra low power single chip transceiver for wireless sensor applications. The chip contains radio modules,

reconfigurable data path, DSP, FPGA, low power reconfigurable state machines and a programmable processor. These are the essential components in a typical sensor that may be used in monitoring applications. Figure 1.2 below shows the architecture of a PicoNode [1].

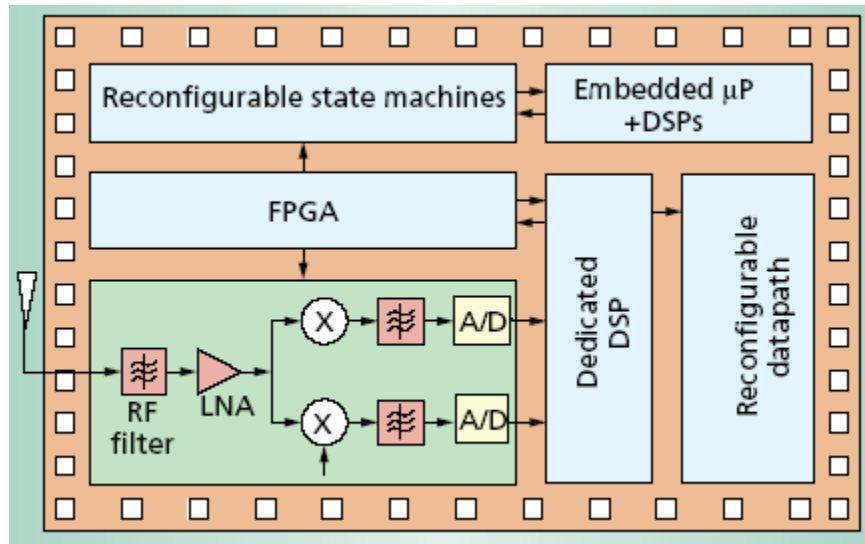


Figure 1.2 Conceptual Architecture of a PicoNode

The PicoRadio Project is a huge project that span over several years to complete. It is divided into three main stages, namely, PicoRadio I, II and III. A higher level of integration is achieved at each stage. Another target is to minimize the power consumption from one stage to the next by a factor of ten.

PicoRadio I, also named the TestBed, consists of PicoNodes. These nodes model the final chip. The models are built using off-the-shelve components. The main purpose of the TestBed is to provide a platform to test the network protocol. The TestBed enables the designers to investigate system-level aspects of a PicoRadio network before it is built. It is a prototyping environment that consists of hardware and algorithms that would later

run on the nodes. Every node has a set of custom circuit boards and software libraries that are used to control the hardware [4]. The boards are small and can be stacked into a custom case. The boards are separated into a digital board and a power board. The digital board contains a Strong ARM 1100 embedded microprocessor and a Xilinx XC4020XLA Field Programmable Gate Array (FPGA) [4]. The power board powers the digital board and it has an auxiliary 5 Volt power supply. Dynamic voltage scaling is utilized. Other than these core boards, a Test Bed node also includes a radio board, a Bluetooth radio board and a sensor board. These sensors could be used to measure temperature, humidity, and light intensity. Also, the sensor board contains a microphone and speaker driver, which are intended to be part of an acoustic anemometer capable of measuring very low levels of air movements indoors [4]. All the boards are shown in the Figure 1.3 [4].

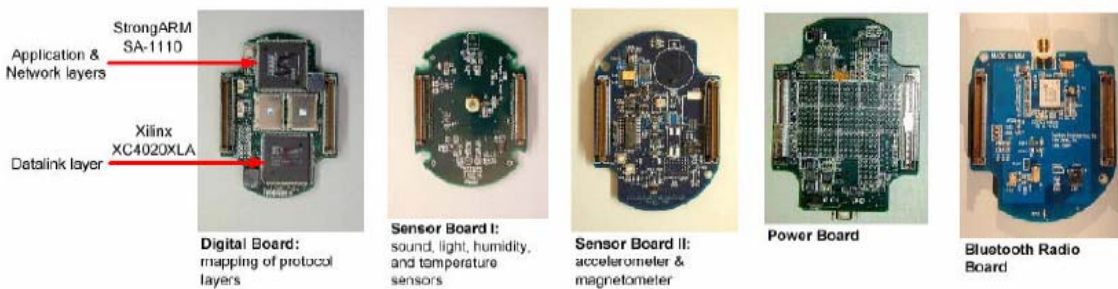


Figure 1.3 Various Boards of a PicoNode

PicoRadio II is built using several customized chips that is used to test the different parts of the final chip. Another purpose is to develop a tool chain for application, protocol, and physical layer implementation and integration.

PicoRadio III is the final stage where the entire integrated system is on a chip. On this chip the different layers will be implemented on flexible low-power computation

fabrics. More information about the different stages of the PicoRadio project can be found at http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/Default.htm.

1.4 PicoRadio Protocol Stack

The PicoRadio is a distributed network that is designed to be robust, power efficient and self-configurable. The nodes in this network are connected in a web structure that creates spatial redundancy which translates to reliability. The protocol stack consists of three layers; the top layer is the Application layer, the middle layer is the Network layer and the bottom layer is the Data Link layer (DLL).

1.4.1 PicoRadio Application Layer

The Application layer implements information delivery services and accessory services that user applications need [13]. The Application layer consists of a sensor board, an optional sensor board, a Monitor software module, and application drivers that provide the interface between the Application and Network layers. Each node is equipped with a sensor board that includes a microphone and temperature, light and humidity sensors. Some nodes have another optional sensor board that allows for more advanced applications such as motion detection using an accelerometer or magnetometer. The Monitor is a small software module that interacts with the real world via user applications that control and monitor the network. The application drivers takes the incoming packets and use it as controls to activate the sensors, the drivers also assemble outgoing sensor measurements and monitor requests into packets [3].

1.4.2 PicoRadio Network Layer

A typical Network layer implements the end-to-end delivery of packets [13]. A packet is a group of bits that are transmitted at precise times. Information is transmitted in the form of packets. This version of the Network layer utilizes the multi hop networking concept. This concept is best illustrated in Figure 1.4. The energy used in forwarding a packet from a source node to a destination node will be reduced if the packet is forwarded via other intermediate nodes compared to transmitting the packet directly from the source node to the destination node. The latter method is often called one-hop networking.

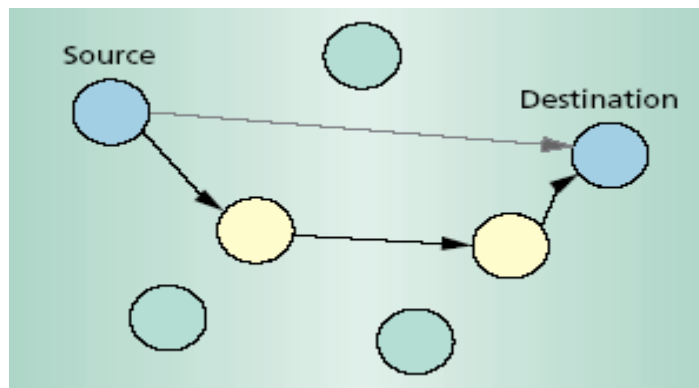


Figure 1.4 Energy Efficient Multi Hop Network Scenario

The Network layer consists of an Energy Aware Routing Algorithm, a Locationing Engine, a Neighbor List Service (NLS) and a Queuing Service. The Locationing Engine is comprised of the algorithm and protocol that allows each node to dynamically discover its position relative to anchor. In the PicoRadio network, a nodes location is analogous to the concept of network address found in most networks. Thus, the Locationing Engine is considered a sub layer of the Network layer because it provides the means by which a PicoNode dynamically configures its network address [3]. Every PicoNode has a network address composed of three numbers that represent its location in

the three dimensional space. The network address is computed and maintained by the Locationing Engine. The Queuing Service provides the interface between the Network and Data Link layers.

The main function of the NLS is to perform address resolution for the Network layer. NLS maintains a table that contains a mapping between its one-hop neighbors' media-access IDs and network addresses. Also, each table entry also includes routing information such as a link quality metric and status indicator. These two functionalities are not implemented in this version of the Network layer. In a typical query, the routing algorithm may provide the NLS with a location and receive the triplet (Status, MAC ID, and Link Metric). Another function of the NLS is to manage the timing of events during the initialization process, in which a node discovers its neighbors, computes its location, configures its MAC ID, and joins the neighbors [3].

1.4.3 PicoRadio Data Link Layer

A conventional data link layer implements a packet transmission in two main steps [13]. The first step is to frame the packets into a specific format at the sending node and extracts the packets at the receiving node. The second step is to supervise the resulting packet delivery. The PicoRadio DLL mainly coordinate the access to the medium, because many radios have to share the same interconnect medium, so the messages can interfere with each other. A typical DLL is composed mainly of three parts: the Transmit Datapath (TXD), the Receive Datapath (RXD), and the Media Access Controller (MAC). This particular version of the DLL also contains the Neighbor List Wrapper; the Queue Interface and the Interface to the Power Management Supervisor (PIF). The TXD and RXD each contains a datapath control block that is implemented in

the Stateflow diagrams. The two datapaths also each contains all the typical datapath components, such as transmit/receive buffers, serializer/de-serializer, cyclic redundancy code (CRC), line balancer and word clock. Some main features of the MAC includes CSMA, Cycled receiver, Broadcast and Unicast data transmissions, (which will be revisited in section 2.3.3, 2.3.4, 4.3 and 4.5) and Two-channel configuration.

The cycled-receiver concept is inspired by paging systems. It is often seen in many MAC designs (e.g., 802.11 sleep mode) as a way to decrease the receiver's idling power consumption. This is achieved by turning the transceiver's idle mode into a low-power sleep mode by periodically duty cycling the receiver, as opposed to leaving the receiver on 100% of the time.

A location based broadcast forwarding mechanism is used to address a sensor node in an unknown topology. The PicoRadio Network has one unique broadcast channel that all nodes listen to. A broadcast transmission is sent once and it is not guaranteed reach its destination since there is no interaction between the nodes and collision or lost of packets may occur. If node A asks node B for data, node A sends out a broadcast request. The broadcast request is forwarded by intermediate nodes to node B. The broadcast forwarding method uses the location addresses of the different nodes to accomplish multi hope relay of packets.

To ensure the transmission of a packet from one node to a neighboring node, unicast transmissions method is utilized. Unicast transmission is session based. A session is initiated by a session setup on a common broadcast channel; the data transmission itself is then on a unique unicast channel. Both the session setup and the data transmission must be acknowledged by the receiving node on the unicast channel.

In the real world, packets may be lost at times. For every packet there is a specified resend interval after which the packet is resent if a response is not received. The exceptions are the 'Data to Transmit' (DTX) and the 'End of Session' (EOS) type transmissions. The DTX re-transmission is based on a negative acknowledgement. That means the DTX is retransmitted if another CTS is received. The EOS is the last packet of the session so it has no acknowledgement. EOS is sent three times to increase the probability of a successful transmission. The resend interval structure could cause an infinite retransmission if the link between two nodes is broken. It could also force unstable network behavior if too many nodes are in the retransmission state. To avoid this undesirable situation, a session timeout was introduced. This timeout ends the session after a given maximal time. In this way, a failure in the EOS transmission is no longer a problem because the session will end when time runs out. If all three EOS transmissions fail, a DTX timeout ends the session after a certain time. The DTX timeout has the same duration as the RTS resend timeout.

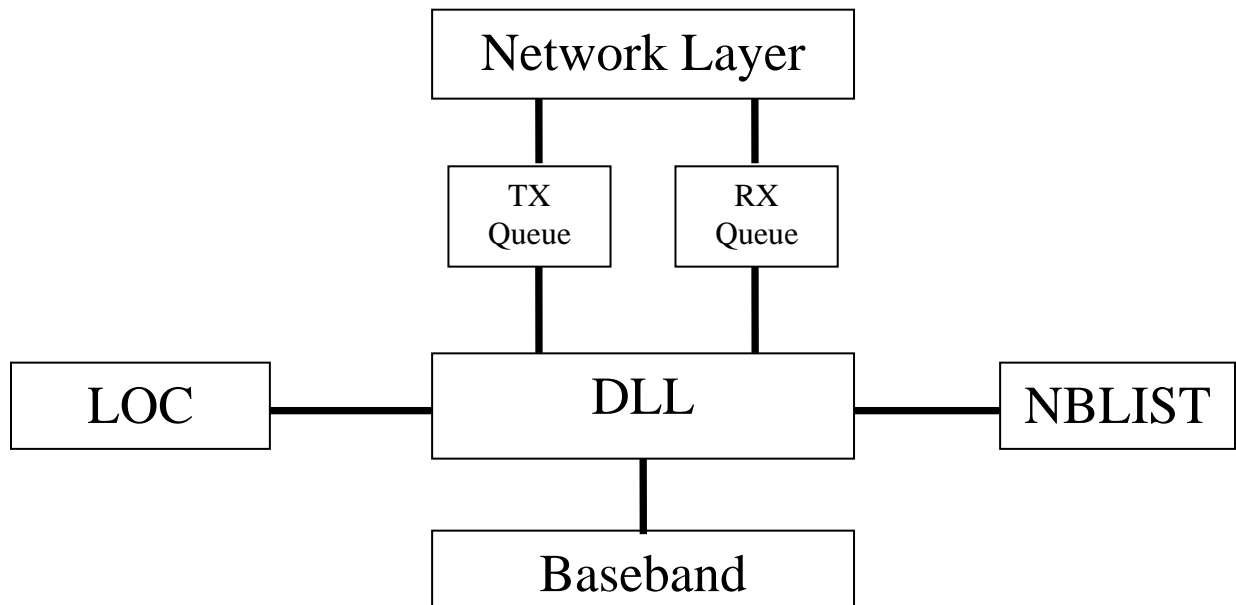


Figure 1.5 Top Level Diagram of the Location of DLL in the PicoRadio Chip

A simple top level diagram showing where DLL fit into the overall PicoRadio chip is shown in Figure 1.5. In this Figure, NBLIST is the neighbor list table that stores all the neighbors' IDs and network addresses for a node, LOC is the locationing engine that assigns the node a local address if no conflict has been received during the initialization process. TXQ is the transmit queue and the RXQ is the receive queue. These queues are memory buffers that store the data until it is time for it to be transmitted to the network layer. DLL sits in the middle since it communicates with all the blocks shown in the Figure.

1.5 Related Work

Considerable amount of work has been done in the past on the design of the Data Link Layer for PicoRadio Networks in VCC [9]. Unlike the current DLL described in this report, the old version of the DLL maintains a Neighbor List Table that provides

neighbor list services such as address resolution for the Network layer. The Neighbor List Table contains a mapping between its one-hop neighbors' media-access (MAC) IDs and network addresses. The entries in the neighbor list table do not contain routing information such as link quality metric and status indicator. Another main difference between the two versions of the DLL is that the old version of the DLL described in [9] does not support the power management scheme. A detailed comparison between the old and current version of the DLL will be mentioned in section 2.4.

1.6 Project Objective

The objective of this project is to implement the data link layer protocol, the neighbor list wrapper, the power management supervisor interface and the transmitting and receiving queue interfaces for the PicoRadio chip. Matlab Stateflow was used to capture and simulate the network behavior in order to verify and improve the protocol. Graphical design of the actual transmit and the receive datapath were done in Simulink/Xilinx environment. BEE (Berkeley Emulation Engine) was used to build the whole DLL design onto FPGA. The entire design was ported to VHDL and tools such as Synopsys Compiler and INSECTA were used on the ASIC side to put DLL on the final PicoRadio chip.

1.7 Thesis Overview

This thesis report will describe the process of design and verification of the data link layer including the various interfaces to the other protocol layers. This report will also comment on the tools used for implementation. It is divided into seven chapters.

- Chapter 1 describes the whole view of the PicoRadio Networks and how does the Data Link Layer fit in the picture. It will also describe briefly the interactions between the Data Link Layer and the other protocol layers.
- Chapter 2 discusses the design methodology for the DLL and the architecture of each DLL sub blocks. The chapter ends with a comparative study of related and previous work done for the data link layer.
- Chapter 3 describes the implementation tool environment, namely VCC, Matlab Simulink and Stateflow, INSECTA, SF2VHD and BEE design flow.
- Chapter 4 presents detailed implementation documentation for every block in the data link layer.
- Chapter 5 is a summary of functional simulation results of this project.
- Chapter 6 is the conclusion and
- Chapter 7 lists the references and diagrams of all the data link layer blocks.

Chapter 2

History

2.1 Data Link Layer Design Overview

Each PicoNode contains a complete protocol stack that is described in section 1.4. The PicoRadio data link layer of this stack does everything that a typical data link layer does plus some extra features to service the neighbor-list. Its main functionality includes initialization and maintenance management, such as assigning local address, discovering and keeping one-hop network topology, and processing control broadcast messages and datapath between the network layer and physical layer for broadcast and unicast messages with FIFO buffers. MAC also uses CSMA and beaconing mechanisms. It discovers neighbors in the initialization phase, maintains a neighbor lookup table, forwards the neighbor information to the network layer, keeps a list of two-hop neighbors' addresses, assigns its own local address at random, computes its own location, maintains on a periodical basis, issues random back-off and designates channel for transmission, etc. In addition, it establishes reliable data transmission between neighboring nodes [4].

It has the following features:

- ❑ Reactive Behavior: All stateflow states are turned off when there is no event coming. All memory lost except the node only remembers to go back to the idle state once it wakes up.
- ❑ Carrier sense multiple access protocol (CSMA) is used to lower the probability of collision, since the level of collision is based upon the traffic flow of the network of nodes and it could be high at certain times.
- ❑ The beaconing mechanism is used to set up a unicast session.
- ❑ A locally unique address is assigned to each node.
- ❑ The physical layer provides 2 radio channels; one is used for broadcast and the other is for unicast. Since there are only two channels, this version of DLL cannot guarantee that no two neighbors will use the same channel simultaneously.
- ❑ The Matlab Simulink and Stateflow are used to graphically capture the functionalities of the data link layer.

2.2 Data Link Layer System Design Methodology

The data link layer consists of five functional system-level procedures. The five procedures include the procedure for initialization; procedure for maintenance; procedure for transmitting and receiving network data; procedure for transmitting broadcast and unicast data and procedure for searching neighbors. Each of these procedures requires the use of one or more sub-blocks to accomplish the tasks. Detailed descriptions are listed in the subsequent sections.

2.2.1 Initialization Process

A PicoNode is designed to initialize itself after it is powered up. During the initialization process, the data link layer of the node discovers its neighbors, updates the neighbor list, and assigns a locally unique address to itself. The main nine steps that a node goes through during the initialization stage are summarized below. A sequence diagram showing the flow of events is displayed in Figure 2.1

1. After a node is powered up, it broadcasts an initialization request to all its neighbors.
2. After receiving the request, each neighbor broadcasts a response with its local address, its location that is represented by 3 numbers and a list of its neighbors.
3. The initializing node updates its neighbor list and available address space according to the responses it gets from the neighboring node.
4. After a certain time, the node assigns itself a local address from the available address space.
5. The node broadcasts an information message containing its new local address, location and neighbor list, and waits for conflict notifications.
6. The neighbors update their neighbor lists based on the information message.
7. If the conflict notification has not been received by the node after a limited amount of time set by a local timer, the node will go to steady state.
8. If there is a conflict message received, the node will reassign its local address and go back to step 5.

9. If any messages are lost during the initialization process, errors will arise. The node can correct these local topology errors by updating and maintaining its neighbor list periodically.

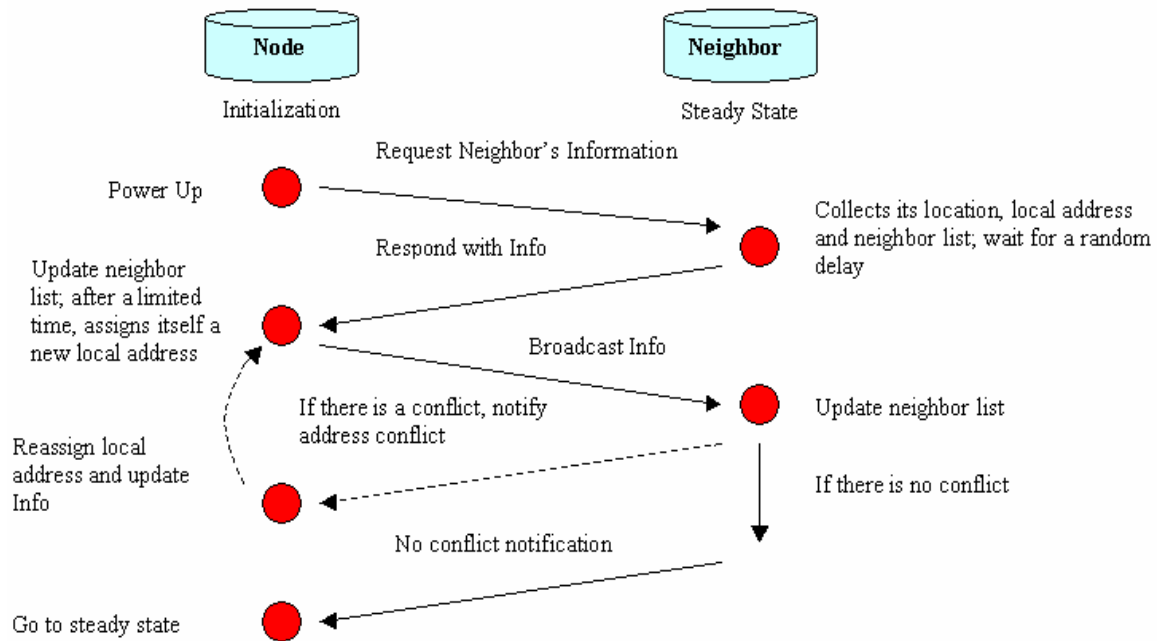


Figure 2.1 Sequence Diagram of the Initialization Process

2.2.2 Maintenance Process

There are two topology maintenance schemes, namely proactive and reactive. For the proactive scheme, a node re-discovers all its neighbors during the maintenance process. For the reactive scheme, a node removes its neighbors that have been inactive for a period of time without notifying them. Each of these schemes has its benefits and drawbacks. The proactive scheme captures the network topology more precisely than the other, but it consumes more energy. The PicoRadio team decided to employ a combination of these two schemes in order to track the topology change accurately and save power.

The maintenance process is self-contained within each node. It is performed based on the link quality of the nodes. A node can only perform a link test with another node that it aware. If a new node comes into the neighborhood, a broadcast packet will inform the node of its new neighbor. The new neighbor will be added to the node's neighbor list during the initialization process.

The frequency of performing a link test is parameterized, so users can control it. In the Test Bed implementation, it is performed very infrequently, e.g. once every 3 to 4 minutes. Data transmission is also used to determine the quality of the link. Therefore if the timeout for the link test is greater than the inter-packet timeout for data transmission, then no link test will be performed.

A detailed flow of events is illustrated in the sequence diagram of the maintenance process in Figure 2.2 and the major steps describing this flow of events are listed below.

1. There is a link test timer during the maintenance process. When the timer expires, the node starts to perform a link test with all its known neighbors within a specified neighborhood radius.
2. If a neighbor is active, then the link test is successful. The node will check its neighbor list and update it. It will then set the link test timer again.
3. If a neighbor is inactive, then the link test fails. The node will remove that neighbor from its own neighbor list.

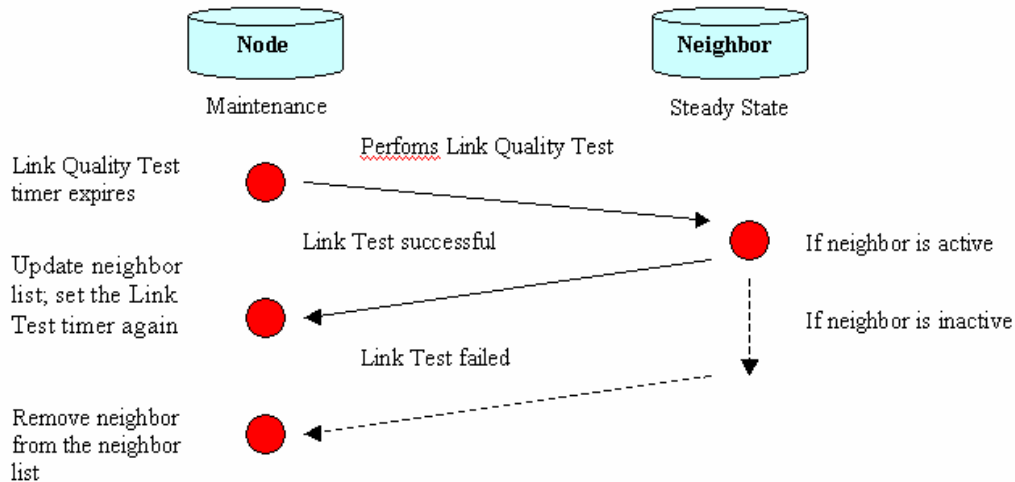


Figure 2.2 Sequence Diagram of the Maintenance Process

2.2.3 Process of Transmit and Receive Network Data

There are only two channels in the PicoRadio III implementation, the broadcast channel and the unicast channel. Neither channel has priority over the other. There are two types of timeouts. The session timeout determines when to close a session and the acknowledgement timeout specifies the waiting time for an acknowledgement.

If a node wants to send data to a neighboring node, it sends a 'session setup request' message on the broadcast channel along with the address of the specified receiver. After sending the session setup request, the sender waits for an acknowledgement in the form of 'ready to receive' message. If there is no acknowledgement after a certain time set by a local timer, the sender will resend the session setup request. If the sender receives an acknowledgement, sender will then send the data on the unicast channel. After sending the data, the sender waits again for an acknowledgement. If there is no acknowledgement, the sender will assume that the receiver node has received the data and it will go to idle state. If there is an

acknowledgement, then the sender assumes that the receiver has not received the data, so the sender resends the data to the receiver. This is called negative acknowledgement timeout method. This method is only used by the sender, not the receiver.

On the receive side, the receiver gets a session setup request message, and then the receiver will periodically send out the 'ready to receive' message on the unicast channel. After this point, the receiver either gets the data after a certain time or the number of tries expires and the receiver close the session on its own without telling the sender. So the sender may be still trying to send the data to the receiver after the receiver has closed the session at its end. To prevent the sender from wasting too much time trying to send the data after the receiver has closed the session, timeout are employed for sending the data on the sender's side and for sending the 'ready to receive' message on the receiver side to be about the same length, so that when the receiver has closed the session on its own, the sender wouldn't waste too much time still trying to send the data to the receiver.

The receiver sends out the 'End of Session' (EOS) message to the sender only after it has received the data. The sender can go to idle state either when it receives the EOS message or the timer for waiting for the 'ready to receive' message has expired. This process is illustrated in Figure 2.3 and a list of the main steps is given below.

1. If there is data to be sent, the sender node attaches a data link layer header to the packets that contains these data.
2. The sender initiates a session by broadcasting a 'session setup request' message and then wait for an acknowledgement. This request message contains the address

of the receiver node. This receiver must already be on the sender's neighbor list; otherwise its location address will be unknown to the sender.

3. The receiver matching the specified location address contained in the session setup request message sends out a 'ready to receive' acknowledgement on the unicast channel if it is ready to receive data, and then tunes to the unicast channel to listen for the data and sets a waiting timer for receiving the data.
4. If the sender gets the 'ready to receive' acknowledgement in time, it sends the data on the unicast channel, and then tunes to unicast channel to listen to an acknowledgement from the receiver and sets a waiting timer for receiving that acknowledgement.
5. If the receiver receives the data in time, it sends back an 'end of session' acknowledgement on the unicast channel and goes to idle state. If this received packet is error free, the payload will be forwarded to the network layer and the status of the sender in the neighbor list will be set to active.
6. If the sender gets the "end session" in time, it goes to idle state.
7. If the receiver has not received any data before its timer expires, the receiver resends the 'ready to receive' acknowledgement to the sender. If the sender receives this acknowledgement in time, it will send the data and go through step 4, 5 and 6.
8. If the receiver has not received any data after its timer expires, the receiver closes the session and go to idle state. No more acknowledgements will be sent to the sender after the session is closed.

9. If the sender has not received any acknowledgement and the acknowledge timer expires, the sender will go to idle state as well.
10. If any of the above waiting timers expires before data or acknowledgement message is received, the retransmission mechanism will start working until reaching the retry limit.

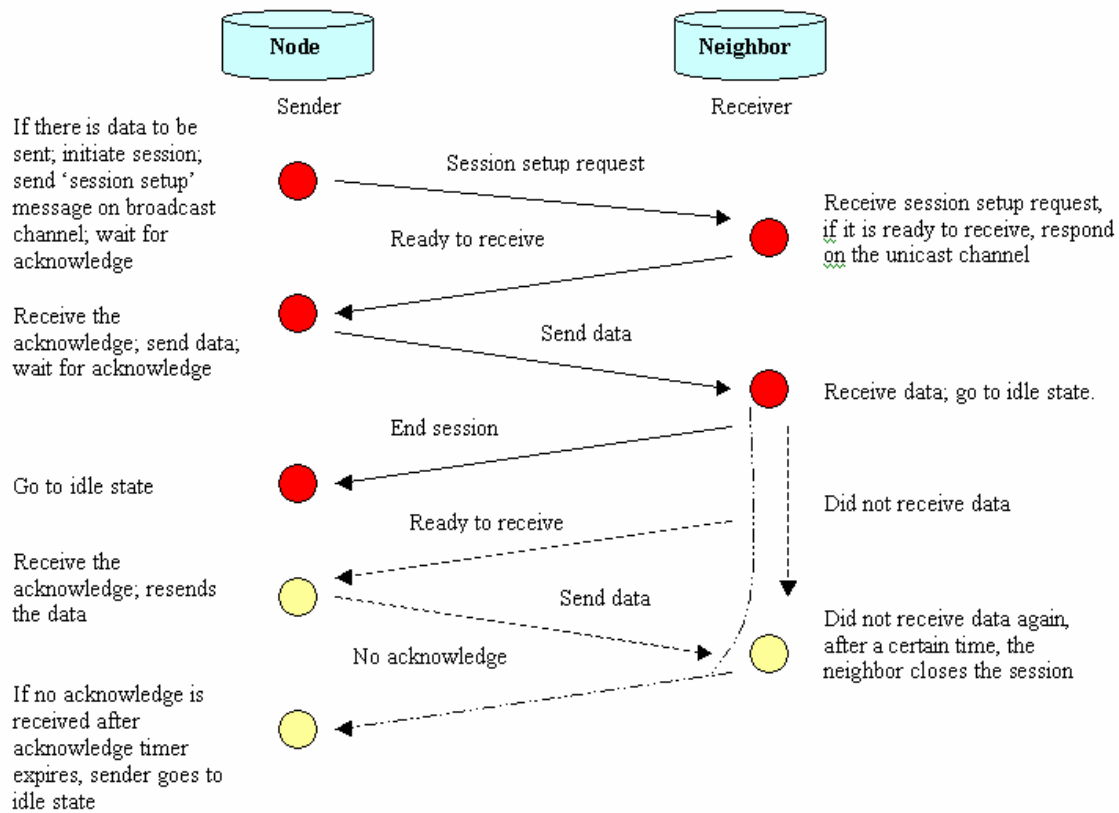


Figure 2.3 Sequence Diagram of the Broadcast and Unicast Session Process

2.2.4 Broadcast and Unicast Session Transmission

The previous section described in detail the use of broadcast and unicast channels during transmitting and receiving data from one node to the other. Broadcast transmission is an efficient and simple way of communication since all nodes use the same channel and there is no handshaking mechanism between nodes. Another advantage of the

broadcast transmission is that the node does not need to know the routing path to send a packet to a destination node. A disadvantage for all nodes to transmit on the same channel is that collisions may occur and packets may be lost as mentioned in section 1.4.3.

To improve the successful rate of a data transmission, unicast session transmission is introduced. Data acquisition power is less in a unicast transmission than a broadcast transmission because the nodes other than the two specific nodes in contact are not affected by a unicast session. Also power consumption is minimized in a unicast session transmission since it is not likely to be interrupted by other network traffic, thus, fewer packets will have to be re-sent.

2.2.5 Search for Neighbors

The Network layer needs to match neighbor's ID and location address information for routing purposes. The Network layer looks for a node in the neighborhood that is closer to the destination node by comparing all the neighbors' ID. In the reactive scheme, the network layer picks out a set of IDs that are close to the destination node and send it to the DLL. The DLL chooses the best neighbor ID out of this bunch of IDs based on how close this chosen ID is to the actual specified destination node.

Unlike the DLL, the network layer looks for a neighbor by searching the destination cache, not the neighbor list table. The destination cache contains possible routes to the destination from several nodes in its neighborhood that are close to the destination. For example, node 1, 2, 3 and 4 are in a sender node's neighborhood, but only node 1 and 2 can reach the destination node. The destination cache contains the

routing information for node 1 and 2 to reach the destination node. There are three scenarios of how the network layer searches for a neighbor:

1. If the network layer sends out a request specifying a location, then the local addresses of neighbors in that location will be returned.
2. If the network layer gives a request specifying a local address, then the location of a neighbor pertaining to that address will be returned.
3. If any of the search returns empty, an error message will return to the network layer.

2.3 Comparative Study

There are a number of significant differences between the previous version of the DLL and the current one. Readers not familiar with the old version of the DLL should visit reference [13] before reading this section. The most obvious change to the DLL is that the current DLL no longer contains a neighbor list table that stores all the IDs and location of a node and its neighbors. This change is made because the BEE design flow can not process arrays or tables and thus the design can not be implemented on the FPGAs. Since the neighbor list table has to be outside of the DLL, a neighbor list wrapper was build to accommodate this change. Also the current version of the DLL is has build-in interface mechanisms so that it can directly communicate to all the other protocol layers including the RF physical layer. The basic functionality of each DLL sub block is preserved, however many basic concept of the protocol implementation has been modified. Some of these major changes are listed below.

- ❑ The idea of a removal-warning message has been abandoned because it is too inefficient to implement. It is replaced by a link test method so each node is self-contained.
- ❑ The 'return data' message type is no longer used in transmit and receive protocols.
- ❑ The concept of destination cache is introduced.
- ❑ The process of handling conflict is entirely changed since 'select location' or removal-warning messages are no longer in use.
- ❑ Neighbor list table physically resides outside of the DLL, because the BEE design flow cannot handle arrays or tables.
- ❑ There is no longer a need for aggregation of packets.
- ❑ Broadcast data no longer has a higher priority over the unicast data. Both types of data have the same forwarding delay and they are both time sensitive.

2.4 Summary

C. Zhong and members of the NAMP group proposed the earliest version of the PicoRadio data link layer protocol, which was used as a guideline to implement the current DLL. Since then, many adjustments have been made to the protocol during the implementation process after careful simulation and examination. The Matlab Simulink and stateflow is a powerful tool in capturing the functionality of the system level processes and the DLL sub-system blocks. Further discussion on the implementation tools will be mentioned in the next chapter.

Implementation Tool Set

3.1 VCC Feature Introduction and Design Flow

The previous version of the DLL was implemented with the Cierto™ Virtual Component Co-Design (VCC) tool. VCC is a development tool set introduced by Cadence. It integrates intellectual property models to simulate, evaluate, and select appropriate virtual components for digital communication, automotive, and multimedia system design [5]. VCC was used to capture the design behavior, run functional simulations in order to debug and verify the design, and to explore different target architectures. Some main features of the VCC are mentioned here:

- VCC is a platform-based tool derived from the design methodology of orthogonalization of interests, such as separation of function and architecture, computation and communication.
- VCC is able to select different system target architectures and hardware/software partitions for the same design and then compare their performance, this is called architectural exploration.
- The underlying model of computation is Co-design Finite State Machine (CFSM). This is an extended finite state machine. CFSM is a control-dominated model that is suited for modeling simple protocols. CFSM include globally asynchronous behavior, which models the fact that communication and computation takes time.
- There are several flexible ways to capture behavior:

Clear box State Transition Diagram (STD): draws CFSM models graphically.

Clear box SDL: writes textual CFSM models.

White box C

Black box C++

The basic VCC design flow is depicted in Figure 3.1 [9] and there are five steps in the design flow:

1. Capture the desired behavior.
2. Run a functional simulation for verification.
3. Choose and compose a target architecture using hardware and software components.
4. Map the behavior diagram onto the architecture.
5. Run a performance simulation for evaluation.

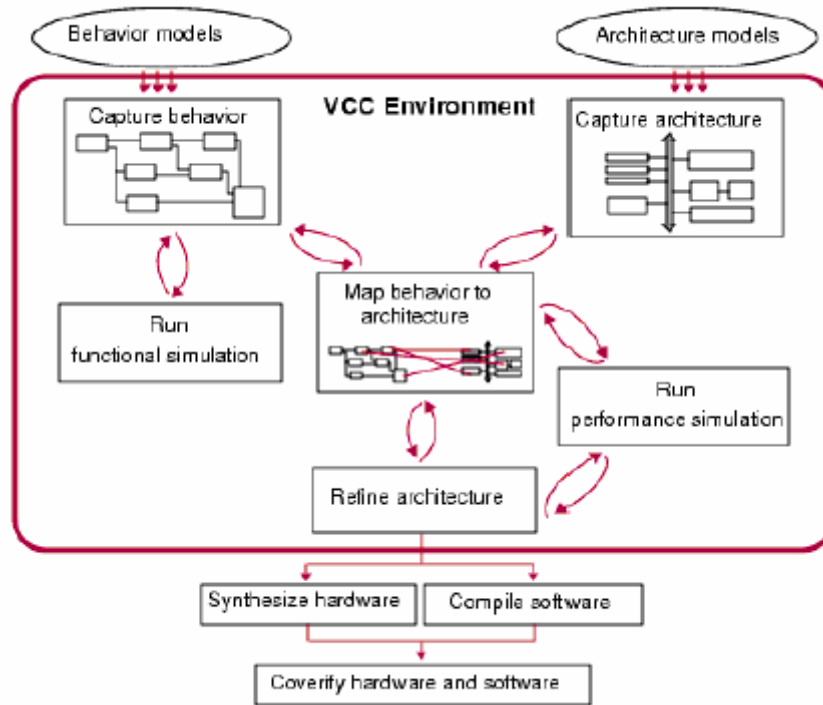


Figure 3.1 VCC Design Flow Chart

3.2 Matlab Stateflow

Stateflow is a design tool for modeling and simulating event-driven systems. It works well with Simulink and MATLAB, Stateflow is an elegant solution for designing embedded systems that contain supervisory logic. The combination of graphical modeling and animated simulation process improves the design procedures.

Stateflow is also a rich graphical and textual language that is mainly based on the original State Charts language defined by Harel [5]. It uses a purely software-based execution model and provides sophisticated features such as state hierarchy, arrays, parallel processes, event triggers, and multi-decision transition conditions.

3.3 Matlab Simulink

Simulink is an interactive tool for modeling, simulating, and analyzing dynamic, multi-domain systems. It allows the user to build a block diagram, simulate the system's behavior and evaluate its performance in order to refine the design. Simulink works with MATLAB, to provide the user with immediate access to a wide range of analysis and design tools. These benefits make Simulink the tool of choice for the DLL system design and simulation purposes.

3.4 Xilinx Block Sets in Simulink

The DLL was initially described only using Stateflow because Matlab version 6.1 did not include Simulink/Xilinx block sets. Shortly after MathWorks partnered with Xilinx, a new tool called the Xilinx System Generator for Simulink was released. As soon as Matlab version 6.5 became available, the real datapath in the DLL was implemented with the new Xilinx block sets. The Xilinx block set includes pre-designed DSP cores, LogiCOREs, to aid in implementation of the DLL design on Xilinx FPGAs. The Xilinx CORE Generator and LogiCORE modules significantly shorten the FPGA hardware design phase.

An advantage of using Xilinx System Generator with the MathWorks Simulink is that the design time can be shortened by quickly iterating between a system representation and a hardware representation of the designs. This is especially important for this DLL design, since many system-level design tradeoffs are based upon the results of the hardware implementation. In addition to shorter design time, the new System Generator allows designers who are not familiar with FPGAs to use MathWorks and Xilinx tools to adopt FPGA technology for DSP applications [10].

3.5 BEE/SSHAFT Design Flow

The DLL is graphically captured using Stateflow and Simulink/Xilinx block sets. The controller of events is implemented in Stateflow and the actual datapaths are implemented with Xilinx block sets. In order to translate the entire design to ASIC, SF2VHD is used to convert Stateflow diagrams into VHDL, which is not yet compatible with ASIC requirements. The System generator automatically generate VHDL file for the Xilinx block sets. The two VHDL files will then be combined and fed into INSECTA. INSECTA is a wrapper, which includes the Synopsys compiler, design and logic compiler and backend place and route. The resulting VHDL file is “bug fixed” so that it can be used directly for ASIC design. Also at this stage of the tool flow, the generated functional VHDL can be verified and tested to see if it matches the behavior of the system that was described in Simulink. The tool flow is similar for porting the DLL onto FPGA, except after the VHDL combination step, BEE_ISE compiler is used instead of INSECTA so that the VHDL code is suitable for FPGA design. The simple overall BEE design flow is shown in Figure 3.2 [8].

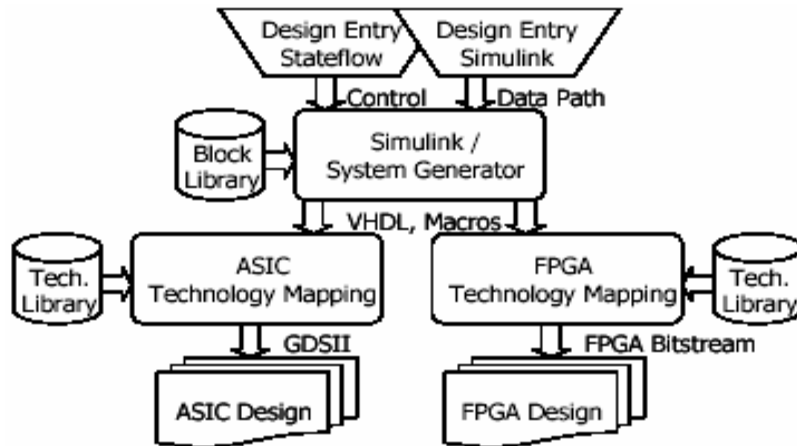


Figure 3.2 BEE Implementation Design Flow

BEE is the acronym for Berkeley Emulation Engine, which is a real-time hardware emulation engine for lower-power digital communication system. Using 20 high-density Xilinx Virtex-E FPGA, each BEE Processing Unit can execute 600 Giga-operations per second and emulating a 10 Million ASIC gate equivalent system [11]. Furthermore, the raw I/O bandwidth of over 200 Giga bits per second can handle some I/O demanding applications, such as ultra wide-band systems, or multi-channel-multi-antenna systems. With the automatic integrated Simulink-to-Implementation design flow, the users can easily implement their design in both BEE system and ASIC, with cycle-to-cycle and bit-true equivalence.

3.6 SF2VHD

Stateflow is a complex graphical language and it does not map easily into hardware, to solve this problem, SF2VHD was emerged at the Berkeley Wireless Research Center. SF2VHD converts state machines in the Stateflow graphical language into synthesizable VHDL for hardware implementation. It utilizes techniques in module

characterization, clock tree generation, performance estimation, and layout generation to quickly design dedicated hardware in a single run [5]. This is a unique methodology that was derived for converting high-level control definitions into hardware.

SF2VHD can find suitable hardware implementations for Stateflow language constructs and achieve an acceptable level of area and power efficiency [6]. While the vast majority of Stateflow features are supported by SF2VHD, several non-essential features were left out for the sake of preserving hardware and simplifying the implementation of the tool. One abandoned feature is arrays.

SF2VHD is written in a hybrid of C and C++ programming languages, and works by parsing in the text file containing the Stateflow description into an internal object model, and then generating the VHDL code for an equivalent state machine. The internal object model resembles the basic functionality of the state machine. The generated VHDL code is composed primarily of two processes. In the first process, Stateflow data types are converted to bit-accurate VHDL data types, both at declaration and “automatically” within each expression operation. In the second process, the Stateflow expression syntax and operators are converted into their VHDL equivalents on a line-by-line basis to implement the functional behavior of the state machine. Functional equivalence to the software model of Stateflow is also provided by an extended synchronous reset and persistent output latches to preserve values across clock cycles [7]. These features are added intrinsically by SF2VHD.

3.7 Summary

The DLL is no longer described by VCC because it is hard to find a way to implement it into hardware. Instead, Matlab Stateflow and Simulink are being used to

capture the behavior of the PicoRadio data link layer, run functional simulations to verify the protocol stack, and run performance simulations to explore potential architectural space. The orthogonalization of function and architecture enables the designer to distinctly separate the behavior description from the actual implementation methods of the protocol, therefore greatly expands the possible design space.

Chapter 4

System Implementation

4.1 Data Link Layer System Architecture

The sub-blocks in Figure 4.1 work together to accomplish the functionalities of the data link layer.

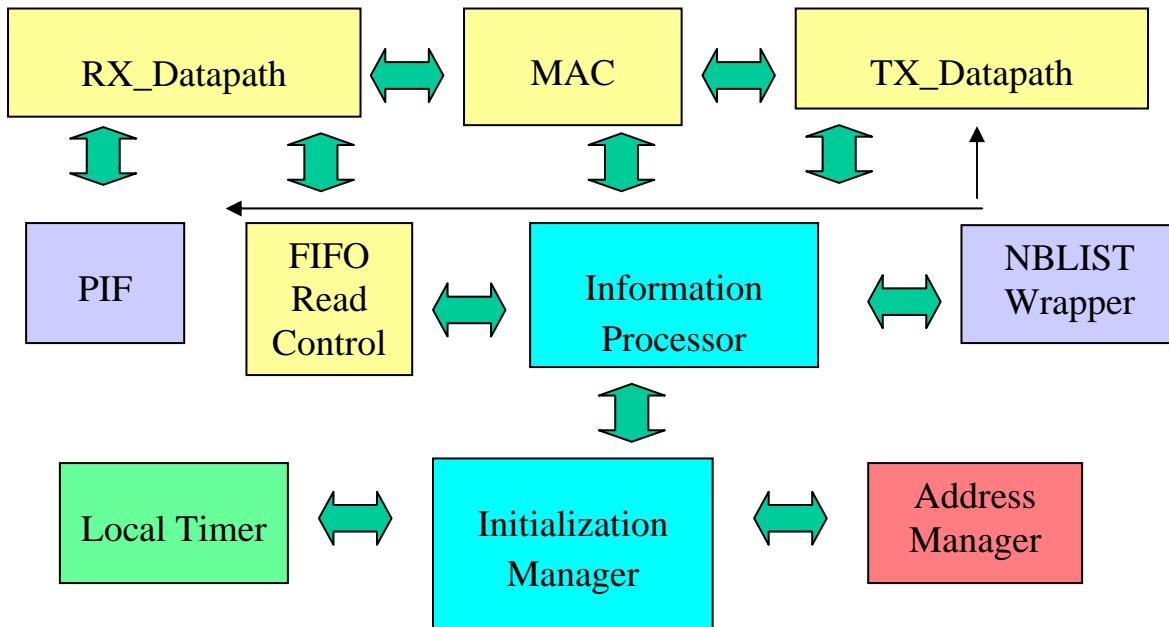


Figure 4.1 Top Level Diagram of Data Link Layer in Matlab Simulink

- ❑ The Initialization Manager (InitMngr) block initializes all the blocks of the DLL.
- ❑ The InitMngr uses the Local Timer to keep track of the initialization processes time interval.
- ❑ The Address Manager block assigns a locally unique address to the node.

- ❑ The Information Processor (InfoProc) block dispatches packets received from the physical layer.
- ❑ The First-In-First-Out (FIFO) Read Control block controls the communication between the receive data path and the Information Processor block.
- ❑ The Media Access Controller (MAC) block controls transmission channel and time.
- ❑ The Transmit DataPath block (TXD) queues outgoing packets and sends packets to the physical layer.
- ❑ The Receive DataPath block (RXD) queues incoming packets from the physical layer and sends packets to MAC in the DLL or to the other layers in the PicoNode protocol stack such as network layer.
- ❑ The Cyclic Redundancy Code (CRC) inside the TXD and RXD blocks encodes and decodes packets for error detection and error correction.
- ❑ The Neighbor List (NBLIST) wrapper is an interface between DLL and the NBLIST Table inside the Neighbor List Service of the Network layer.
- ❑ The Power Management Supervisor Interface (PIF) is used when one block, for example the DLL wants to establish a connection with another block, for instance, the Locationing Block. To accomplish this, the DLL has to notify the Power Management Supervisor of its intent through the PIF, and wait for the Power Management Supervisor to turn on the Locationing Block, before the DLL can communicate with the Locationing Block.

4.2 Initialization Manager (InitMngr) Block

The InitMngr block controls the timing and the procedures for the initialization process of a PicoNode. Its main functionality is to control the timing of all the blocks’

initialization by communicating with the supervisor, the InfoProc and the AddrMngr blocks. The InitMngr block also informs the InfoProc when to send out the new neighbor's information or access the available address space. This block is implemented in the Stateflow diagrams. The top-level view of the block is shown in Figure 4.2 and a detailed Stateflow diagram of this block can be found in the Appendix.

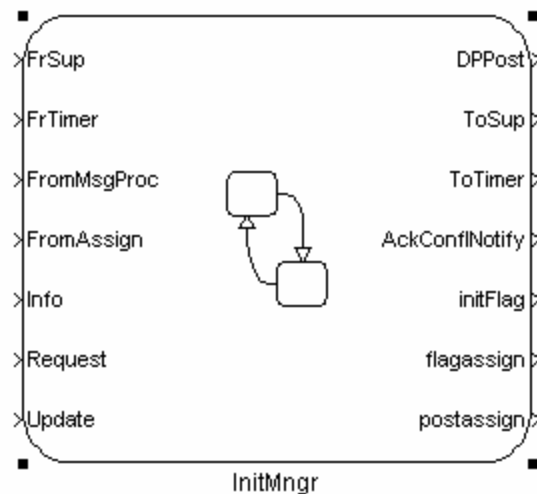


Figure 4.2 InitMngr Input and Output Ports

A PicoNode's initialization procedures are described in six main steps:

1. When the Supervisor sends a signal telling the InitMngr to start the initialization process [FrSup==1], Timer 1 is set to indicate the start of the initialization process. InitMngr notifies the InfoProc block to enter the "init" state by setting 'initflag' to 1. During the 'init' state, the InfoProc does not send conflict notifications; it does not communicate with the network layer or respond to any requests from the network layer or the NBLIST table. It contains no ID and Location (LOC) pair; The InitMngr asks the InfoProc to create an initialization request message (type 0); Send

discovery request (DPPost) and wait for an acknowledgement from InfoProc. The discovery request is send out to discover all the nodes in its neighborhood. Timer 2 is set to measure the time for the neighbor discovery process.

2. When the InitMngr timer expires, this block notifies the AddrMngr block to assign a locally unique address to the node itself.
3. After InitMngr gets the acknowledgement from the AddrMngr block that the self address assignment is complete, it asks the InfoProc block to create an “information” message containing its newly assigned local address, location and neighbor’s addresses; Timer 3 is set to wait for conflict notifications.
4. After Timer 3 expires, if no conflict notification has been received, then the initialization process will end successfully. The InitMngr will then enter the “steady” state, and it will then inform other blocks to do the same.
5. After Timer 3 expires, if there is a conflict notification, then it indicates that some other node in the one-hop neighborhood already claims this ID address. In this case, the InitMngr block commands the AddrMngr block to reassign a local address for the node itself.
6. After the acknowledgement comes back from the AddrMngr block, InitMngr asks the InfoProc block to create an “update address” message (type 4) containing the old and new local addresses; Timer 4 is set to wait for further conflict notifications. At this point, the procedure loops back to step 4 until the conflict has been resolved.

There are several timers that are designed to help with the initialization timing control process. Short time interval wait states are for implementing handshaking

mechanism such as channels. Timeouts are reconfigurable, for instance, the value of the timeouts for waiting for conflicts will be set by the user. There is a different code for each timeouts. Below is an explanation of common commands used in the design.

- ‘To Timer=1’: sets timer 1 interval to the cycle time for the cycle receiver, this is a relatively long time; this command sets the timer 1 for the retransmission of discovery request. Timer 1 specifies the transmission time.
- ‘To Timer=2’ sets timer 2 to the time between transmissions
- ‘To Timer=3’: sets timer 3 to the waiting time after sending all the requests
- ‘To Timer= 4’: sets timer 4 to the time designated to wait for conflict notifications.
- ‘To Timer= 5’: sets timer 5 to the whole time for sending ‘Info’ messages.
- ‘To Timer =6’: sets timer 6 to the interval for each transmission, the supervisor determines the interval of timer 6.
- ‘Frtimer ==1’: timer 1 expires; InitMngr waits to get an event from the supervisor saying location is ready. While waiting for that event, it periodically sends out retransmission of discovery requests.
- ‘Frtimer ==2’: timer 2 expires; InitMngr stops sending out the discovery requests and assigns itself an ID. At this time, timer 2 is canceled and timer 3 is initiated. This command is also used for all the other timers mentioned above to indicate timeouts.
- ‘FrSup==1’: this command means that the supervisor is telling a node to initialize itself.

- 'FrSup==2' this command is sent out by the supervisor when the location of the node itself is ready.

The purpose of these timers is to make sure that the procedure of sending discovery request; waiting for location ready; compute ID in NBLIST and send out the node's information messages to the node's neighbors happens in the right order.

Figure 4.3 is a simple diagram showing the relationship of some of these timers mentioned above. Timer 1 is set to T; this indicates the starting point of the initialization process. Then the 'initflag' is set so that this block notifies the InfoProc to enter the 'init' state. Timer 2 indicates the time between sending discovery requests and timer 3 starts the waiting time for the response.

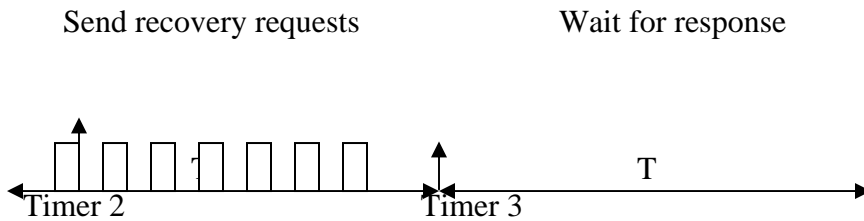


Figure 4.3 Diagram of Timers

4.3 Local Address Manager (AddrMngr) Block

The AddrMngr block randomly assigns a node a locally unique address from the available address space whenever requested by the InitMngr block. This block is implemented in the Stateflow diagrams. Figure 4.4 is a top-level of the block and a detailed Stateflow diagram of this block can be found in the Appendix.

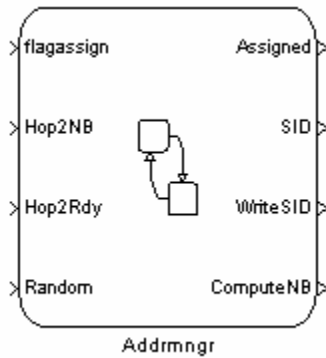


Figure 4.4 Address Manager Input and Output Ports

Assignment of an address is a simple procedure that can be performed in five steps:

1. Whenever an “assign” event comes from the InitMngr block, the random number generator inside this block generates a random number.
2. After the random number is generated, this block reads the number of available address memory, and then scales down the random number to the number of available addresses. In this way an unused local address is chosen.
3. This chosen address is marked on the available address space and the AddrMgr decrements the number of available addresses.
4. The local address is written into the local address memory so that other blocks can access it.
5. The AddrMgr acknowledges the InitMngr block that the self address assignment is completed.

4.4 Information Processor (InfoProc) Block

The functionality of the InfoProc block is to send out, receive, assemble and disassemble received data link layer control packets on behalf of the NBLIST that would

allow the NBLIST to maintain its list so that the NBLIST can perform address resolution for the network layer. The reason for assemble and disassemble of packets is that the network layer and the physical layer only accept and transmit packet byte and byte. This block is implemented in the Stateflow diagrams. The top-level view of the block is shown in Figure 4.5 and a detailed Stateflow diagram of this block can be found in the Appendix.

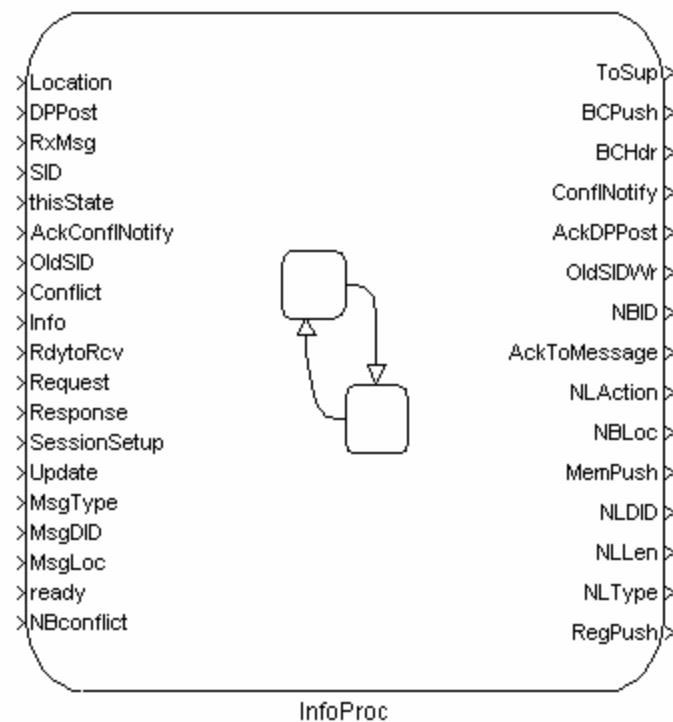


Figure 4.5 Information Processor Input and Output Ports

Whenever InitMgr wants to send out a packet, it informs the InfoProc block. The InfoProc put the message type into the appropriate field and tells the TXD to send out the packet. TXD tells the transmit datapath control (TXSF) block to ask the MAC for a channel. If the channel is ready, TXD sends out the packet to the physical layer. If the channel is not ready, the MAC will do two things depending on the type of the packet. If

the packet is session-based messages, then the MAC will repeatedly send it out. If the packet is not session based messages, then the MAC will only try to send it out once and then the packet, disregarding what type it is, will be kept in the TXD buffer that is big enough to hold the longest Packet. The session based packet types or messages are:

1. RTS (Ready to Send), which means session setup.
2. CTS (Clear to Send) which means ready to receive.
3. DTX (Data to Transmit), which is the actual data that is passed from DLL to the upper layers and passed back.
4. EOS (End of Session), which means the end of a session.

RTS and CTS are produced and consumed by DLL. None of the session based packet types are handled by the InfoProc. The only packet types that InfoProc handles are:

1. Conflict Notification
2. Info (short for Information)
3. Response (short for Discovery Response)
4. Request (short for Discovery Request)
5. Ping
6. Update Address

All the message types are produced during the initialization process except 'Conflict Notification', which is generated inside the NBLIST. InfoProc assembles headers by taking in 'LOC', 'ID and LOC' pair from the NBLIST and then push them out to the Queue. Below is an explanation of all the message types.

- 'Conflict Notification' message is issued when there is a conflict detected and the NBLIST wrapper issues a result code telling the InfoProc whether there is a conflict.
- 'Info' message contains the ID and LOC pair of a node's neighbors.
- 'Response' message are send out by the neighbors as a response to the 'discovery request' message that the node has sent out. The 'Response' message is identical to the 'Info' message in that it contains the ID and LOC pair of the neighbor; the difference is that the 'Response' message has a different header than the 'Info' message.
- 'Request' message is send out by a node to discover its neighbors.
- 'Ping' message is send out by the Maintenance block. Its purpose is to periodically discover neighbors while the cycle receiver is off. It is similar to the 'Request' message except that 'Request' message is send out when the cycle receiver is on.
- 'Update Address' message contains the old ID and the new ID and LOC pair so that the NBLIST can use the old ID to know which entry should be updated. Update action is performed by deleting the old ID and adding the new ID and LOC to that address space.

Procedures for the InfoProc block tasks are:

- I. The InitMngr requests InfoProc block to perform a task via the “DPPost” port:
 - a. If the InitMngr block asks the InfoProc block to create an “update address” message (DPPost=Update), The InfoProc will fill in the message type field as “4” in the old address field, then forwards the message to the FIFOCtrl block.
 - b. If a ‘Conflict Notification’ message is received by InfoProc from the result code in the NBLIST wrapper, the InfoProc will assemble the conflict message and send it to the InitMngr.

- II. The Receive Datapath (RXD) block informs that a data link layer control message has been received via the “FromRX” port:
 - a. If a “response” message is received (FromRX = 1) and the node is in the initialization state, then this block updates the available address space memory and attaches the sender’s address, location and link metric to the end of the neighbor list. If this address is already in the list, then it creates a “select location” message (type 5).
 - b. If an “information” message is received (FromRX = 2) and the node is in the steady state, this block updates the available address space memory, and appends the sender’s information to the end of the NBLIST. If the address is already in the

- list, then it checks if the nodes' own address is in the sender's NBLIST. If yes, it informs the InitMngr block to reassign the local address.
- c. If a “conflict notification” message is received (FromRX = 3) and the node is in the initialization state, then an address conflict is detected. This block notifies the InitMngr block of the conflict and then appends the sender's address to the end of the NBLIST. If the address is already in the list, it will create a “select location” message (type 5) to ask the sender to reassign its address.
 - d. If an “update address” message is received (FromRX = 4) and the node is in the steady state, this block will update a neighbor's address by updating the available address space and replacing the old address with the new one in the NBLIST.

4.5 FIFO Control Block

The functionality of the First-In_First-Out Control (FIFOCtrl) block is to control the timing of the data flow elements within the TXD. Such data flow elements include the serializer/deserializer word clock, line balancer, CRC and the transmit/receive buffers. The FIFOCtrl block also controls the sinking of the word clock within the TXD. This block is implemented in the Stateflow diagrams. Figure 4.6 is a top-level diagram of the FIFOCtrl block and a detailed Stateflow diagram of this block can be found in the Appendix.

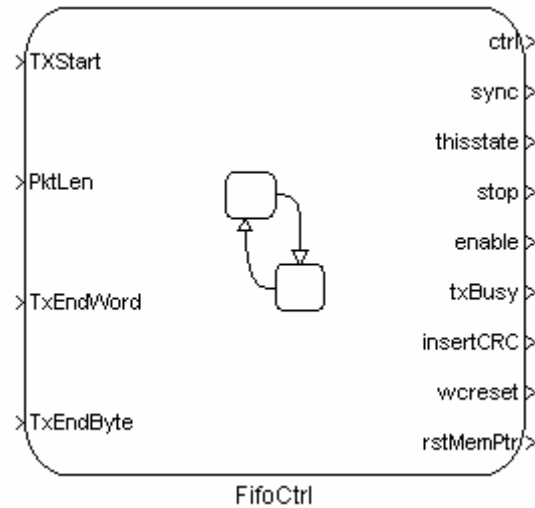


Figure 4.6 First-In_First_Out Control Block Input and Output Ports

4.6 Transmit Datapath (TXD) Block

The TXD block queues outgoing packets from the network layer and data link layer, requests transmission permission, and sends out the packets. There are two queues, one for broadcast packets and one for unicast packets. The queues are indexed by channel ID (address) of the receiver of packets. The TXD block contains a transmit datapath control block that is implemented in the Stateflow diagrams. It also each contains the typical datapath components, such as transmit/receive buffers, serializer/de-serializer, cyclic redundancy code (CRC), line balancer and word clock. This block is implemented in both the Stateflow and the Simulink/Xilinx diagrams. The top-level view of TXD block is shown in Figure 4.7 and a detailed diagram of this block can be found in the Appendix.

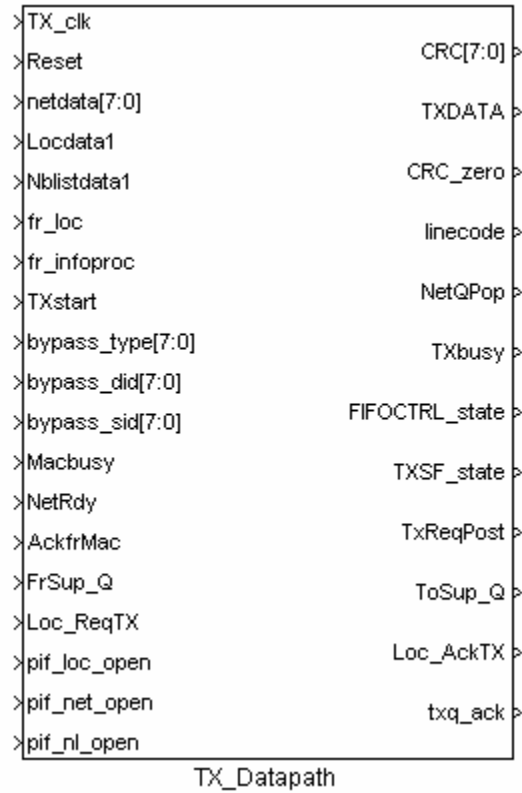


Figure 4.7 Transmit Data Path Block Input and Output Ports

- I. Transmitting a network unicast packet
 - a. If the queue is empty, a queue timer will be set to the waiting time of the packet.
 - b. If the queue is not full, packets will be pushed the packet into the queue, and the number of packets parameter will be incremented.
 - c. If the queue is full, discard the packet.
 - d. If the number of packets reaches its maximum, cancels the queue timer. To set up the unicast session with the destination node, the TXD block sends a transmission request to the MAC block for a “session setup” message. A network unicast packet has the third transmission priority.

- e. If any timer associated with a unicast queue expires, the packet in the queue will be sent out immediately. The TXD block sends a transmission request to the MAC block for a “session setup” message.

II. Transmitting a network broadcast packet

The TXD block first buffers the packet in the broadcast queue, and then sends a transmission request to the MAC block. A network broadcast packet has the second transmission priority. Whenever a transmission permit is granted, the packet will be sent out immediately.

III. Transmitting a data link layer control packet

If a message is send out by the InfoProc block, the TXD block will buffer it in the broadcast queue, and sends a transmission request to the MAC block. A data link layer packet has the first transmission priority.

4.7 Receive Datapath (RXD) Block

The RXD block dispatches network layer packets, the location engine packets and data link layer control packets received from the physical layer. The RXD block contains a transmit datapath control block (RXSF) that is implemented in the Stateflow diagrams. It also each contains the typical datapath components, such as transmit/receive buffers, serializer/de-serializer, cyclic redundancy code (CRC), line balancer and word clock.

This block is implemented in both the Stateflow and the Simulink/Xilinx diagrams. Figure 4.8 is a top-level diagram of the receive data path block and a detailed diagram of this block can be found in the Appendix.

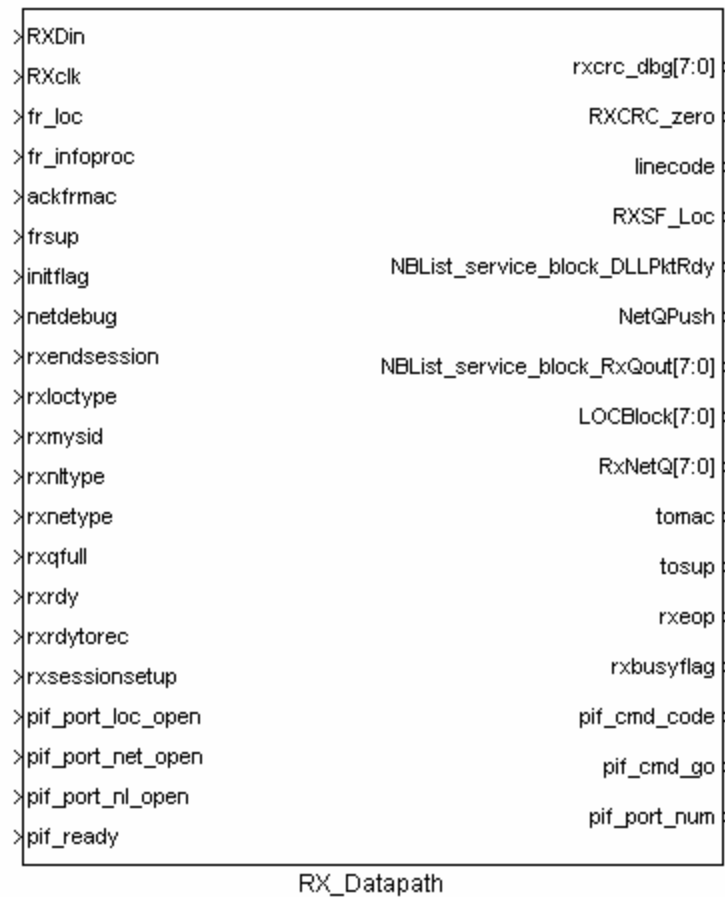


Figure 4.8 Receive Data Path Block Input and Output Ports

- I. If the packet is a broadcast packet, then the RXD block dispatches the packet according to its type field.
 - a. If the message type is “discovery request”, then the RXD block forwards the packet to the MAC block.

- b. If the message type is “response”, “information”, “conflict notification” or “update address”, then the RXD block forwards the packet to the InfoProc block for further processing.
 - c. If the message type is “session setup”, then the RXD block forwards the packet to both the MAC and the InfoProc blocks.
 - d. If the message type is “network broadcast data”, then the RXD block forwards the packet to the network layer.
- II. If the packet is a unicast packet
- a. If the message type is “network unicast data”, then RXD block sends the message type and sender’s address to the MAC block to check if there is return data. It also forwards the packet to the network layer.
 - b. If the message type is “ready to receive” or “end session”, then RXD block sends the message type to the MAC block. For “end session” message, this block also forwards the packet to InfoProc block for further processing.

4.8 Media Access Control (MAC) Block

The MAC block is implemented in Stateflow diagrams. It allocates physical channel resources, controls and coordinates actions of transmitting and receiving packets. The MAC block designates a radio channel for transmission, and performs random back off when needed. The new feature of this version of DLL is that the MAC block supports carrier sensing mechanism (CSMA) and the sleep mode. Figure 4.9 is a top-level diagram of the MAC block and a detailed diagram of this block can be found in the Appendix.

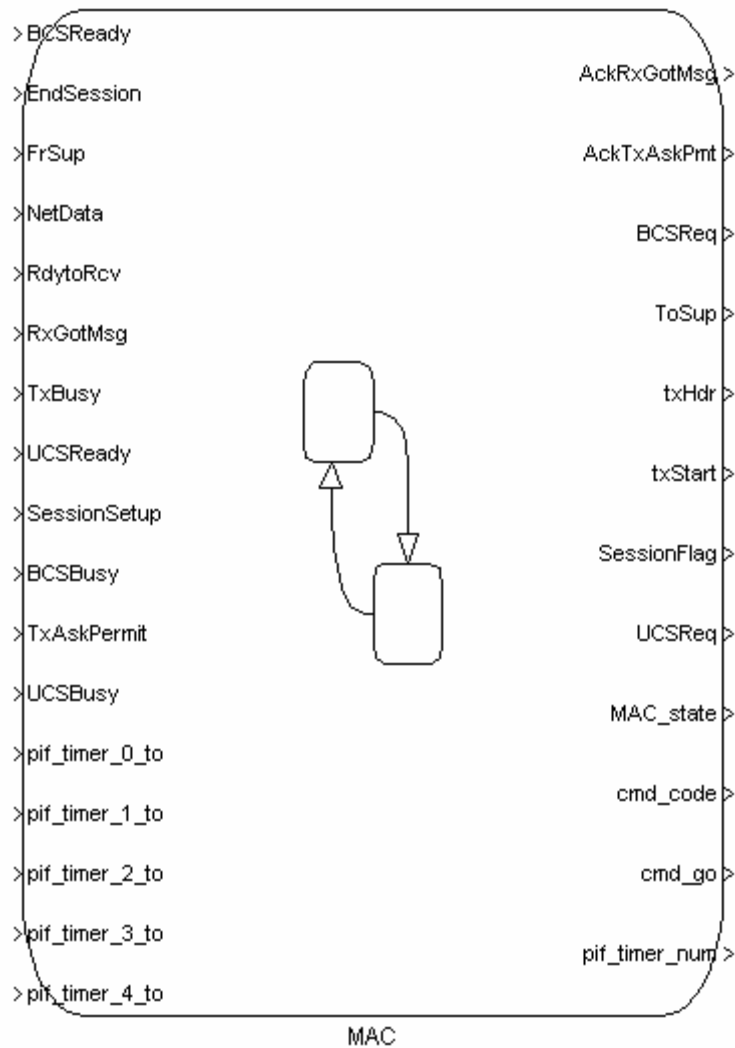


Figure 4.9 Media Access Control Block Input and Output Ports

For simplicity of explanation, a sender node's channel is used to transmit all packets during a unicast session, instead of toggling the channel as stated in the protocol description. The MAC block has three operating modes: Initialization, Session setup and Random back off.

I. Initialization

This block is in the “startup” state when first powered up. It enters the “initialization” state when the InitMngr block starts the initialization process. During this state, the channel is set for broadcast, the transmit datapath control (TXSF) block waits for acknowledgement from the MAC to start transmitting data link layer control messages. Whenever the initialization process ends in the InitMngr block, MAC block enters the “steady” state, listens to the broadcast channel, and informs the TXSF block that network packets can be handled.

II part I. Session Setup as a Sender Node (TX and MAC)

- a. If the TXSF block asks for a transmission permit for a ‘session setup’ message when MAC is in ‘steady’ state, the MAC block will create a ‘session setup’ message. It denies the transmission permit for now since the message has not been created yet. The MAC block enters the ‘session setup’ state.
- b. The TXSF block asks permit for the ‘session setup’ message again after the MAC block creates it. The MAC block grants the permit and sets a channel for broadcast, then it tunes to the node’s own unicast channel for receiving packets. The MAC timer is then set to wait for an acknowledgement.
- c. If the timer expires and the number of retries is less than 3, then the MAC will retransmit the ‘session setup’ message by going back to (a).

- d. If the timer expires and the number of retries is equal to 3, then the MAC will give up and goes back to 'steady' state.
- e. If the RXSF block gets a 'ready to receive' message (type 9) before the timer expires, the MAC block cancels the timer, grants permit for unicast data transmission, then starts a new timer waiting for acknowledgement.
- f. The Message block creates an 'end of session' message when the session timer expires and it goes back to 'steady' state and tunes to the broadcast channel.
- g. If the RXSF block gets an 'end of session' packet (type 10), it cancels the timer, goes back to 'steady' state, and tunes back to the broadcast channel.
- h. If the timer expires and the number of retries is less than 3, then it will retransmit the network unicast data and goes back to (f).
- i. If the timer expires and the number of retries is equal to 3, then it will give up transmission, goes back to "steady state", and listens to the broadcast channel.

II part II. Session Setup as a Destination Node (RX and MAC)

- a. If the RXD block receives a 'session setup' packet (type 6) and this block is in "steady' state, it asks the MAC block to create a "ready to receive" message (type 9). Whenever the MAC block requests permit for the message, it grants permit and sets channel to the sender node's ID for transmission, and then listens to the channel.

- b. If there are data in the transmit queue, then the MAC listens to the channel. It waits until an 'end of session' packet is received, and then goes back to "steady" state and listens to the broadcast channel.

III. Random Back off

In order to avoid potential conflict with other responding neighbors in the broadcast channel, the MAC block performs a random back off based on CSMA. The MAC interacts with Baseband to receive status information such as when a packet is ready to be sent, so it knows exactly when to back off.

MAC block creates outgoing data link layer control packets by filling in all applicable fields, whenever requested by the InitMngr and InfoProc.

Procedures for the MAC block are:

- I. The InfoProc block requests to create a message.
 - a. If the message type is a "response" message that is a response to a 'removal warning' message received earlier. The MAC will specify the channel number as "0" to indicate that this channel is designated for broadcast, fills in the fields of the message type, the node's address, location, number of neighbors and neighbors' addresses, and forwards the message to the TXD block.

- b. If the message type is a “conflict notification” message. The MAC will specify the channel number as “0”, fills in the fields of the message type, the node’s address, location, number of neighbors and neighbors’ addresses, and forwards the message to the TXD block.
- c. If the message type is an “update address” message. The MAC will specify the channel number as “0”, fills in the fields of the message type, the node’s old and new addresses, and forwards the message to the TXD block.

II. The InitMngr block asks to create a message.

- a. If the message type is an “initialization request” message. The MAC will specify the channel number as “0”, fills in the message type field, and forwards the message to the TXD block.
- b. If the message type is a “response” message to reply an “initialization request” message received. The MAC will specify the channel number as “0”, fills in fields of the message type, the node’s address, location, number of neighbors and neighbors’ addresses, and forwards the message to the TXD block.

4.9 Summary

This chapter discusses the functionalities of each PicoRadio data link layer sub blocks and how each block is implemented to perform their functionalities using timers, input/output ports and inter-block interactions. Seven major functional blocks are depicted in the symbol view and the detailed execution of each block is shown in the Appendix.

Most of the DLL blocks design was based on the previous DLL work by Mei Xu. The NBLIST Wrapper and the PIF block were implemented mainly by Fred Burghardt. Major modifications of each block were completed under the valuable supervision of Dr. Johnathan Reason.

Chapter 5

Analysis of Results

The description of the data link layer was completed using Matlab Simulink and Stateflow tools. Simulation and analysis of the results of the entire data link layer in several key scenarios such as one node loop test and two nodes inter-communication scenario will be discussed. The Stateflow part of the DLL is converted into VHDL file with the help of SF2VHD and the Xilinx system generator automatically generate VHDL file for the Xilinx datapath blocks. The two VHDL files are then combined and the resulted VHDL code is ported through BEE design flow, in order for the DLL design to be built on FGPAs. A concurrent effort was done to bring the final debugged version of DLL all the way to ASIC implementation. In this process, the combined VHDL files was fed into INSECTA and with the aid of Synopsys complier, design and logic compiler and backend place and route, the resulting “bug fixed” VHDL file was used directly for ASIC design. The entire DLL was translated into VHDL files for further synthesis. The total resulted gate count is 11703.

5.1 Simulink/Xilinx Simulations

Datapath components such as CRC and serializer/de-serializer was built and tested in Simulink/Xilinx, but their simulations will not be included in this report for the following reasons. First, these blocks are commonly used and the designs are already standardized [12]. Second, the behavior of the data link layer will be very similar if these

blocks are bypassed and the input/output ports are connected with packet structures directly to the simulated physical layer, thus the function verification of the protocol will be preserved. These blocks can then be easily added back during synthesis.

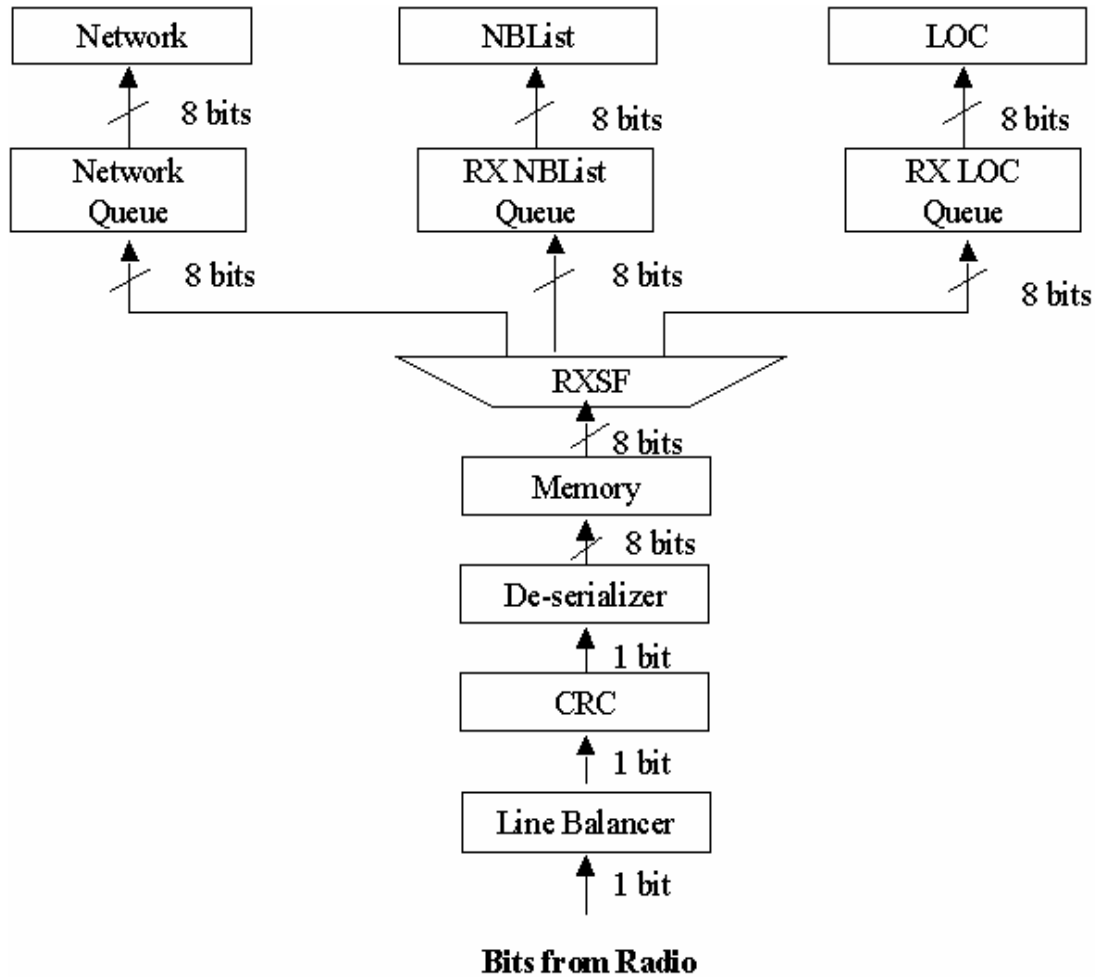


Figure 5.1 Top Level Diagram of Simulating the Receive Datapath

The process of simulating the RXD is shown in Figure 5.1. As described in this diagram, a simulated 1 bit data signal is sent from the RF radio/Baseband. This 1 bit data signal first enters the line balancer of the RXD. Then it is processed and propagated through the CRC and the serializer/de-serializer block. The de-serializer de-serializes this

1 bit signal into 8 bits and then it is sent into the memory buffer. A multiplexer is waiting there to select the destination of these 8 bit data based on the select signal produced inside the RXSF control block. According to the multiplexer, these 8 bits data will either go to the network queue, the NBLIST queue or the LOC queue. This data signal will remain in the queue until it is its turn to be processed and sent to the appropriate destination, such as the network layer, the NBLIST table or the locationing block. The process of simulating TXD is similar to the RXD simulation process except the data signal would come in from one of these other protocol layers and be transmitted out to the Radio/Baseband. The top level diagram of the transmit datapath would be exactly the same as shown in Figure 5.1 except the path that the signal takes is in the reverse order.

5.2 Functional Simulations

Functional simulations were performed based on discrete-event models and data-flow models. Execution of an event takes zero time. Functional simulations are used for debugging and design verification purposes. Fake traces are inserted at input ports to create the desired simulation scenarios. Probes and scopes are placed on output ports and certain crucial midway places to view the simulated results. Simulation scenarios or test vectors for the whole DLL system are thought through in advance and a predicted result for each simulation scenario is used to compare with the actual simulation results to see if the block under test is functioning correctly.

5.2.1 Single Node Self Loop Test

Single node loop scenario is the first verification step towards testing if the DLL basic functionalities were designed correctly. In this scenario setup, the output of the

TXD was directly wired to the input of the RXD. A fake data signal was then fed into the RXD and following the simulation traces, it can be seen that this signal has propagated through the correct designed routes all the way to the output of the TXD. A more illustrative and useful scenario is the two node scenario which is mentioned in the following section.

5.2.2 Two Nodes Transmit and Receive Test

Figure 5.2 shows the conceptual idea of testing two nodes communication scenarios. In this scenario setup, the datapaths of the two identical data link layers were directly connected together. This scenario is set up to watch the communication between two nodes, 1 and 2. A random input data was generated using a random number generator in the Simulink library and it is inserted at the receive side of node 1, a test scope is placed at the output of the transmit side of node 2 to see if the resulted data signal matches the input data. This test was performed multiple times, each time with a different randomly generated data signal to make sure that the input data that was fed into node 1 was correctly received and produced by node 2.

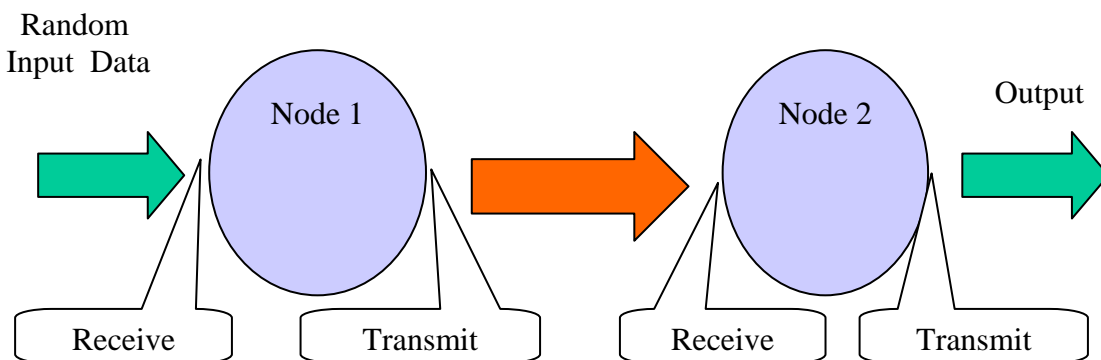


Figure 5.2 Two Nodes Scenario Diagram

The actual setup of the two nodes communication scenario was performed in Simulink environment. Figure 5.3 shows the middle section of the conceptual setup where the transmit side of node 1 is directly connected to the receive side of the node 2. In this simulation, the InitMngr timer of node 1 is set to expire after 20 seconds in order to trigger the initialization process. During this initialization time, if node 1 finds no neighbor, it will assign itself a local address and broadcast this address information. After 200 seconds into this simulation, the InitMngr timer of node 2 expires and node 2 will start its initialization process. In the ideal situation, the two nodes should discover each other at this time, and form a neighborhood and thus a link. It is shown in Figure 5.3, the 'RXDin' box contains the random data signal inputed to node 1. This signal was passed into the 'Din' (Data In) port of TXD CRC block of node 1. This signal then comes out of the 'Dout' (Data Out) of the TXCRC and was fed into the 'Din' port of the TXD line balancer. The resulted 8 bit data comes out of the 'Dout' port of the TXD line balancer of node 1 and it is directly transferred to the 'Din' port of node 2's RXD line balancer. This data finally arrives at the 'Dout' of node 2's receive side. Test scopes are placed in each 'Din' and 'Dout' port to monitor the process of receiving and transmitting this random data signal. If the simulation is successful, the test scope at the 'Dout' port of node 2's TXD will show the same waveform as the test scope placed at the 'Din' port of node 1's RXD. This correct result also requires that the scope placed at the 'Dout' port of node 1's TXD match the test scope waveform placed at the 'Din' port of node 2's RXD.

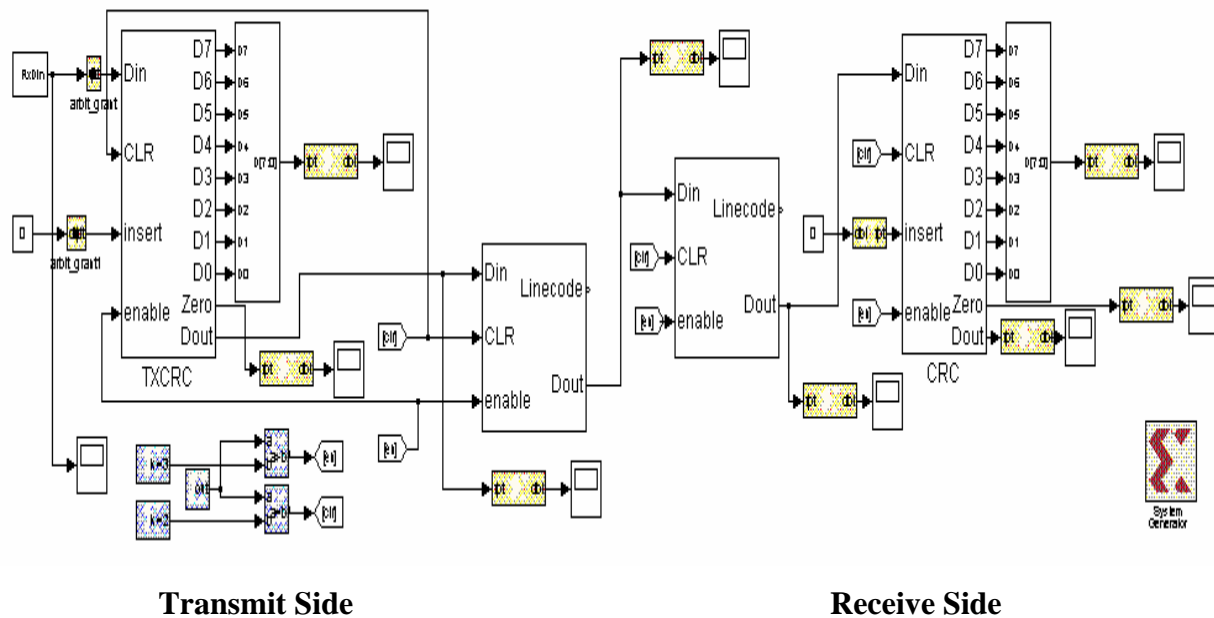


Figure 5.3 Functional Simulation Setup Diagram

Figure 5.4 shows two test probes results simultaneously. One of these test probes is placed at the input of node 1 and the other is at the output of node 2. In Figure 5.4, the top waveform is produced by the input test scope at the 'Din' port of node 1's RXD and the bottom waveform is produced by the output test scope at the 'Dout' port of node 2's TXD. The graph shows that the results of these two scopes indeed match each other.

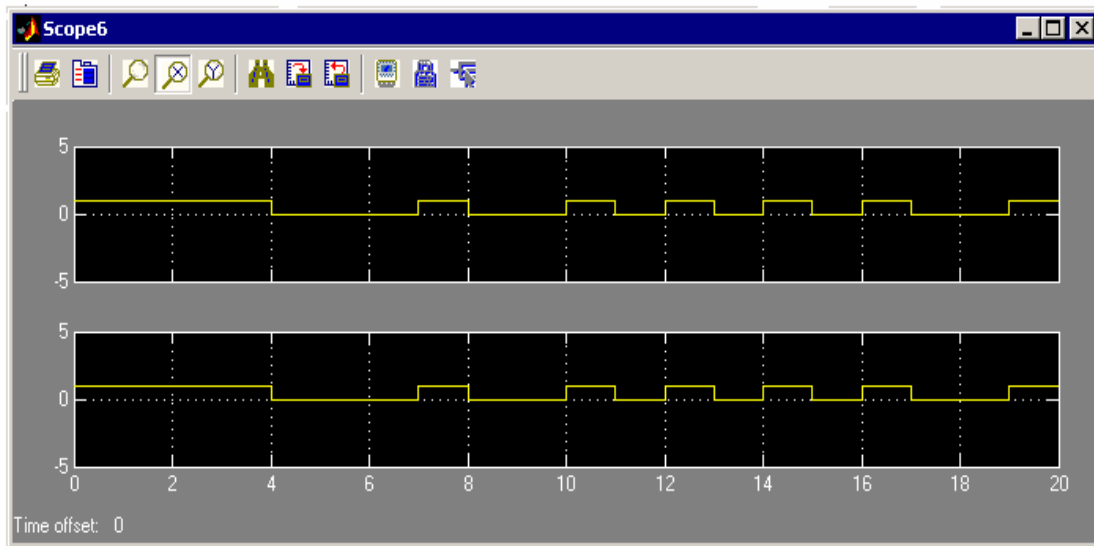


Figure 5.4 Matlab Results of Two Nodes Scenario Simulation

5.3 Stateflow Simulations

Stateflow is a graphical and textual language that is primarily based on the original State Charts language defined by Harel [5]. The Stateflow tool is based on behavioral models and used for functional simulations only. Implementation decisions are made based on experiences. It uses a software-based execution model to provide essential features such as state hierarchy, arrays, parallel processes, event triggers, and multi-decision transition conditions. To give an example of stateflow simulations, the stateflow diagram of the MAC block is displayed in Figure 5.5.

This current version of the MAC block supports the sleep mode of the physical layer by using a cycled receiver with a parameterized duty cycle as explained in Chapter 4. The MAC block uses clock-gating technique to switch between TX/RX modes in the physical layer. To test the functionalities of the MAC block, fake traces is again

inserted at the input. The path of signal travel is highlighted one after another to show the designer where the signal is located at any instance during the simulation process. If the signal path is highlighted in the corrected order during the running simulation period, then this design has passed its initial verification step.

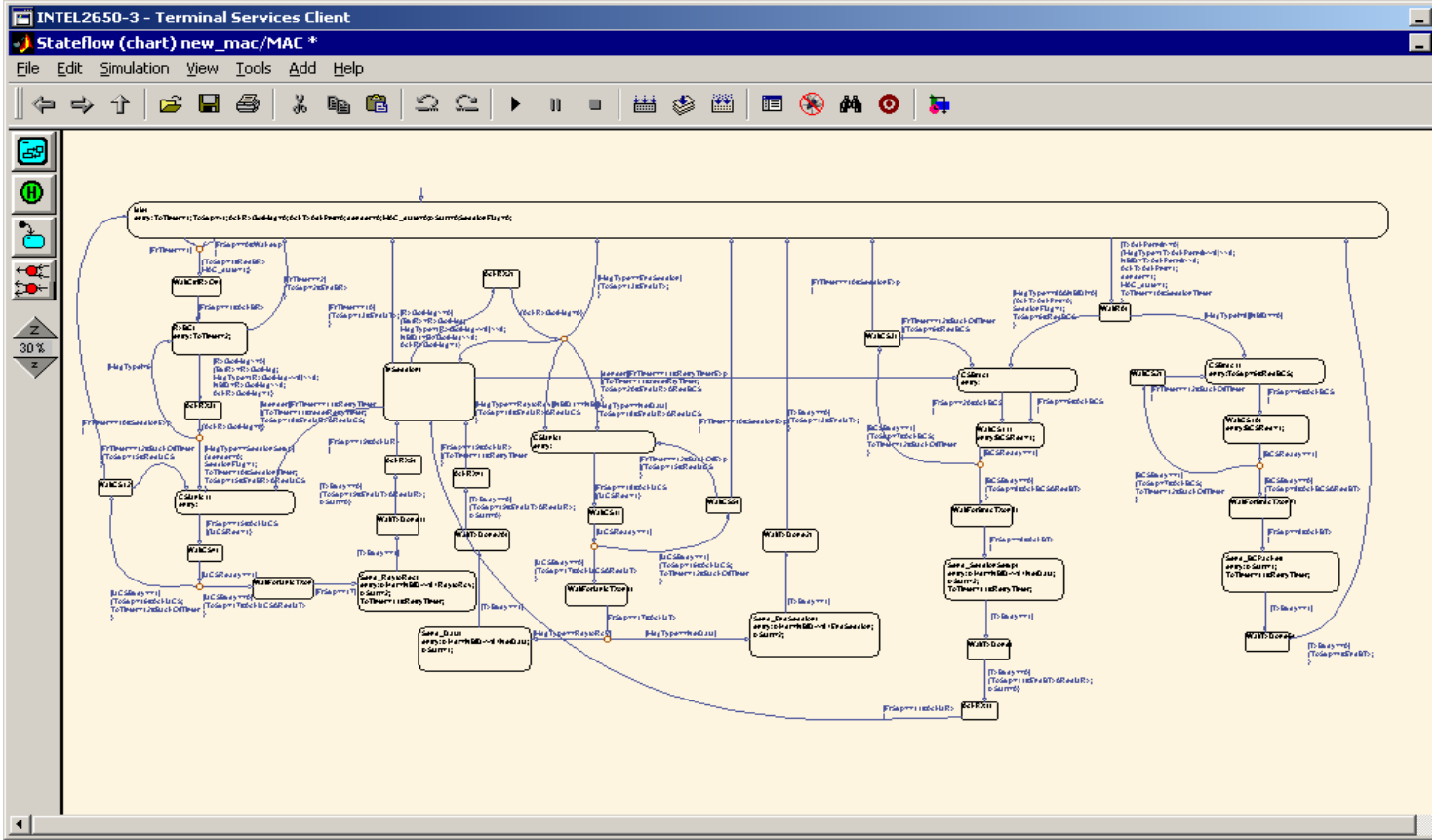


Figure 5.5 MAC Implementation in Stateflow

All the Stateflow blocks such as InitMgr or the InfoProc were tested in similar procedures. Mei Xu and Dr. Reason have contributed significant efforts in designing and simulating most of these stateflow blocks.

5.4 Generating VHDL Files

Porting of the DLL into the ‘bug fixed’ synthesizable VHDL codes was successfully completed. This accomplishment allowed the comparison of the verified DLL Simulink behavior with its VHDL behavior. The generation of VHDL files follows a simple procedure that is outlined below:

1. Setup the INSECTA environment, Make a BEE Project folder.
2. Simulate in the Simulink environment.
3. Run Xilinx System Generator to produce the raw VHDL files for Xilinx blocks.
4. Run BEE_ise to generate SF2VHD files for the stateflow blocks.
5. Run the INSECTA flow to debug and fix the files from step 2.

Several major bottlenecks during the porting of the DLL to the VHDL files are listed below:

- Stateflow arrays that are used to design the NBLIST table can not be processed by the BEE tool flow, thus the NBLIST table was taken out of the DLL and was directly created via VHDL codes.
- Algebraic loops was a problem whenever an output signal of a block is fed back to the same block as an input even after it goes through some other blocks. Figure 5.6 displays such an instance, the InitMngr block needs an independent fake timer block to set up the values for the InitMngr timers, so the output port ‘To Timer’ of the InitMngr is connected to the input port ‘set timer’ of the fake timer block and comes out of the port ‘Timer Expire’ of the fake timer and this signal is fed back to the

input port 'frtimer' or 'From Timer' of the InitMngr block. This setup will not cause any problems during the stateflow behavioral simulation; however it will create an algebraic loop when the stateflow blocks are simulated with the Simulink/Xilinx blocks. The solution to this problem is that whenever there is such a timing loop problem; specify the explicit sampling time for all the components in the loop and add a delay block as shown in Figure 5.6.

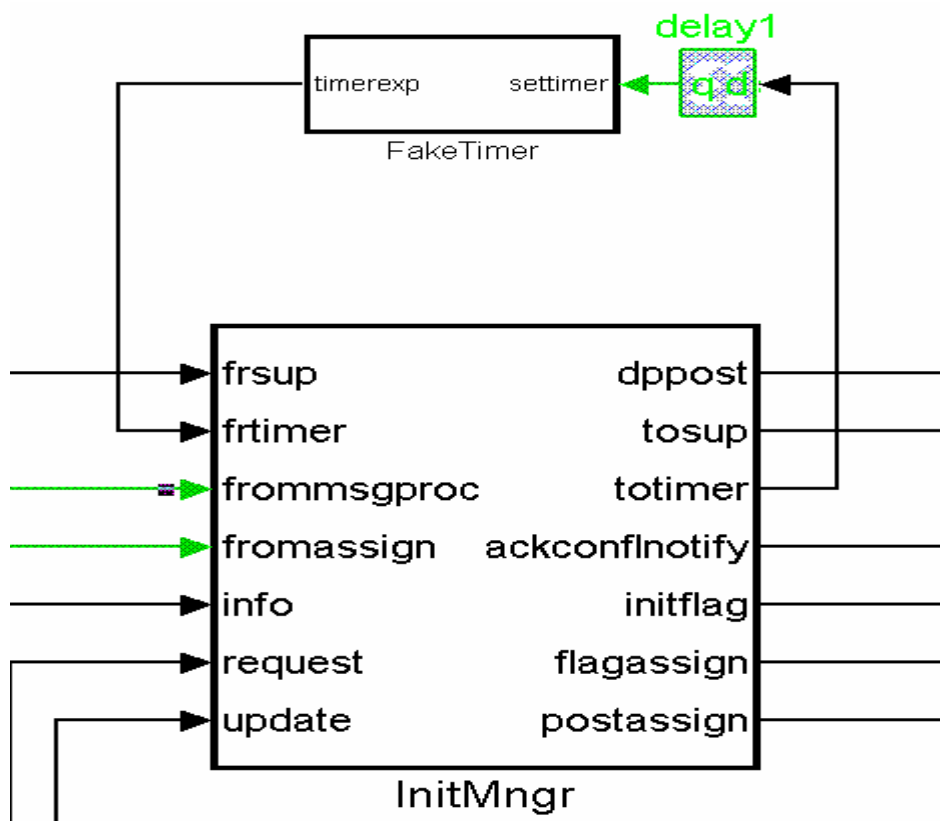


Figure 5.6 Algebraic Timing Loop

- The port type 'bool' meaning Boolean in the Stateflow blocks does not compile when it is connected to the same type port labeled 'Boolean' inside a Simulink/Xilinx block. In order to solve this problem, the port type 'bool' was changed to 'auto' for all the stateflow blocks.

5.5 Summary

This chapter summarizes the verification processes of the PicoRadio data link layer protocol. The single node self loop test and the two-node simulation helped to debug the DLL. An example of the stateflow simulations is shown to demonstrate the process of verifying any stateflow block functionalities. These analyses points to some useful guidelines for further improvements on implementation decisions. One lesson learned from the simulation process is that packet types such as ‘Conflict Notification’, ‘Info’ and ‘Ready to Receive’ should be parameterized instead of being assigned a constant value inside the DLL. Two main reasons that calls for this decision is that:

1. It is easier to add a new packet type later when necessary.
2. If after the entire DLL has been designed and tested, it is found that the current packet types may not be appropriate for the tasks and more space in the packet length is desired.

In these two instances, it makes more sense to bring these message types to the outside of the DLL and make them as inputs so that the user can change these message types as desired. Parameterization makes the DLL programmable at run time and it is used for debugging purposes. Timer values are also parameterized, so they can be set in software.

With tremendous help from Ruth Wang, Tufan Karalar, Mike Sheets and Dr. Jonathan Reason, the entire DLL was synthesized with the rest of the PicoRadio chip components. The total resulted gate count of the DLL is 11703 gates.

Chapter 6

Conclusions

PicoRadio III or PicoNode3 (PN3) project is a group effort intended to build a chip, named “Quark” in the ASIC environment. Simulink and Stateflow tools were chosen to implement the data link layer so that the DLL can be ported to the VHDL files and be built both on FPGAs and ASIC chip. New features are added into the Quark, such as power management and sleep mode to pursue lower power consumption. A CSMA scheme is also incorporated into the MAC block of the DLL to adapt to the physical layer changes. The current design of DLL is able to collect statistics such as network traffic patterns and packet error rate, this information is used to further understand the sensor network behaviors and provide feedback on its improvement. The main lessons learned from working on this project have been mentioned in each chapter’s conclusion section. Thus, in this final chapter, a comment on using the various design tools will be briefly discussed.

6.1 Using Stateflow

Stateflow works very well with Simulink and MATLAB, Its combination of graphical modeling and animated simulation improves the design process. Stateflow blocks can be included in a Simulink model and simulate together with Simulink blocks. Therefore, behavioral level co-simulations of the protocol stack can be simulated with other layers in a node, and of multiple nodes to watch the network behavior. Unlike VCC, which includes architectural models and performance analyses, Stateflow deals with

behavioral models and functional simulations only. Implementation decision was made based on the designer experiences instead of on the analysis results in VCC. The user interface of Stateflow is user-friendly and easy to learn compared to that of VCC. The debugging environment in Stateflow is also better than in VCC because Stateflow simulations can show states inside of the state transition diagrams.

6.2 Using SSHAFT/BEE

Since there is no synthesis path from Simulink to VHDL, the SF2VHD from SSHAFT translator was used to compile the Stateflow CFSMs to VHDL. The SF2VHD only supports a subset of Stateflow, which is enough for our application. The BEE tool flow was never used on a huge and complicated system block such as the DLL. There is room for debugging and improving the BEE tool flow. One such improvement could be to allow strings in counters, which is a common occurrence in digital communication designs.

6.3 Pros and Cons of Using VCC

The DLL is no longer described by VCC because it is hard to find a way to implement it into hardware. It is a great concept to include high-level behavior captures, functional verification, architecture exploration, and performance analysis in one tool. The underlying model of block to block communication in VCC is a non-blocking buffer of size one between the blocks. Therefore, upon the arrival of the next event, the previous event will get lost if it has not yet been consumed. To make communication between blocks reliable, a handshaking mechanism must be implemented. However this separation of computation and communication is not allowed in VCC. A next generation platform

called Metropolis is in the research phase, in which models of communication are explicitly stated and design specific. VCC should provide various accurate architectural models, such as ASICs and processors, in order to get good performance estimation results. The current models are too simple to aid in making reliable design decisions based on the performance analysis results. No native support for power estimation and analysis, which is crucial to low-power designs like the PicoRadio project. Although Cadence suggests calling power architectural services through the behavior description, this method violates the principle of function and architecture orthogonalization [8]. Cadence revoked its promise for providing link to implementation in VCC. To find a synthesis path, the design in VCC has to be ported manually to other tools, which is unwise.

Chapter 7

Glossary

DLL:	Data Link Layer
Packet:	a group of bits, typically from a few hundreds to thousands that are transmitted at precise times. Information is transmitted in the form of packets.
PIF:	Interface to the Power Management Supervisor
CRC:	Cyclic Redundancy Code
CSMA:	Carrier sense multiple access protocol
Broadcast:	Transmission over a channel that all nodes listen to
Session:	Unicast transmission mode
Handshaking:	request and acknowledgement between two nodes
RTS:	Ready to send; session setup
CTS:	Clear to send; ready to receive
DTX:	Data to transmit; data that is transmitted to and from DLL to upper layers.
EOS:	End of Session
DPPost:	Send discovery request
NAMP:	Group Network, Application, MAC and Positioning Group
Node:	PicoNode
PicoNode:	A unit of the PicoRadio Network

PicoRadio I:	The first phase of the PicoRadio project; TestBed
PicoRadio II:	The second phase of the PicoRadio project. Multi chip implementation
PicoRadio III:	The final phase of the PicoRadio project. System on a chip
SMS:	Statistic and Management Service
SMS Controller:	Controls the other part of the SMS
SMS Recorder:	Records packet information. Part of the enhanced SMS service

References

- [1] Jan Rabaey, J. Ammer, J.L. Silva Jr., D Patel, S. Roundy, “PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking”, Computer Magazine IEEE, July, 2000
- [2] Jan M. Rabaey, Josie Ammer, Tufan Karalar, Suetfei Li, Brian Otis, Mike Sheets, Tim Tuan, “PicoRadios for Wireless Sensor Networks— The Next Challenge in Ultra-Low Power Design”, Proceedings of the International Solid-State Circuits Conference, San Francisco, CA, February 3-7, 2002
- [3] Jan M. Rabaey, Robert W. Brodersen, Kannan Ramchandran, Paul Wright, M. Josie Ammer, Fred Burghardt, Rong Chen, Jana van Greunen, Chunlong Guo, Tufan Karalar, Gary Kelson, Mika Kuusula, Suet Fei Li, En-Yi Lin, Susan Mellers, Richard Lu, Michael Montero, Dan Odell, Brian Otis, Huifang Qin, Johnathan Reason, Shad Roundy, Ulrich Schuster, Marco Sgroi, Rahul Shah, Michael Sheets, Mei Xu, Charlie Zhong, *PICORADIO: (COMMUNICATION / COMPUTATION PICONODES FOR SENSOR NETWORKS) Final Report*”, Berkeley Wireless Research Center, December 2002
- [4] Fred Burghardt, Susan Mellers, Jan Rabaey, “The PicoRadio Test Bed”, Berkeley Wireless Research Center, December 30, 2002
- [5] K. Camera, “SF2VHD: A Stateflow to VHDL Translator,” M.S. May 2001, pp1
- [6] K. Camera, “SF2VHD: A Stateflow to VHDL Translator,” M.S. May 2001, pp2
- [7] Rahul C. Shah and Jan Rabaey, “Energy Aware Routing for Low Energy Ad Hoc Sensor Networks”, IEEE Wireless Communications and Networking Conference (WCNC), Orlando FL, March 17-21, 2002. [CDROM:\References\wcnc.rahul.pdf]

- [8] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. W. Brodersen, B. Nikolic, "Rapid Design and Analysis of Communication Systems Using the BEE Hardware Emulation Environment," Proc. IEEE Rapid System Prototyping Workshop, June 2003
- [9] M. Xu, "Implementation and Simulation of PicoRadio Data Link Layer in VCC," M.S. January 2003, pp12
- [10] MathWorks on the web about Xilinx and Simulink
- [11] K. Camera, "SF2VHD: A Stateflow to VHDL Translator," M.S. May 2001, pp5
- [12] M. Xu, "Implementation and Simulation of PicoRadio Data Link Layer in VCC," M.S. January 2003
- [13] Jean Walrand, "Communication Networks a First Course", second edition, WCB McGraw-Hill, 1998

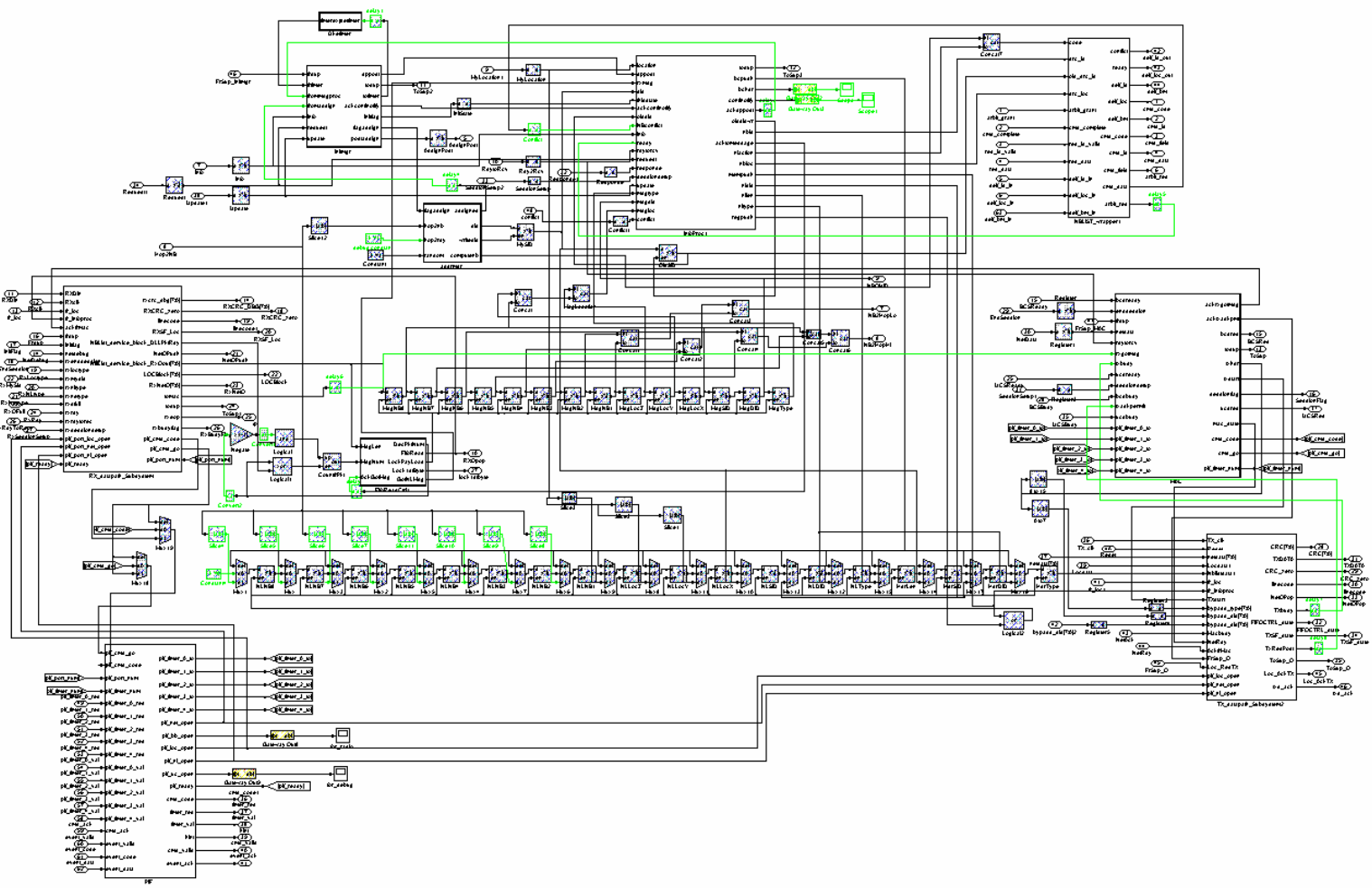
Appendix

Ten Simulink/Xilinx Block Diagrams Include:

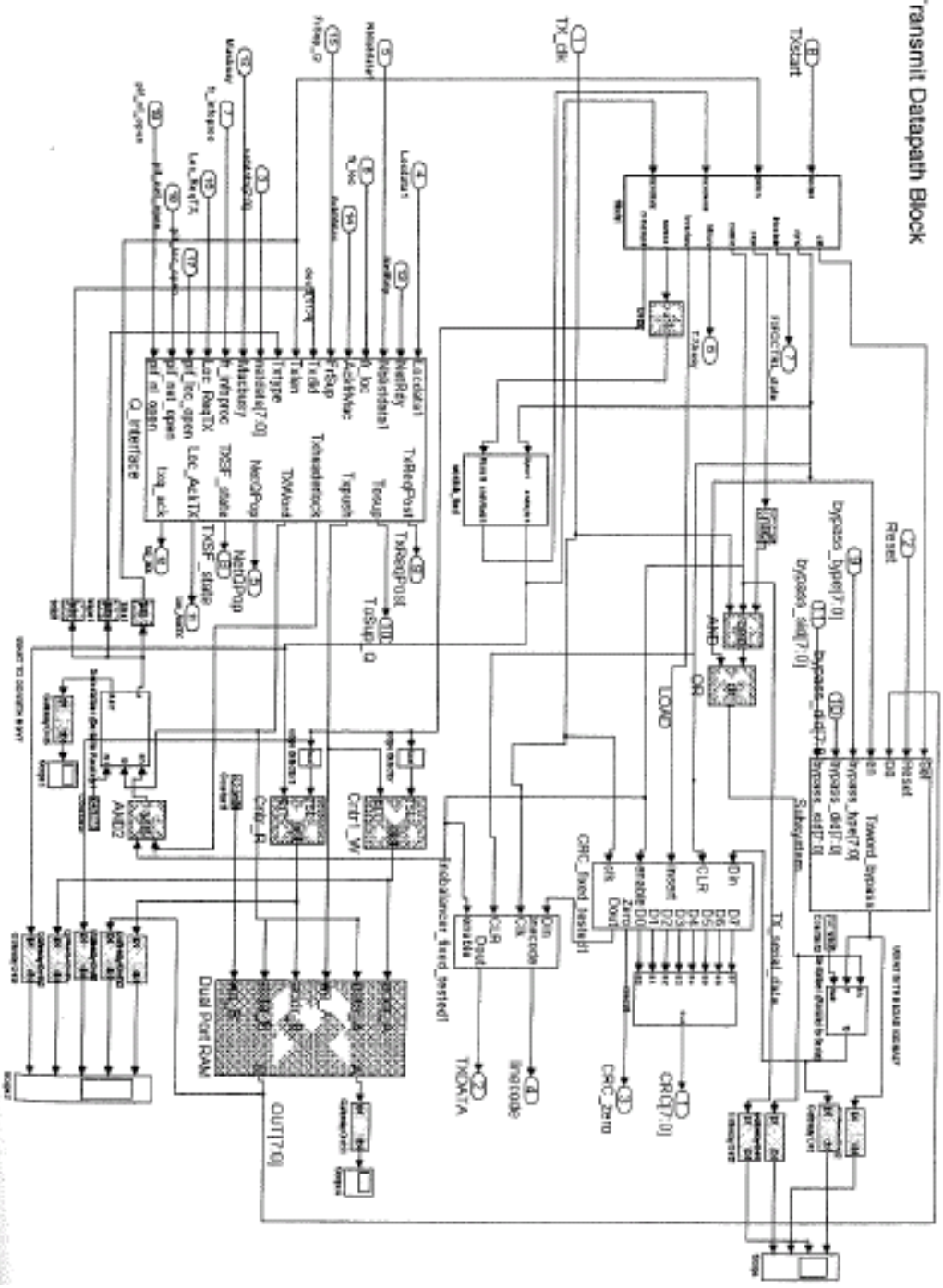
1. Data Link Layer Overview
2. Transmit Datapath Block
3. Transmit Datapath_Queue Interface Block
4. Transmit Datapath_CRC (Cyclic redundancy Code) Block (Same in RXD)
5. Transmit Datapath_Line Balancer Block (Same in RXD)
6. Transmit Datapath_Bypass Word Generator Block
7. Transmit Datapath_Word Clock Block (Same in RXD)
8. Receive Datapath Block
9. Receive Datapath_Queue Interface Block
10. Receive Datapath_Queue_Demux Block

Eight Stateflow Block Diagrams Include:

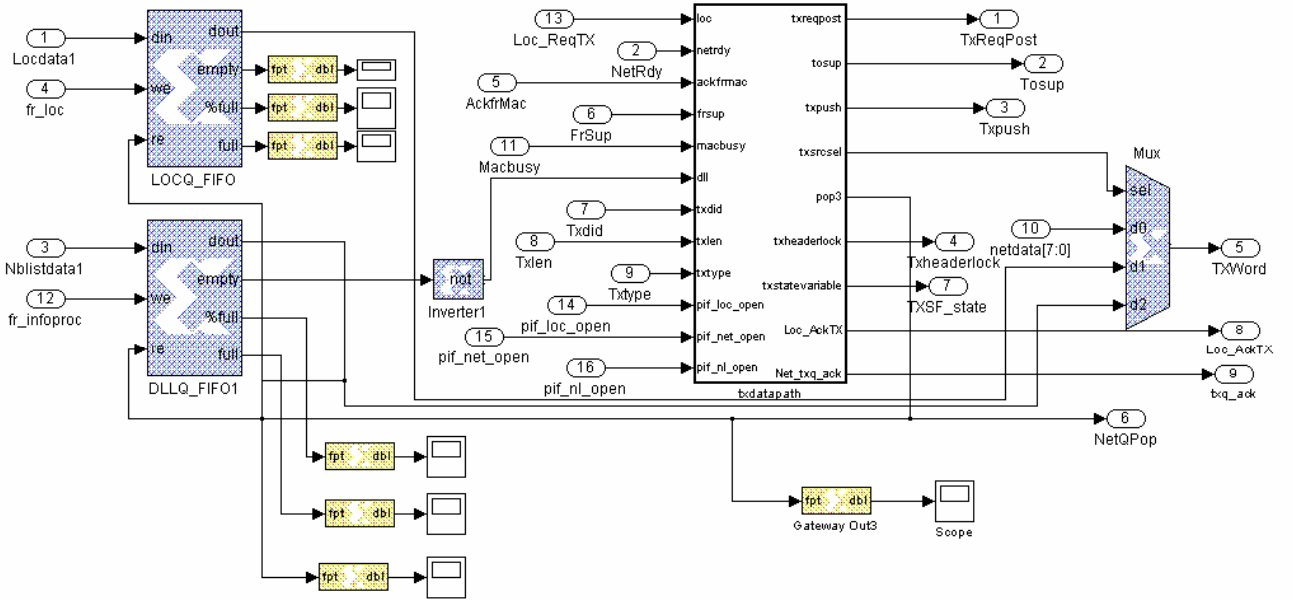
1. Media Access Control Block
2. Initialization Manager Block
3. Address Manager Block
4. Information Processing Block
5. Transmit Datapath Control Block
6. Receive Datapath Control Block
7. FIFO Control Block
8. FIFO Read Control Block



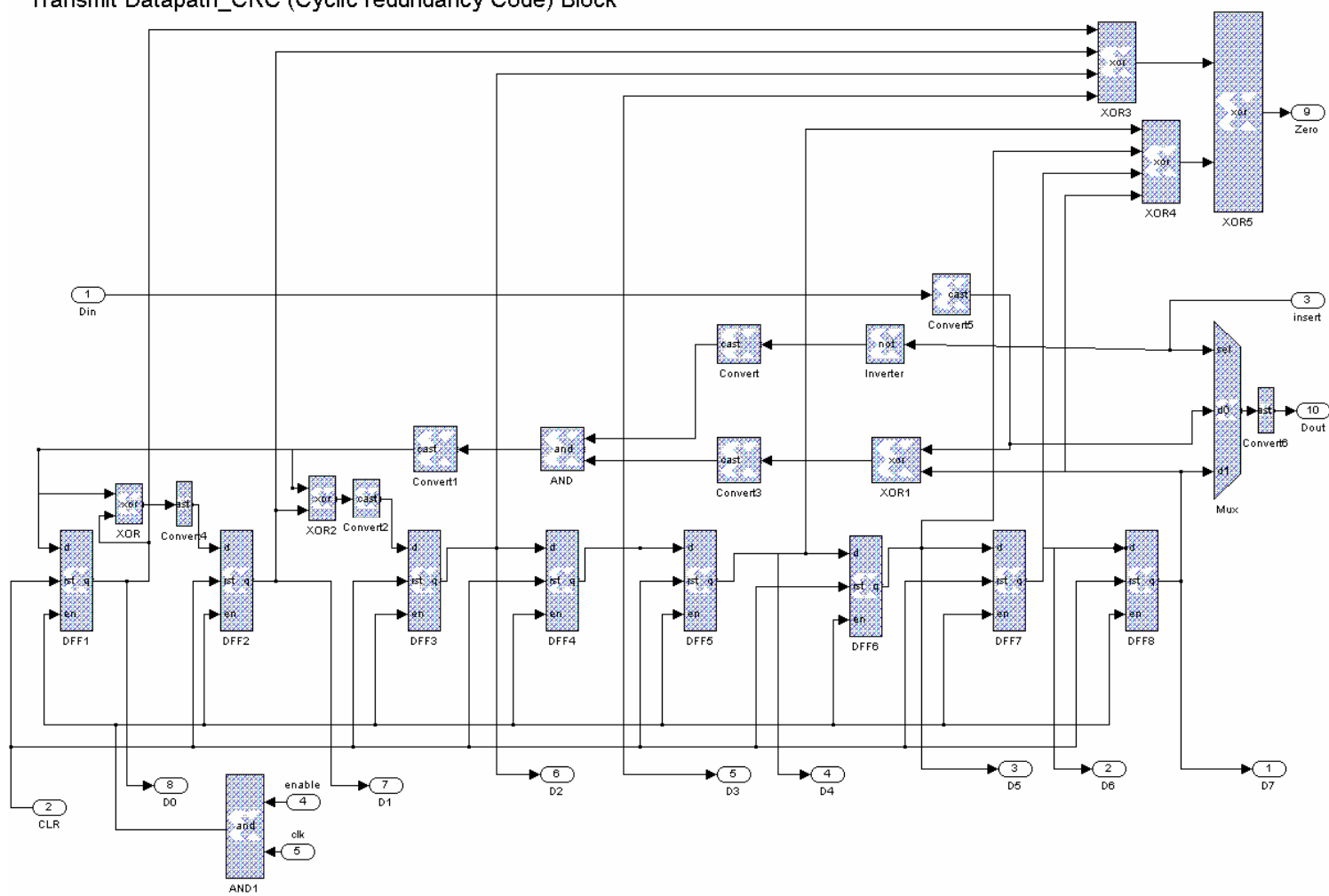
Transmit Datapath Block



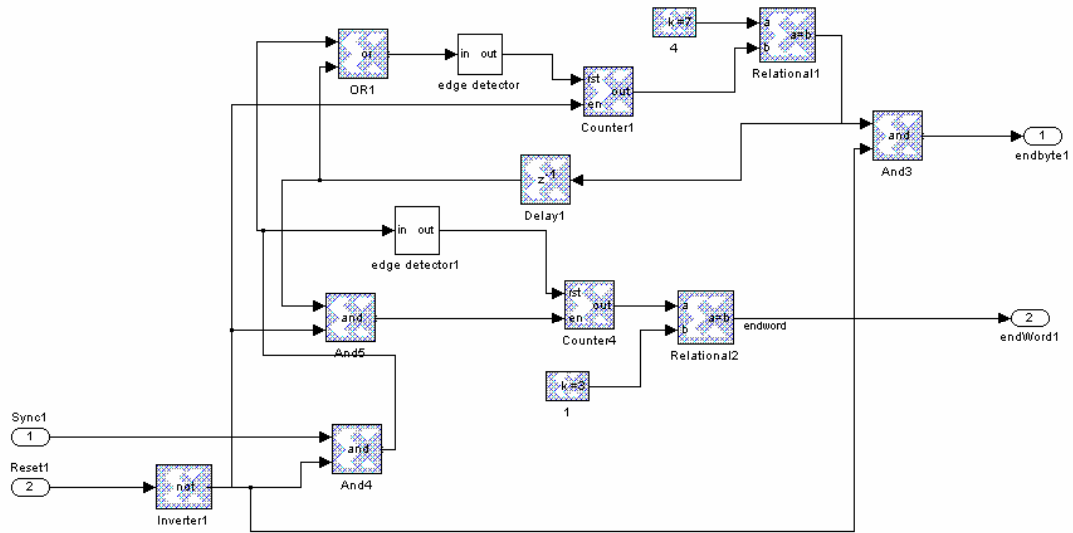
Transmit Datapath_Queue Interface Block



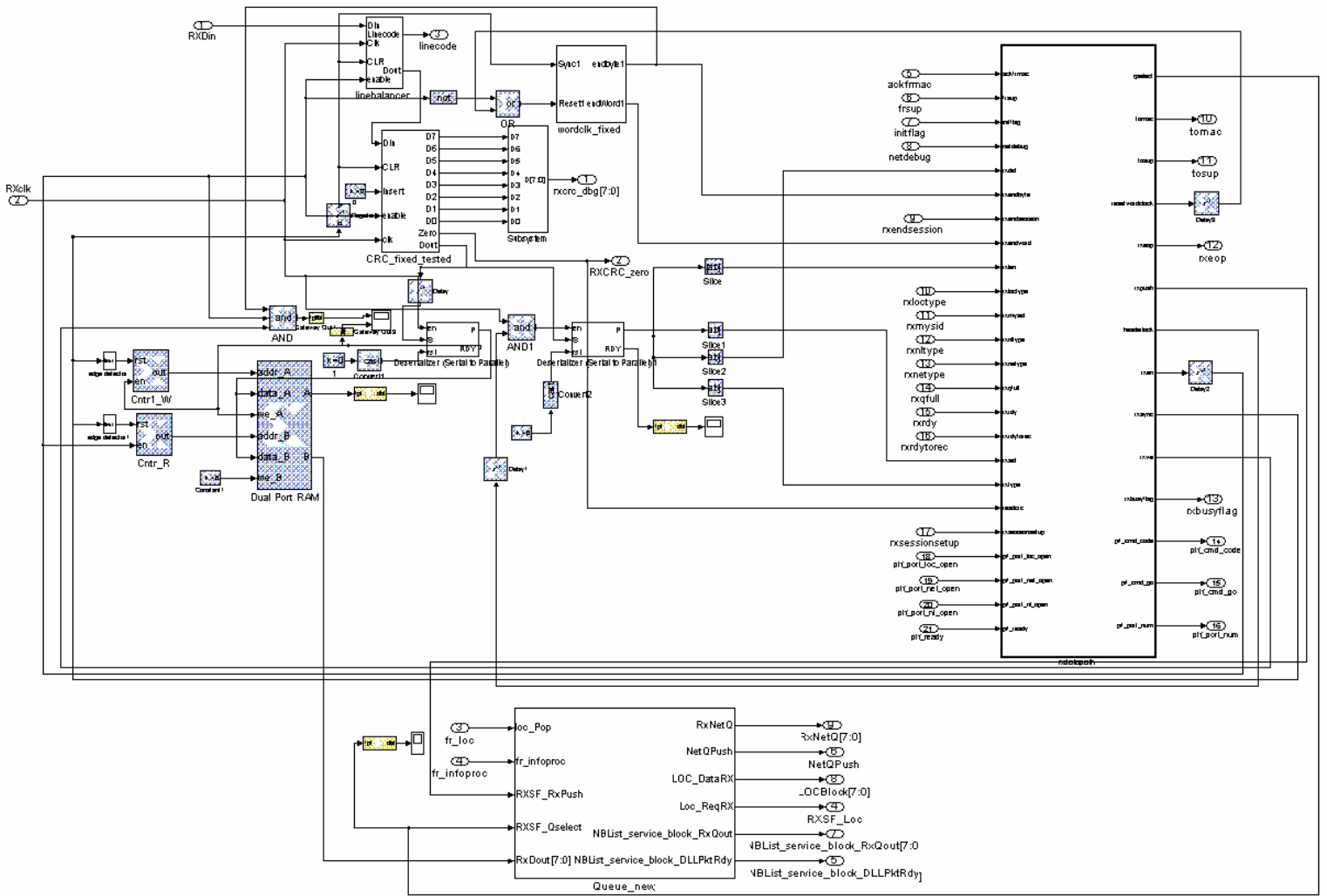
Transmit Datapath_CRC (Cyclic redundancy Code) Block



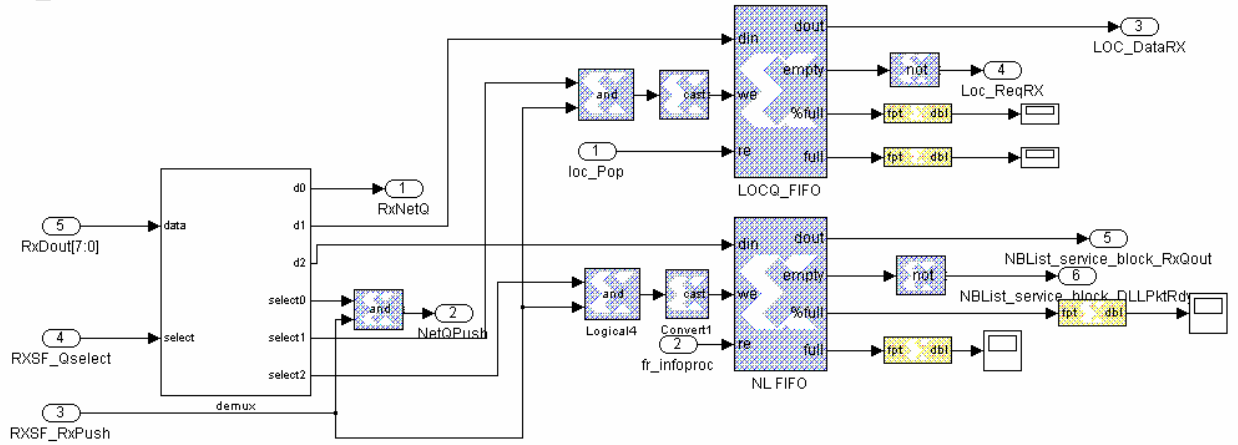
Transmit Datapath_Word Clock Block



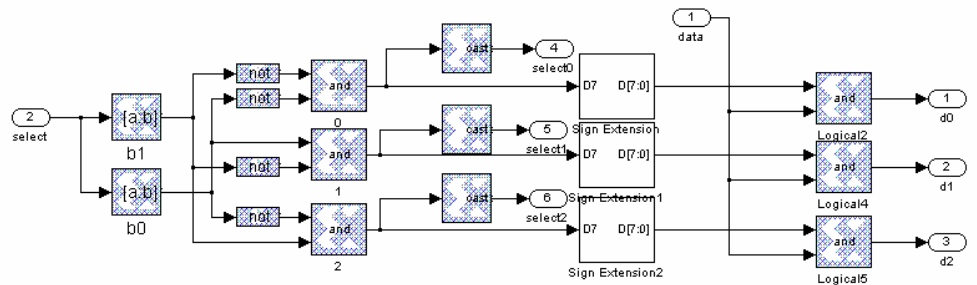
Receive Datapath Block



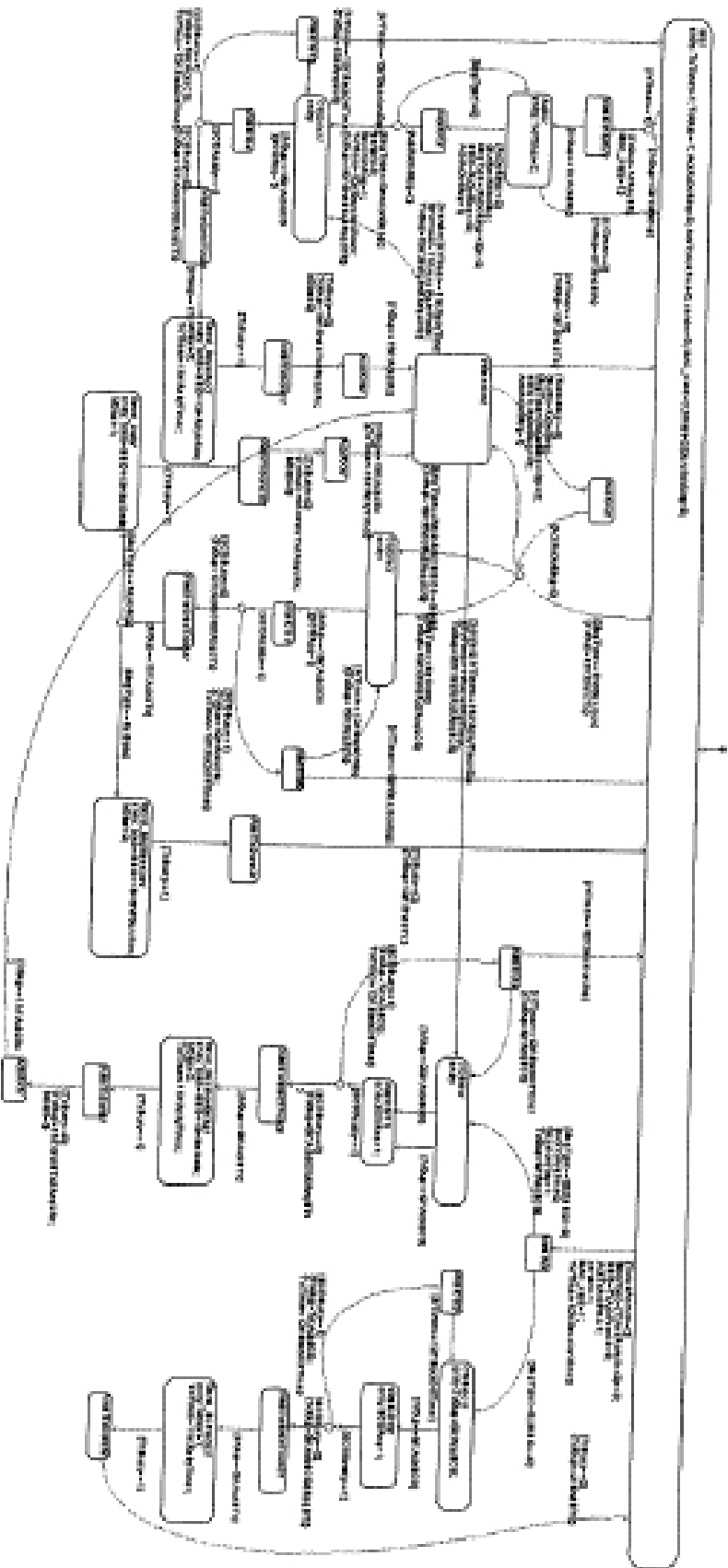
Receive Datapath_Queue Block



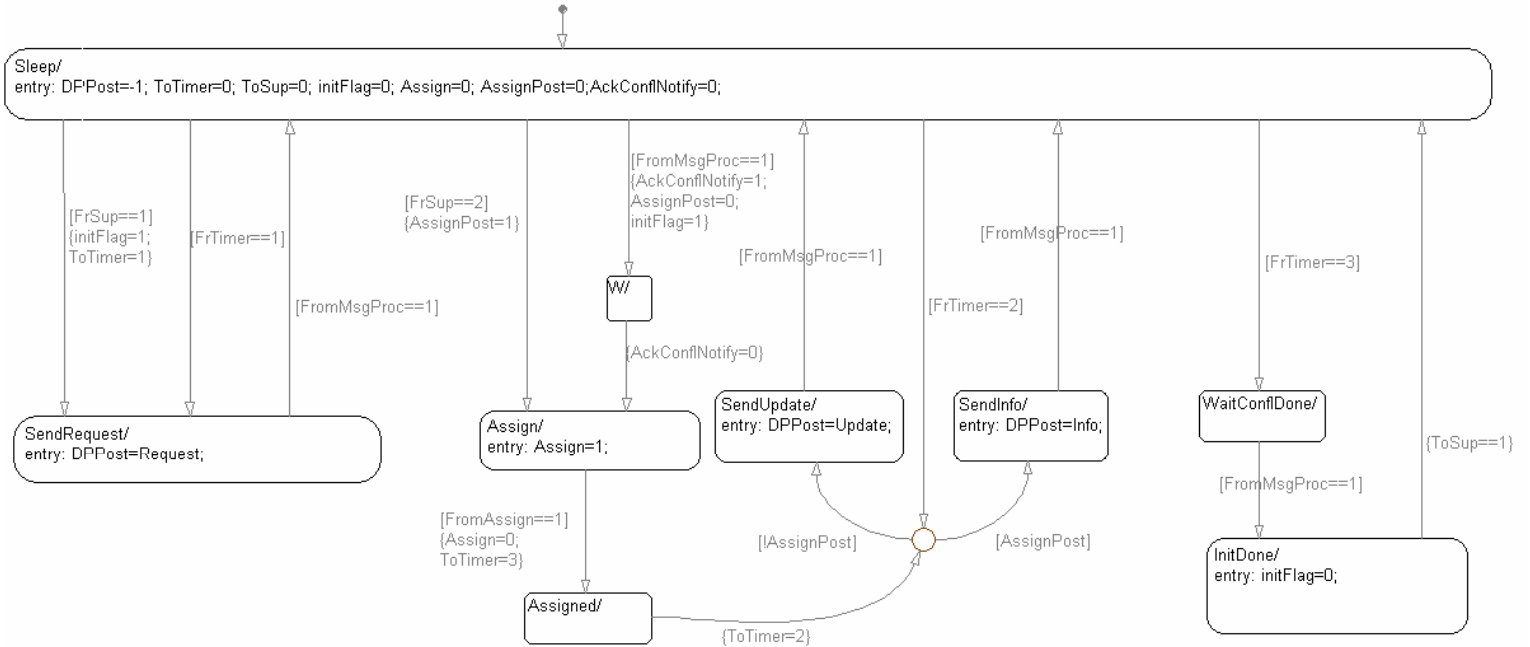
Receive Datapath_Queue_Demux Block



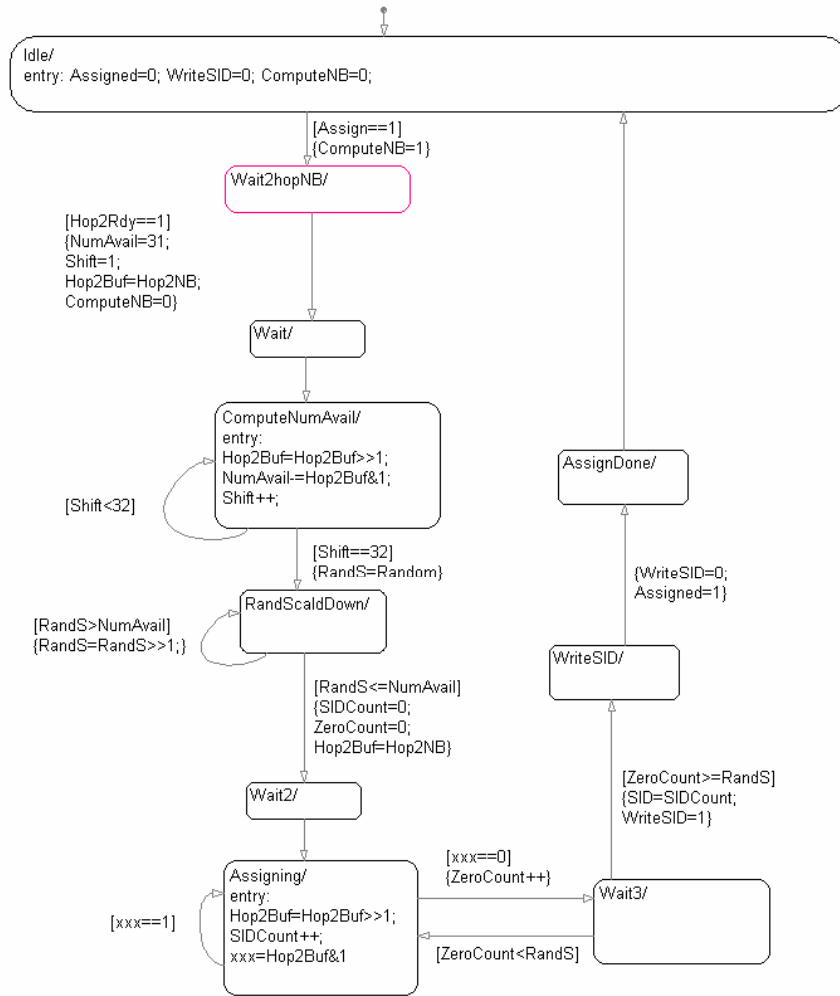
Media Access Control Block



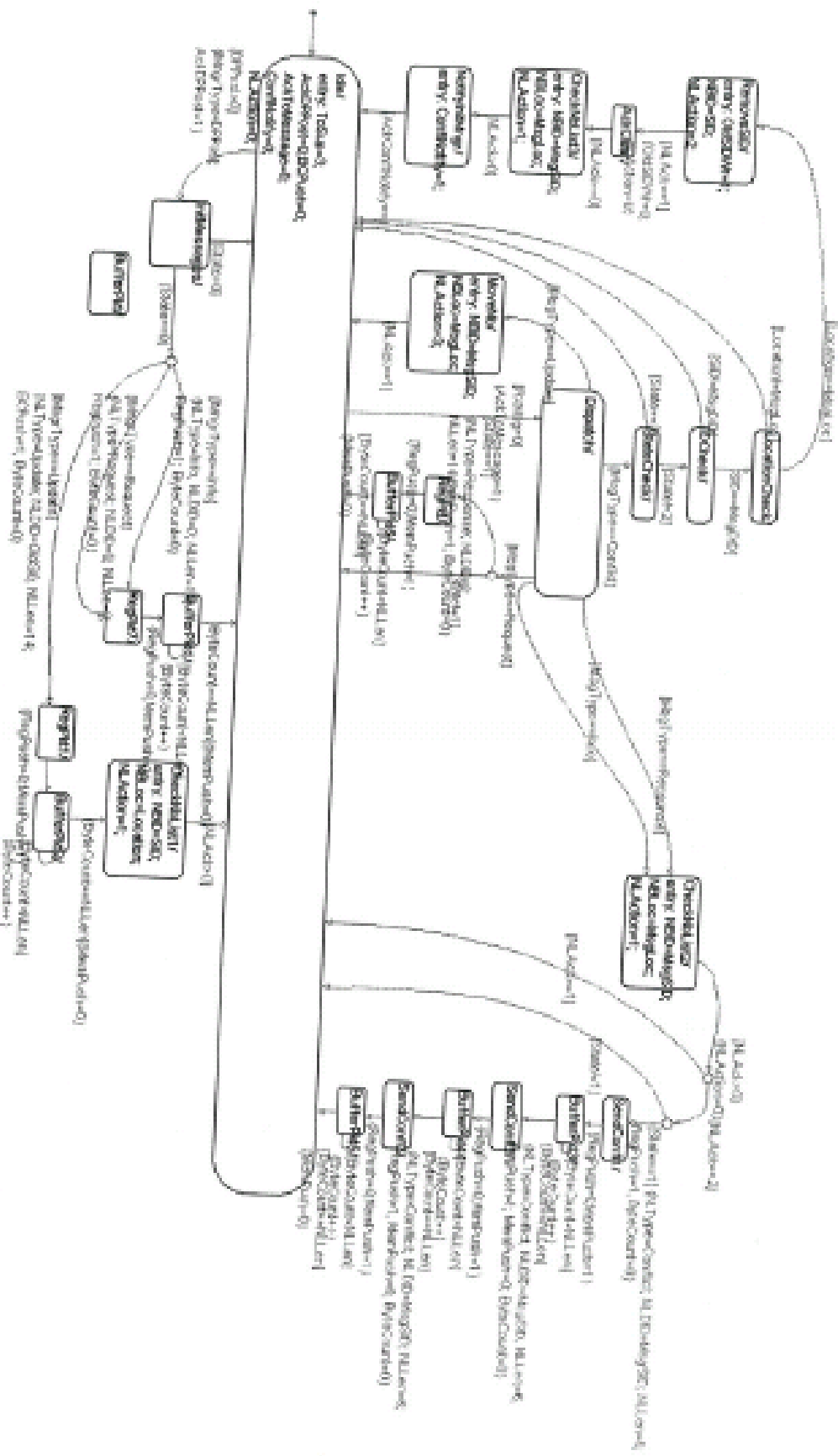
Initialization Manager Block



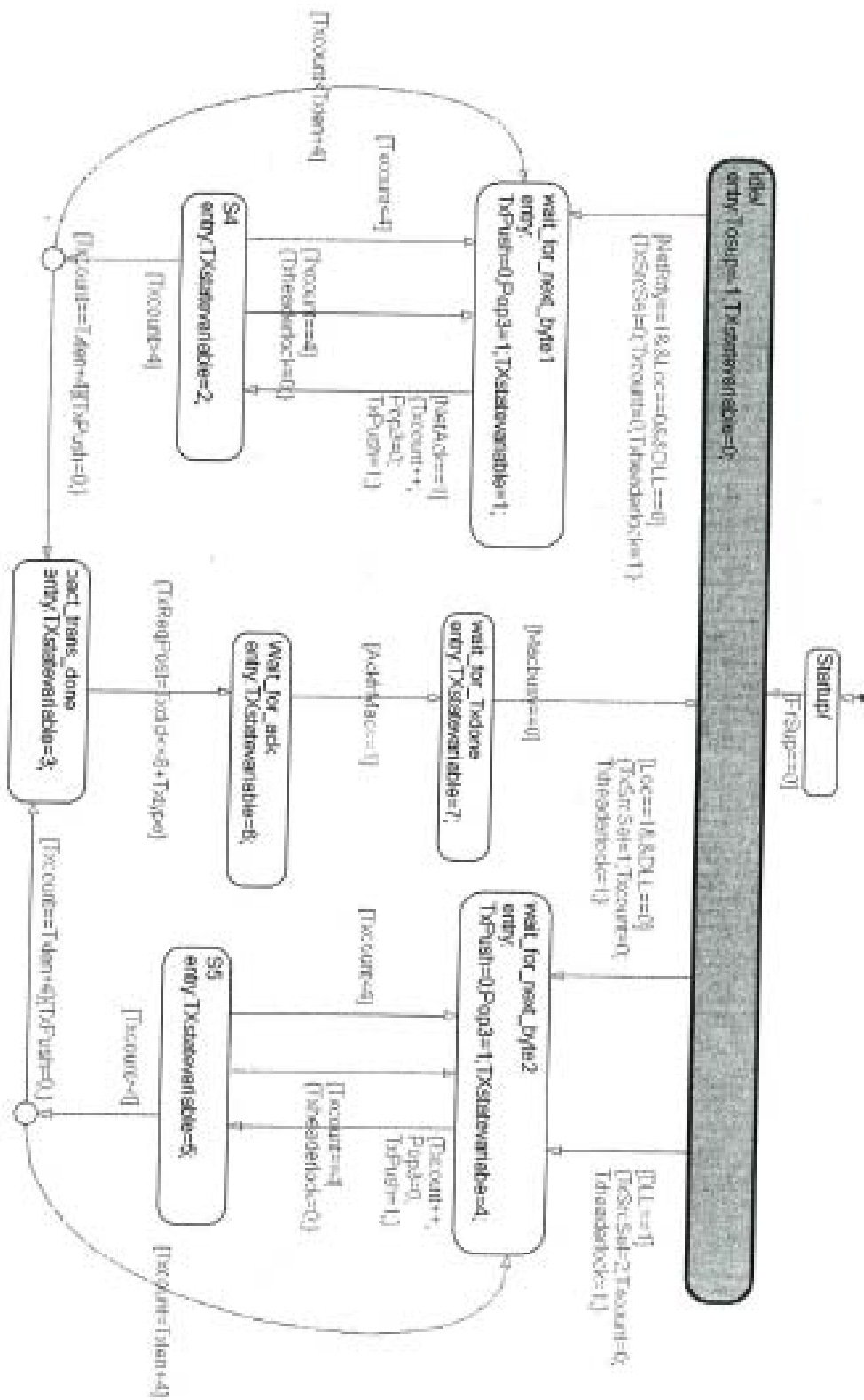
Address Manager Block



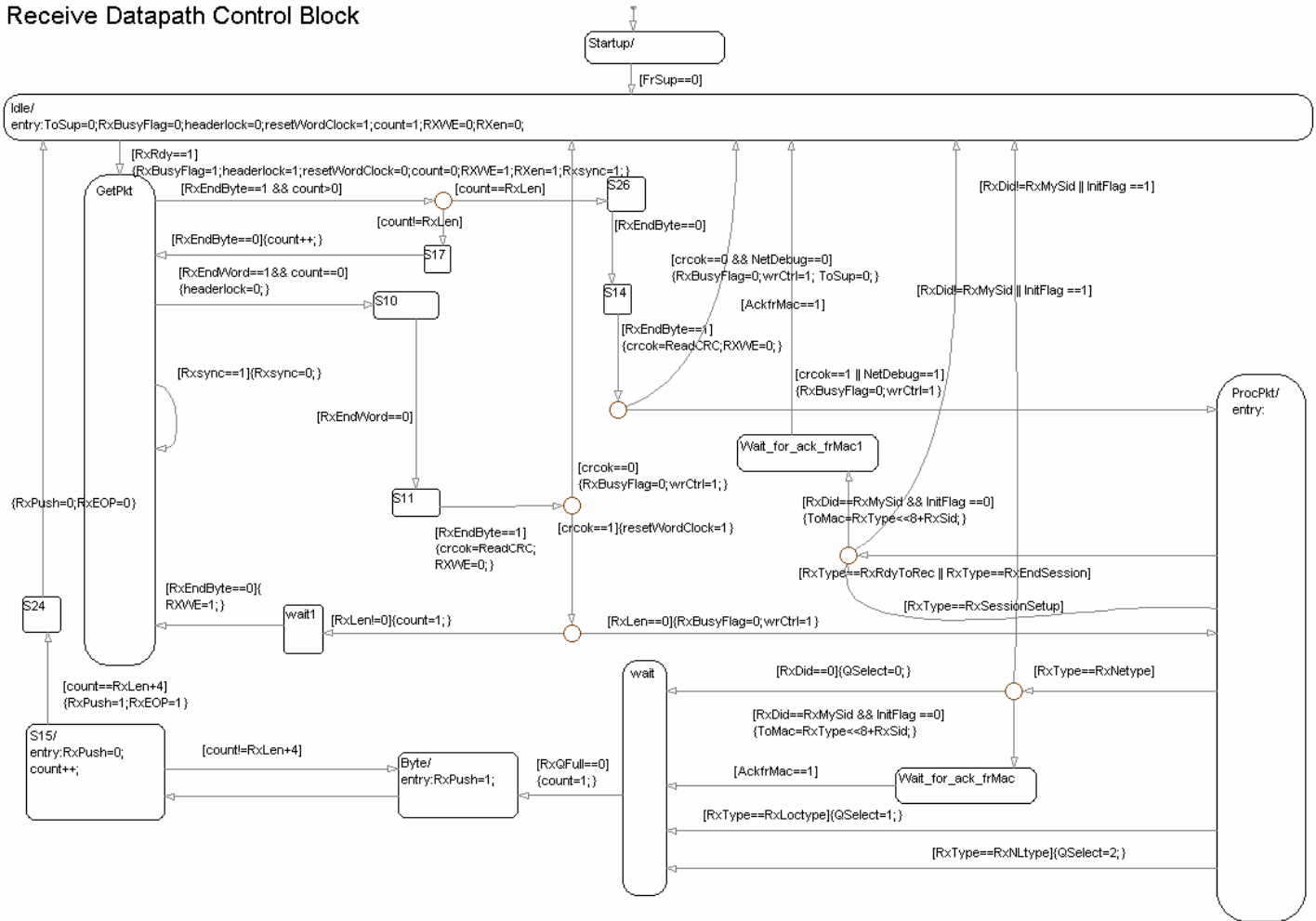
Information Processing Block



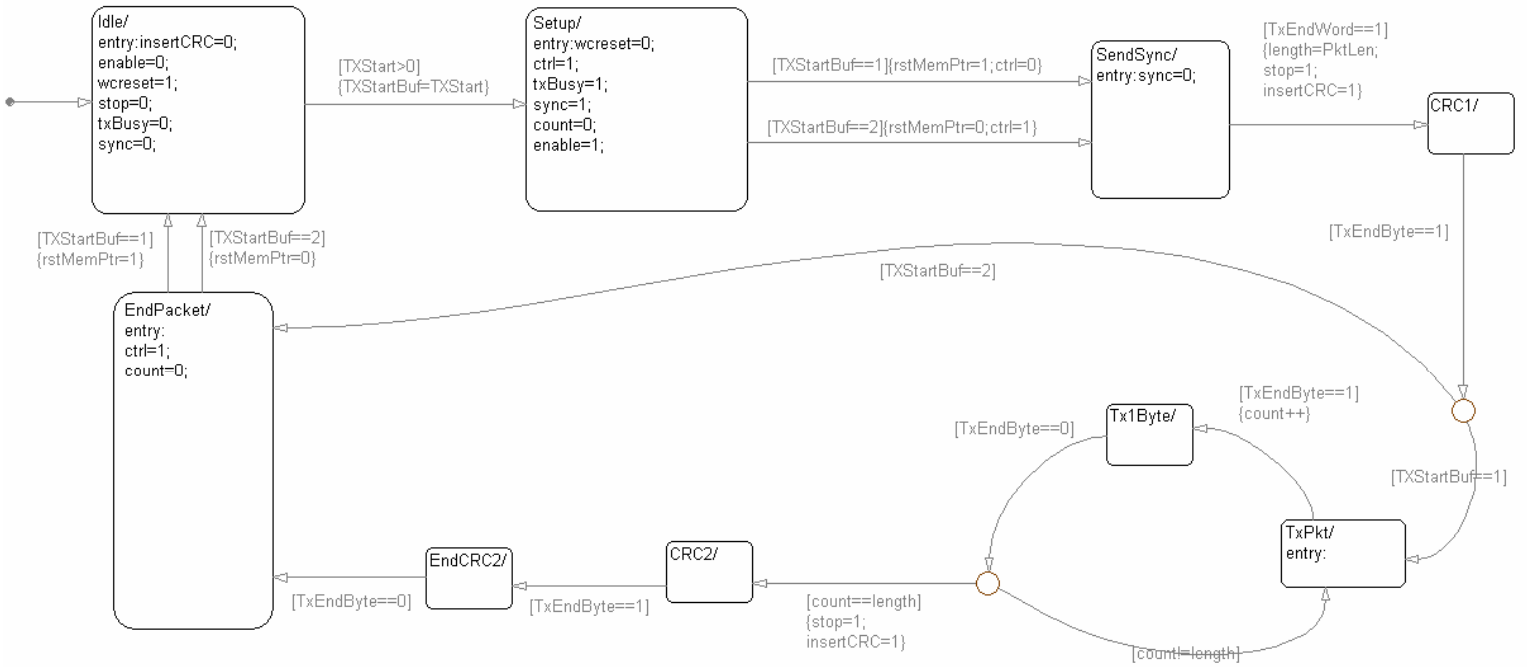
Transmit Datapath Control Block



Receive Datapath Control Block



FIFO Control Block



FIFO Read Control Block

