Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

THE DEVELOPMENT OF AN EMBEDDED WIRELESS MODEM CONFORMING TO ADVANCED MOBILE PHONE SYSTEM AND CELLULAR DIGITAL PACKET DATA STANDARDS

by
Xin Ren

Introduced by AT&T Bell Labs in 1970s, today Advanced Mobile Phone Systems (AMPS) - a first generation analog cellular system - serve millions of customers in the United States and 55 other countries, and the number of subscribers is still rapidly increasing. Although digital cellular systems are emerging in recent years, research efforts are still being made to the enhancement of AMPS systems because of their popularity. An example is the new Cellular Digital Packet Data (CDPD) standard announced in 1994. CDPD is an overlay of the existing AMPS, and has the capability of transmitting data packets over such an analog cellular network. This thesis intends to first introduce this new technology, and then discusses the design and implementation of an embedded wireless modem which conforms to the CDPD public standard. The emphasis is on software design, implementation, simulation, and test of the modem. The real-time test and demonstration shows that the newly developed modem and related software meet the requirements and specifications in terms of functionality, modularity, robustness, and low power consumption.

THE DEVELOPMENT OF AN EMBEDDED WIRELESS MODEM CONFORMING TO ADVANCED MOBILE PHONE SYSTEM AND CELLULAR DIGITAL PACKET DATA STANDARDS

by
Xin Ren

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Department of Electrical and Computer Engineering

January 1996

APPROVAL PAGE

THE DEVELOPMENT OF AN EMBEDDED WIRELESS MODEM COMFORMING TO ADVANCED MOBILE PHONE SYSTEM AND CELLULAR DIGITAL PACKET DATA STANDARDS

Xin Ren

Dr. MengChu Zhou, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. John Carpinelli, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. YunQing Shi, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Xin Ren

Degree: Master of Science

Date: January 1996

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering
  New Jersey Institute of Technology, Newark, NJ, 1996

- Bachelor of Science in Electrical Engineering
  Shanghai Jiao Tong University, Shanghai, China, 1992

Major: Electrical Engineering

Presentations and Publications:

Xin Ren and MengChu Zhou
"Tactical Scheduling for Rail Operations, A Petri-Net Approach"
Proc. of 1995 IEEE Int'l Conf. on Systems, Man & Cybernetics, pp. 3087-3092, Vancouver, Canada, December 1995

Xin Ren, Raafat Kamel, Gary Ellerbusch, John Carpinelli, Sol Rosenstark
"CoE 493: Computer Engineering Design Laboratory"
New Jersey Institute of Technology, 1995

To my mother

ACKNOWLEDGMENT

I would like to express my heart-felt appreciation to Dr. MengChu Zhou, who served as my supervisor, providing me with not only deep insight into difficulties encountered in the project, but also constant encouragement in writing this thesis. Special thanks are dedicated to Dr. John Carpinelli and Dr. YunQing Shi, who actively participating in my committee. Also, I would like to express my respect to Dr. Y. Bar-Ness, who found the funding through Anadigics Inc..

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Background

Today, telecommunications are undergoing fundamental changes, due largely to the booming fiber optics and mobile communication technologies. The cellular radio systems, which are concerned in this thesis, provide their subscribers with the opportunity to travel freely while simultaneously communicate with any other subscriber or any wireline telephone user. More sophisticated cellular systems can even provide services like fax, data transmission and electronic mail. In addition to these personal communication services, mobile technology has also proved its indispensableness in many other applications such as improving traffic safety and providing vital communication links during emergencies.

In the current cellular communication market, there exist at least seven different, partially incompatible standards, namely, North American AMPS (Advanced Mobile Phone System), Japanese MCS, Scandinavian NMT, British TACS, European GSM, Digital AMPS, and QUALCOMM CDMA. Among these standards, the North America AMPS is the oldest and most widely-used one. Introduced in 1970s by AT&T Bell Laboratories, it became the de facto technical standard for later systems such as the British TACS and the Japanese MCS. As a first generation analog system, AMPS is incorporated by some second generation digital systems such as IS-54. The latter specifies a dual-mode operation, which adds a digital voice transmission capacity to new subscriber equipment, while the analog AMPS remains the same. Due to its popularity,

research efforts are still being made to the enhancement of the existing AMPS system. The most recent enhancement is the CDPD (Cellular Digital Packet Data) standard, which is a major topic of this thesis.

1.2 Purposes of the Thesis

Introduced in early 1994, the emerging CDPD technology represents a low cost upgrade to the existing cellular voice infrastructure of AMPS, in the sense that it can provide packet data radio services over existing cellular networks. The CDPD standard utilizes the excess capacity left over by AMPS, and using the channel for packet data communications in the absence of voice traffic. Thus, the CDPD system can provide a broad range of wireless data and voice solutions for various communication needs - speech, electronic mail, fax and computer data transfer. For this reason, many electronics manufacturers are currently developing their product lines of wireless modems that conform to this standard.

Affiliated with a major IC manufacturer in New Jersey, the author has participated in a research and design project for such a modem. This thesis, however, is not intended to be just a detailed description of the hardware or software developed, but rather a discussion on the promising CDPD technology and the design philosophy involved.

1.3 Organization of the Thesis

Chapter 2 gives a brief description of the infrastructure of the North America AMPS and the CDPD network with emphasis on the latter. Chapter 3 and Chapter 4 present

the design philosophy and considerations from a theoretical standpoint. Some details of the software and hardware design are also discussed in these two chapters. As a supplement to the above chapters, Chapter 5 summarizes the results obtained and conclusions are drawn. In Appendix A and B, part of the source code for this modem is given, in either C language or assembly language.

CHAPTER 2

AMPS AND THE CDPD STANDARD

2.1 Overview of Cellular System

Conventional mobile telephone systems have many drawbacks: limited service capability, poor service performance, and insufficient frequency spectrum utilization, etc.. Basically, such a system selects one or more channels from a specific frequency allocation in a certain zone. The coverage area of each zone is normally planned to be as large as possible. A user who initiates a call in one zone is not guaranteed to be able to continue when he moves to an adjacent zone. The poor service performance of the conventional system is obvious. For example, in 1976, NYC had two conventional systems to serve a total of 545 users, with 3700 customers on a waiting list [3]. This number of subscribers has already created a high blocking probability during busy hours. The frequency spectrum utilization is not satisfactory, either.

Introduced as a concept in 1940s, cellular systems now have millions of users all over the world. Cellular systems have overcomed the major drawbacks of a conventional system. In a cellular system, continuation of conversation is guaranteed by the so-called hand-off capacity. Hand-off is a process which automatically changes the frequency while the user moves across zones, and this process is transparent to the user. Also, in a cellular system, because of the new concept of frequency reuse, the frequency utilization has been improved to a large extent.

The spectral efficiency of cellular systems was shown at the beginning of 1970s. In 1979, FCC authorized Illinois Bell Telephone Co. (IBT) to operate the first cellular system in the Chicago area, and thus commenced a new era of wireless communications.

The infrastructure of a cellular system is illustrated in Figure 2.1. Theoretically, the coverage of one "cell" should be a circular area. However, for the sake of mathematical

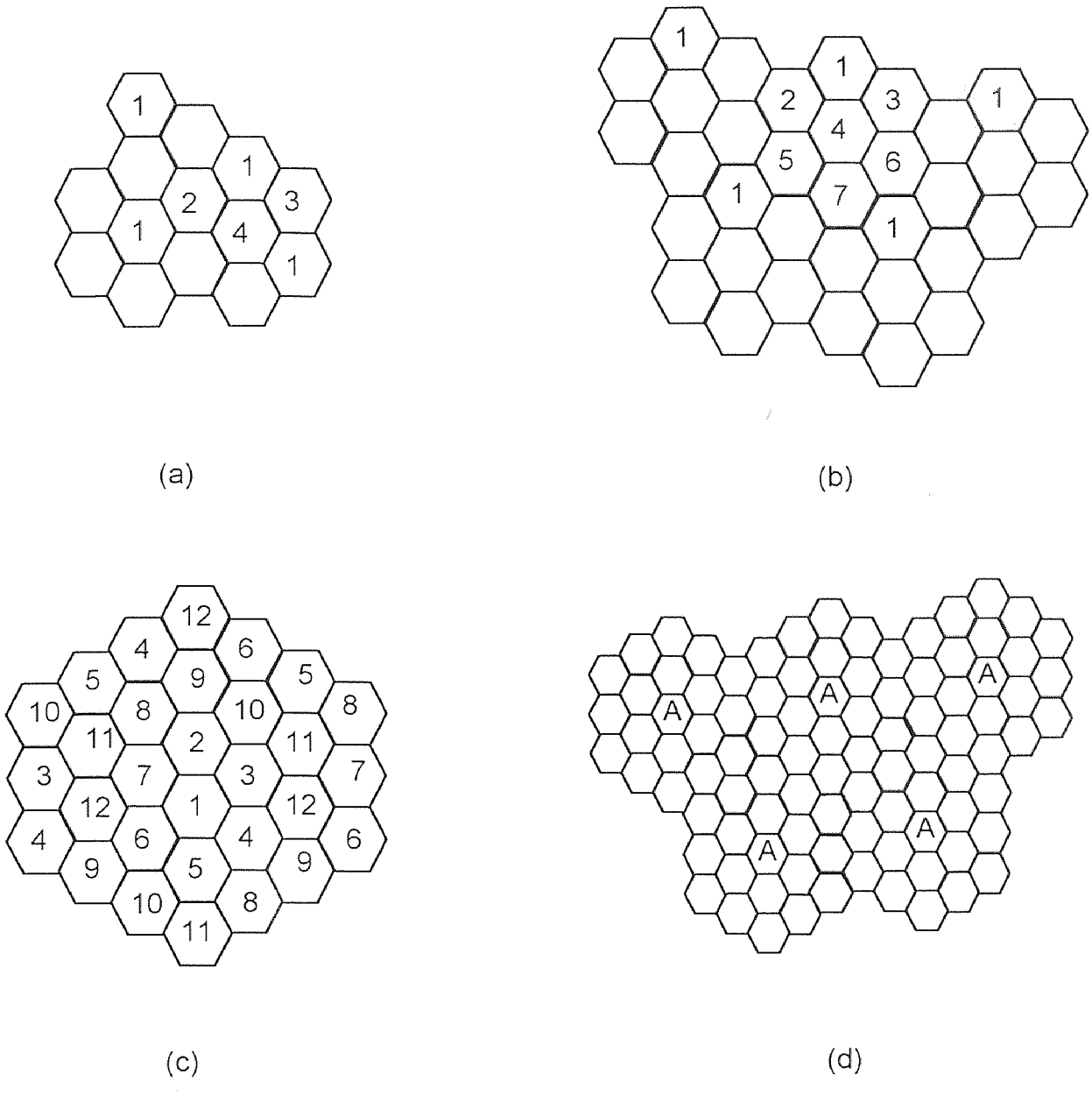

Figure 2.1 Cell Reuse Pattern in a Cellular Network. (a) K=4. (b) K=7. (c) K=12. (d) K=19.

as well as visual simplicity, a hexagon shape is assumed, and the discrepancy is negligible.

2.2 The Advanced Mobile Phone System (AMPS)

2.2.1 Historical Notes

Introduced in 1970s by AT&T Bell Laboratories, AMPS served approximately 3.5 million customers as of the end of 1989, and the number of subscribers is still rapidly increasing. According to many market forecasts, AMPS will accommodate 15-20 million users at the end of this century [2].

AMPS employs an FM-FDMA modulation technique, and a signaling bit rate of 10kb/s [1]. Compared to the huge traffic load, the frequency allocation for AMPS is rather limited. In 1974, FCC authorized a 20 MHz band for cellular systems. Recently, additional bandwidth is allocated, but there is still no substantial improvement. This spectrum is used by 666 channels in AMPS, where each channel has a bandwidth of 30 kHz. The inefficiency of AMPS (a first generation cellular system) led to the development of the second generation (GSM, ADC, JDC) and the third generation cellular (CDMA) systems. However, owing to its popularity, research efforts are still being made to AMPS in order to increase its capacity and enhance its performance.

2.2.2 The EIA/TIA 553 Standard for AMPS

The EIA/TIA 553 standard is actually a set of compatibility requirements for cellular mobile telecommunication systems. Its purpose is to ensure that a mobile station can obtain service anywhere in an AMPS cellular system. In this section, instead of discussing the standard in detail (which could be extremely lengthy), we try to catch its essence, and explain no more than necessary to understand the successive discussions.

In technical standards, a list of definitions of terms is usually given before any further discussion. Following this convention, we give some of the important ones as follows [5]:

Land Station. A station in the Domestic Public Cellular Radio Telecommunications Service, other than a mobile station, used for radio communications with mobile stations.

Mobile Station. A station in the Domestic Public Cellular Radio Telecommunications Service intended to be used while in motion or during halts at unspecified points. It is assumed that mobile stations include portable units (e.g., hand-held "personal" units) as well as units installed in vehicles.

Control Channel. A channel used for the transmission of digital control information from a land station to a mobile station (Forward Control Channel) or from a mobile station to a land station (Reverse Control Channel).

Voice Channel. A channel on which a voice conversation occurs and on which brief digital messages may be sent from a land station to a mobile station (Forward Voice Channel) or from a mobile station to a land station (Reverse Voice Channel).

Access Channel. A control channel used by a mobile station to access a system to obtain service.

Paging Channel. A forward control channel that is used to page mobile stations and send orders.

Mobile Identification Number (MIN). The 34-bit number that is a digital representation of the 10-digit directory telephone number assigned to a mobile station.

System Identification (SID). A digital identification associated with a cellular system; each system is assigned a unique number.

Supervisory Audio Tone (SAT). One of three tones in the 6-kilohertz region, which are transmitted by a land station and transponded by a mobile station.

Our discussion concentrates on mobile station compatibility requirements, for the wireless modem is part of a mobile station. Moreover, the characteristics of the physical link, including transmitter and receiver's frequency parameters, power control, and modulation techniques are not discussed here, because these lower-level tasks are accomplished by existing IC products. Instead, we are much more concerned about *call processing*, which describes mobile station's operations as controlled by a land station.

The call processing of AMPS can be depicted as a state machine with four states: *initialization*, *idle*, *system access*, and *conversation*, as shown in Figure 2.2.

Once power is applied to a mobile station, it should enter the *initialization* state in which it retrieves system parameters and selects a paging channel with enough signal strength. After *initialization*, the mobile station enters the *idle* task, in which it monitors the control messages for orders, or a user can initiate a call. In either case, the mobile station should enter the *system access* state. In the *system access* state, the mobile station scans and tries to seize a reverse control channel. If succeeded, it then uses this reverse control channel to communicate with base station and enters the final state *conversation*. Otherwise, the station returns to its *initialization* state.

2.3 The Cellular Digital Packet Data (CDPD) Standard

2.3.1 Historical Notes

The Cellular Digital Packet Data is a fast, inexpensive and effective way to transmit data packets over the existing AMPS cellular network. This very new standard (announced in 1994) is promoted by a consortium of Ameritech Mobile Communications, Inc., Bell Altantic Mobile Systems, Contel Cellular, Inc., GTE Mobile Communications, Inc., NYNEX Mobile Communications, Inc., PacTel Cellular, Southwestern Bell Mobile Systems, and McCaw Cellular Communications, Inc..

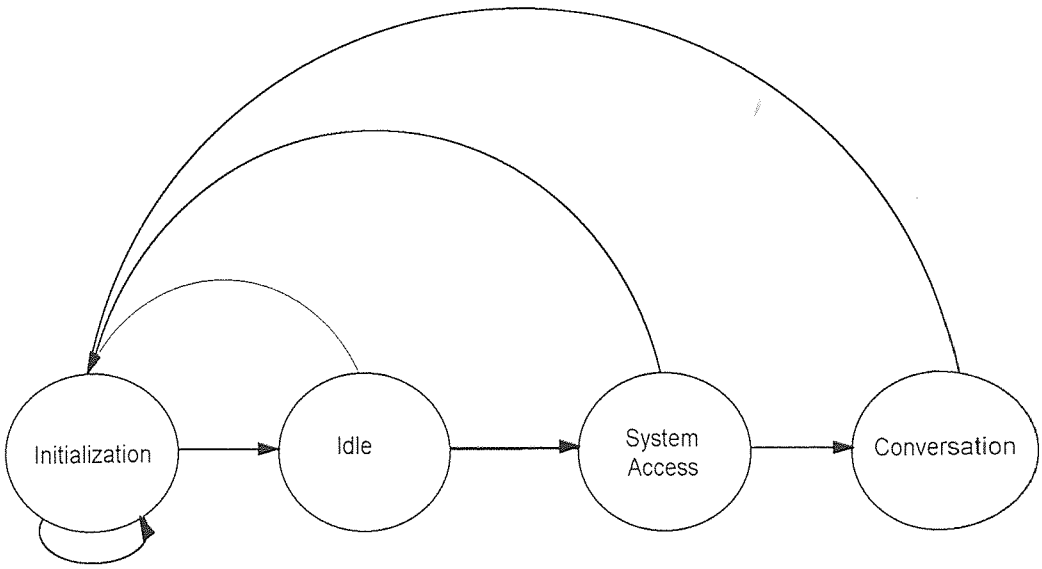

Figure 2.2 State Machine Representation of an AMPS Mobile Station.

CDPD uses existing cellular infrastructure, technologies, and even the same frequency spectrum allocated to AMPS to transmit data at the rate of 19.2 kbps. Indeed, it can be viewed as a cellular telephone network overlay by adding a small amount of CDPD equipment to existing cell cites. Cell operators are able to provide both voice and data on the same frequency. Figure 2.3a shows infrastructure of the CDPD network.

According to the research into AMPS cellular systems, 30% or more air time, even during heavy traffic, is unused [4]. CDPD technology can detect and utilize these otherwise wasted idle moments to transmit packet data, therefore enhance the efficiency of a cellular channel (see Figure 2.3b). This characteristic makes the CDPD standard the most cost-effective internet extension for the cellular carriers. There is also good news for the CDPD users: due to its packet data nature, users are charged only for the amount of data transferred, not for the amount of time used.

When this standard first appeared in 1994, it was predicted that it would occupy the market in a very short time. Now this prediction turns out to be too optimistic, as CDPD is still unavailable in most places. However, with the millions of AMPS users across the US and 55 other countries, there is no doubt that CDPD will be offered at affordable cost in the near future.

2.3.2 The CDPD Public Standard

The CDPD public standard implements the protocols of network layer (layer 3) and below of the OSI (Open System Interconnection) model, which means CDPD network can easily inter-operate with existing data communication networks, where users have the flexibility to employ applications over the remaining four layers. Subscribers of a CDPD system initiate the communication via a Mobile End System (M-ES). M-ES sends data packets to the Mobile Data Base Station (MDBS), employing GMSK filtering and FM modulation. Mobile Data Base Stations are at the same locations as the

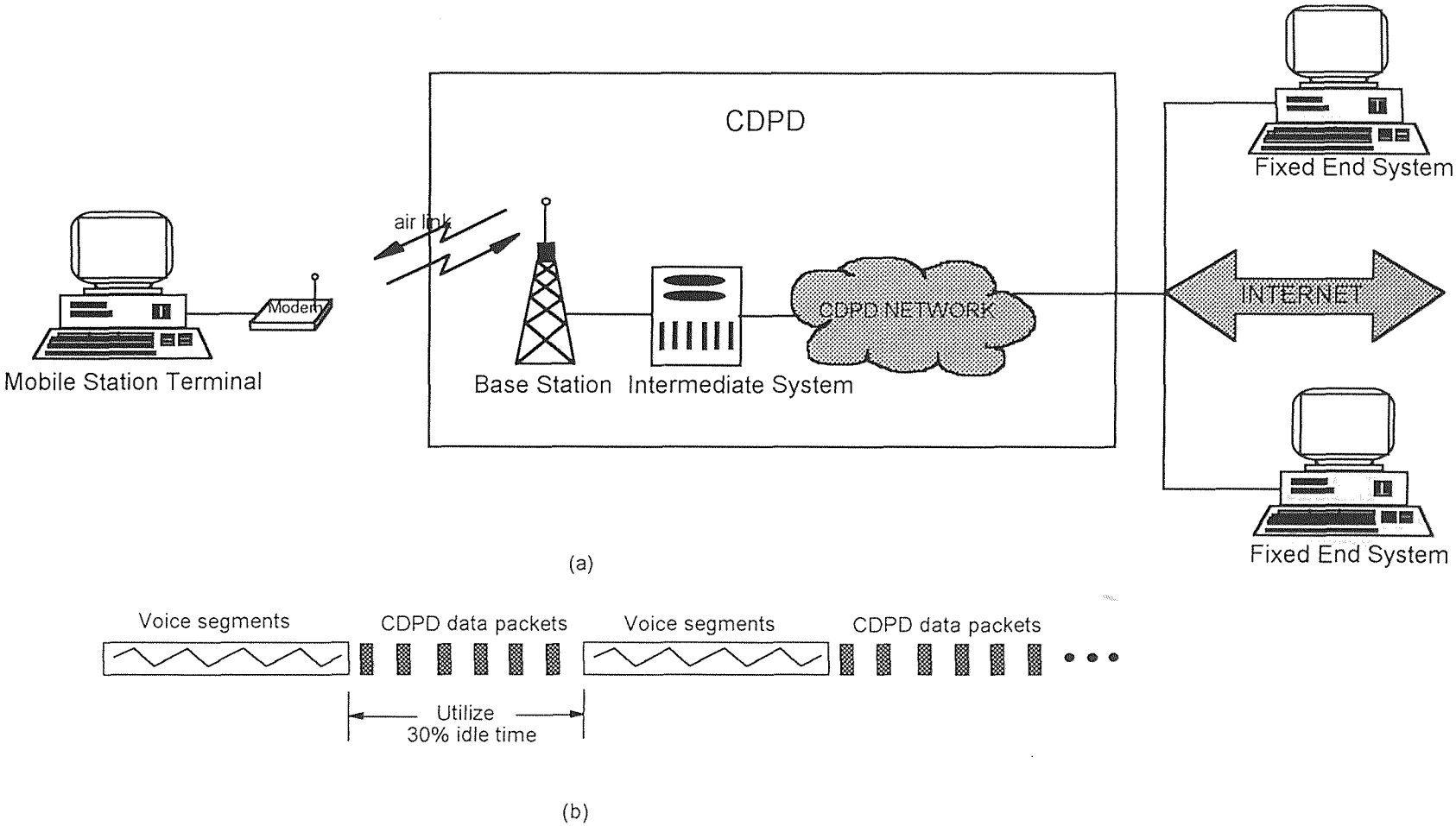

Figure 2.3 Infrastructure of the CDPD Network. (a) Infrastructure. (b)Time Utilization.

base stations of AMPS, but use different communication equipment. The MDBS is hardwired to a local Mobile Data Intermediate System (MD-IS), where the state-of-the-art network management is employed. A MD-IS is a *node* in a CDPD network. Usually there is a MD-IS for several cells. Data packets are then forwarded to their destination by MD-IS.

The destinations for data packets are either F-ES (Fixed End Systems) or other M-ES's. Generally, F-ES refers to fixed computer systems with internet connection. These interconnections are shown in Figure 2.4.

The CDPD standard implements the Network Layer (Layer 3) with Internet Protocol (IP) and Connectionless Network Protocol (CLNP). Data Link Control (DLC) Layer (Layer 2) is implemented using the Mobile Data Link Protocol (MDLP), where functions like framing/deframing packet segments, and appending/removing frame headers are executed. Below this is the Medium Access Control (MAC) Sublayer, which inserts/deletes flags, blocks/unblocks frames with Reed-Solomon FEC encoding/decoding, and scrambles/unscrambles the data stream with a Pseudorandom Noise sequence. For the lowest layer, Physical Layer (Layer 1), the CDPD standard uses FM modulation and GMSK filtering with a time-bandwidth product of 0.5, which is introduced to limit the frequency spectrum to the required channel bandwidth (30kHz) in AMPS.

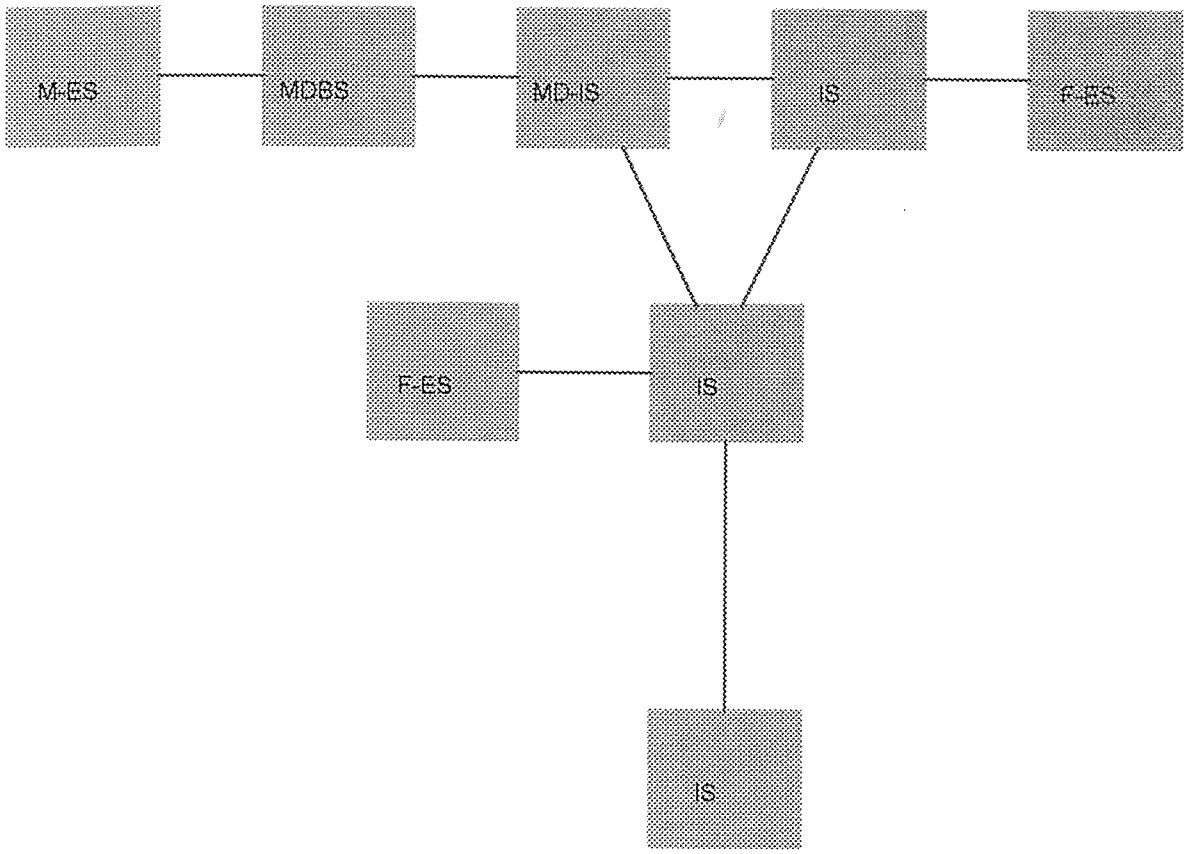

Figure 2.4 CDPD Reference Architecture

CHAPTER 3

HARDWARE IMPLEMENTATION OF THE EMBEDDED WIRELESS MODEM

3.1 Overview

It is well-known that digital signals are not suitable for transmission over airlinks or telephone lines, due largely to the distortions that occur in the course of transmission. A conversion device, collectively known as *modem*, or modulator-demodulator, is used to carry digital signals on an analog transmission medium.

Modems come with different configurations and flavors, and can be categorized according to several criteria. Based on the methods of fabrication, a modem can be categorized as stand-alone, fabricated on adapter cards, or fabricated on rack-mount cards. Based on the type of data transmitted, they fall into two categories: modems dealing with synchronous data transmission, and ones dealing with asynchronous data streams.

The most frequently used categorization, however, devides modems into intelligent and dumb or non-intelligent ones. Until 1970s, most modems were dumb devices, in the sense that they could only perform certain limited functions preset by the manufacturers. Influenced by the booming VLSI technology, microcontrollers were introduced in modem design. Systems that incorporate microcontrollers are known as *embedded systems*, and a modem that contains a microcontroller is by definition an embedded system. An embedded modem is also referred to as *intelligent modem*. Intelligent modems can perform a wide range of functions - such as automatic redialing, negotiation of modulation method, error detection and correction, and many more - based on requests initiated by users.

The wireless modem discussed in this thesis has an embedded Intel 8051 family microprocessor, which is the central control unit of the modem. Together with other

components, the modem provides both sets of functionality required by AMPS and the CDPD standard.

The design philosophy varies as each kind of modems has to be fit to a different application. In the following sections, we will address issues involved in design process of our particular embedded modem.

3.2 Design Philosophy and Considerations

The key requirements for a wireless modem, used in a hand-held portable cellular set, are:

- Small physical size,
- high reliability,
- Low power consumption,
- Low cost.

By carefully selecting the chip set, we can achieve acceptable physical size. As will be discussed shortly, the specialized cellular radio chips made by Philips and Rockwell meet the requirement very well.

There are two different approaches to achieving high reliability. One is in the hardware sense, and the other is from a software standpoint. For hardware design, fewer interconnections between hardware components generally mean higher reliability. The I^2C bus, an industry standard first introduced by Philips, gives a simple yet effective solution to the problem. From a software standpoint, a robust software design can render high reliability. Here, robustness means that all the software modules are stable and can regain normal operation shortly after something goes wrong. Unexpected states are likely to be entered in real-world applications, as in the case that an erroneous command packet is received.

Low power consumption is crucial for any cellular communications system, and it is an important factor to be considered in selecting a proper modulation method.

However, what concerns us here is what we can do in our hardware and software design to reduce the power consumption. Choosing low current consumption hardware components certainly helps; but more importantly, a software implementation that puts the microcontroller and other components into sleep or power-down mode whenever possible, gives us a more satisfactory low level of power consumption.

As will be discussed later, the design philosophy described above is successfully applied to our hardware and software implementation.

3.3 Structure of the Wireless Modem

A system schematic for the AMPS/CDPD embedded wireless modem is shown in Figure 3.1. As can be seen from the figure, the radio section of the modem is composed of a transmitter, a receiver, an RSSI (Received Signal Strength Indicator) A/D converter, a power level control D/A converter, and two phase locked loops. The RF receiver receives the incoming RF signal, downconverts and translates it into the first IF, then this first IF is further downconverted and demodulated into the desired audio/data and RSSI signals. The Rx-PLL (Receiver Phase Locked Loop) provides the correct phase as needed. The transmitter and Tx-PLL (Transmitter Phase Locked Loop) do the same thing, except in the opposite direction. The analog RSSI is converted into digital signal, used by the microcontroller for control purposes. The power level control signals issued by the microcontroller are converted to analog signals for use by the transmitter.

The baseband section is composed of an Audio Processor (APROC) and a Digital Processor (DPROC), both made by Philips. APROC consists of two chips: SA5752 and SA5753, which provide companding, VOX, filtering and other control functions. The DPROC is actually the UMA1000LT data processor, and incorporates all the data transceiving, processing and SAT functions. APROC and DPROC together fulfill a large part of the functionality requirements for AMPS.

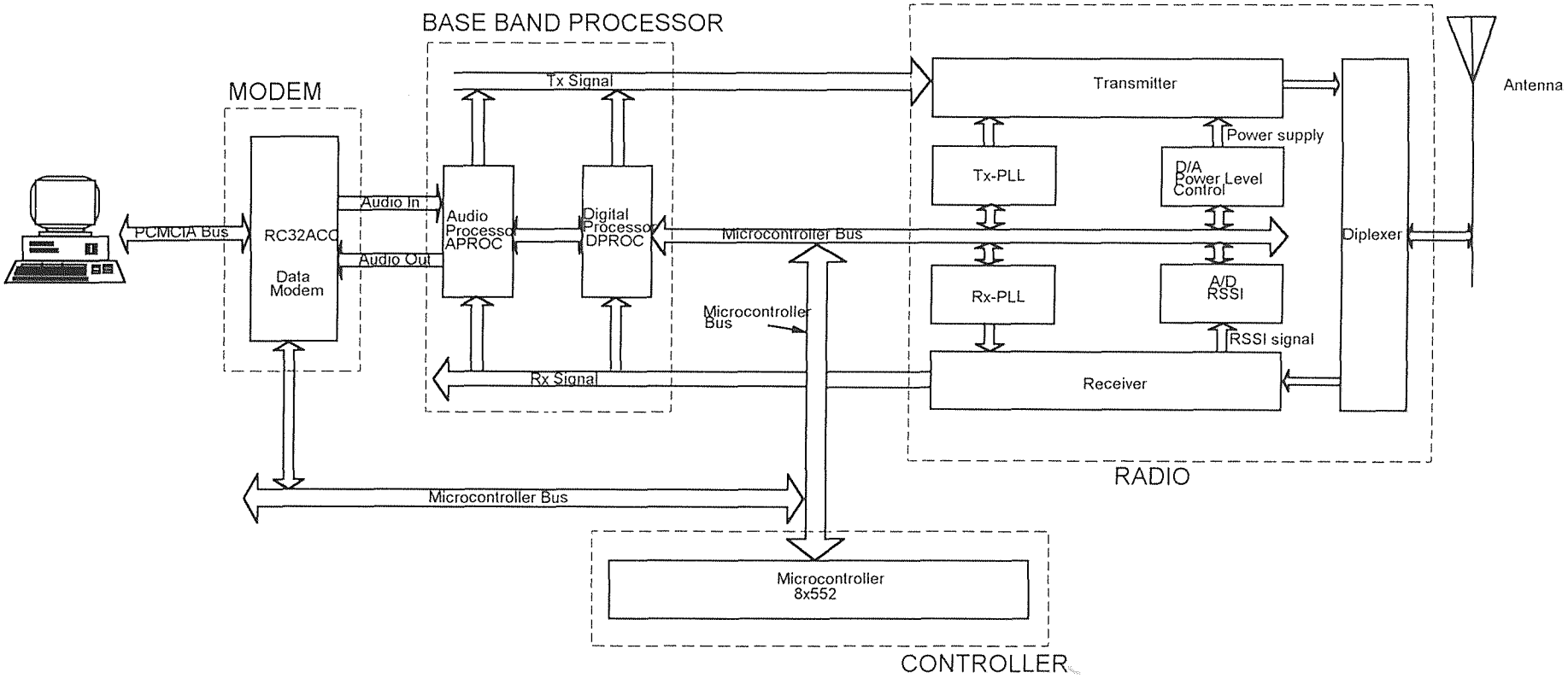

Figure 3.1 Structure of the Wireless Modem

In the modem portion, we used a Rockwell RC32ACC data modem. The RC32ACC modem is a multiple device set that provides the protocols and baseband signal processing needed to support the CDPD standard. RC32ACC modem is itself a stand-alone intelligent modem, but it becomes a component in our bigger intelligent modem, which has many more additional features and the capability to combine AMPS and CDPD functionality together.

In the controller portion, an 8x552 microcontroller and its supporting program and data memory provide the control for the entire wireless modem. The microcontroller is connected to other components via the I^2C bus (DPROC, A/D RSSI, D/A power level control) or standard serial bus (RC32ACC modem). The user input orders through the Keyboard Scan Bus.

All the hardware components mentioned here will be discussed in more detail in the next section.

3.4 Hardware Implementation

3.4.1 The 8x552 Microcontroller

The 8x552 microcontroller is one important member of the well-known 8051 family [6]. In order to fully understand the structure of 8x552, we give a brief review of some important features offered by the 8051 microcontroller family. The core features of 8051 include:

- 8-bit CPU and instruction set optimized for control applications,
- Extensive Boolean processing capabilities, suitable for the control of on-off operations,
- Four register banks, suitable for interrupt processing (no need to save register contents upon entering an interrupt service routine).

As a relatively young member of the family, 8x552 retains all the merits of 8051, and in the meantime it is equipped with many new features, including

- I^2C bus serial I/O port, plus the original full-duplex UART serial interface. These serial buses ensure the minimum interconnection among modem components,
- An on-chip 10-bit ADC with eight multiplexed analog inputs. This ADC is used to convert the incoming RSSI into digital signal in our application,
- Two 8-bit resolution pulse width modulation outputs. They are used to control the power level of the transmitter in our application,
- Two modes of power reduction--idle mode and power-down mode. These features are frequently used in software to achieve a satisfactory power consumption. In the idle mode, the CPU goes to sleep while at the same time keeping the on-chip peripherals active. In the power-down mode, the oscillator and all other on-chip peripherals are stopped, but the internal RAM contents are saved. Both modes can be entered by executing certain instructions in software and invoked by a hardware reset. Another way to terminate idle mode is through any enabled interrupt,
- 8k x 8 ROM, 256 x 8 RAM, both expandable externally to 64k bytes. This feature is very important for complex design projects (including the project discussed in this thesis), which generally need more memory for program and data storage, and
- Very small physical size.

It has been shown that 8x552 is a good and cost-effective choice, as will become more clear in our later discussion. The block diagram of 8x552 is shown in Figure 3.2 [6].

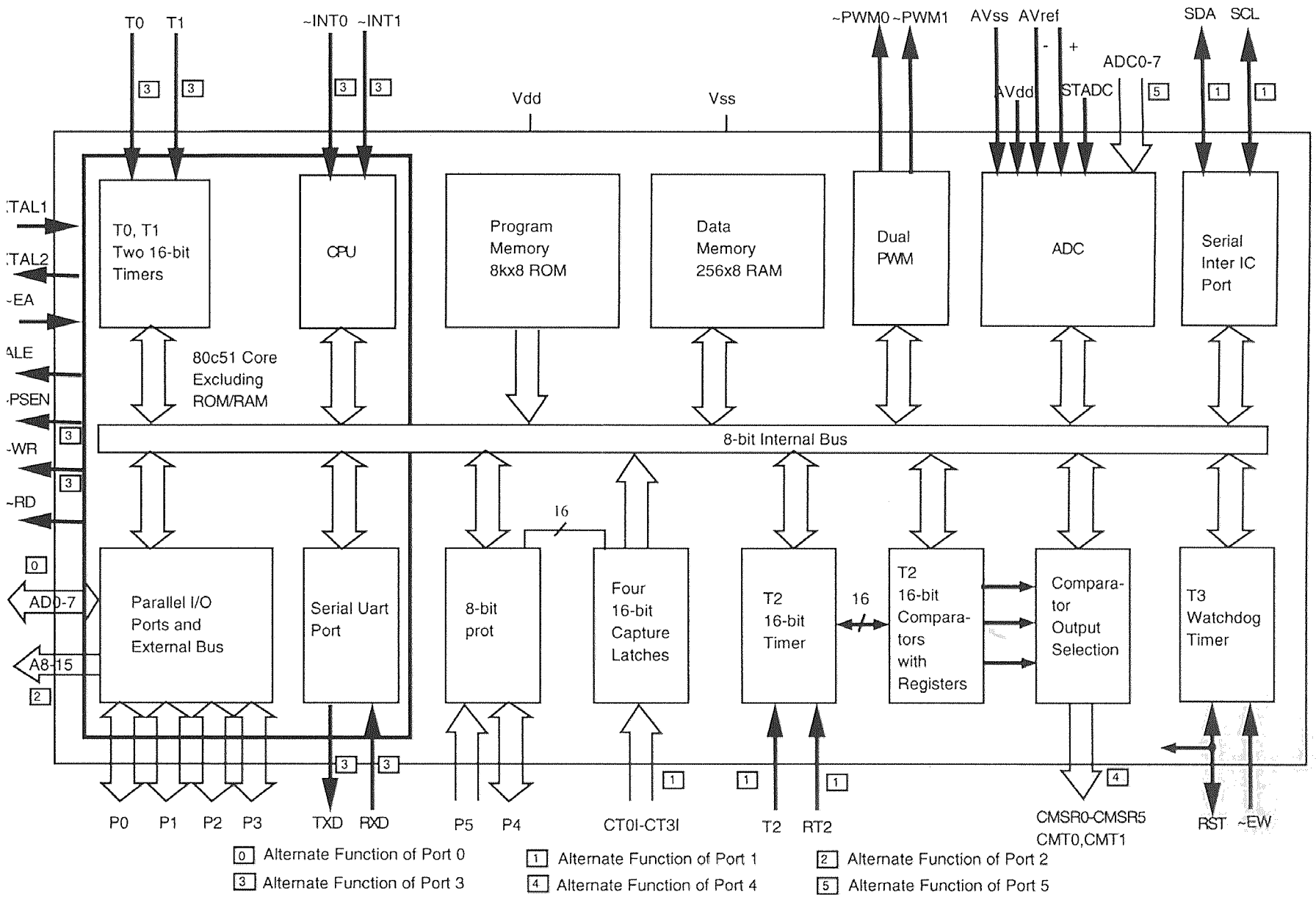

Figure 3.2 Block Diagram of 8x552 [6]

3.4.2 The RC32ACC Wireless Data Modem

The RC32ACC data modem consists of an L39 microcontroller unit (MCU) and a RC32DPC modem data pump (MDP) device. The block diagram given in Figure 3.3 [4] shows the interconnections between RC32ACC and its peripherals. This device set can simultaneously provide two sets of functionality: the CDPD packet radio modem functionality and the wireline data and fax functionality. An application can use either one of them or both for wireless and/or wireline connectivity. In this particular project, we use only its CDPD packet radio modem part, which implements physical, MAC (Medium Access Control), and DLC (Data Link Control) layers of the CDPD network. More specifically, the modem should be able to accomplish tasks like data framing and formatting, Reed-Solomon block coding/decoding, data scrambling/ descrambling, and GMSK filtering/data detection.

These functions are accomplished by hardware as well as supporting software/ firmware of the RC32ACC data modem. The RC32ACC device set receives and transmits commands and data from/to both host/DTE (Data Terminal Equipment) and RFM (Radio Frequency Module). When communicating with host/DTE, AT commands are sent to RC32ACC by the host/DTE to direct the device set to enter either the *configuration mode* or the *communication mode*. As already indicated by their names, in configuration mode, identity and characteristics of the DTE/DCE (Data Communication Equipment) interface information are exchanged between host and the RC32ACC modem; while in communication mode, data packets are transferred. Also, in communication mode, commands and response frames other than data packets are transferred between host and the RC32ACC modem. These command and response frames are known as CDPD Device Programming Interface (CDPI) frames, and have a fixed format.

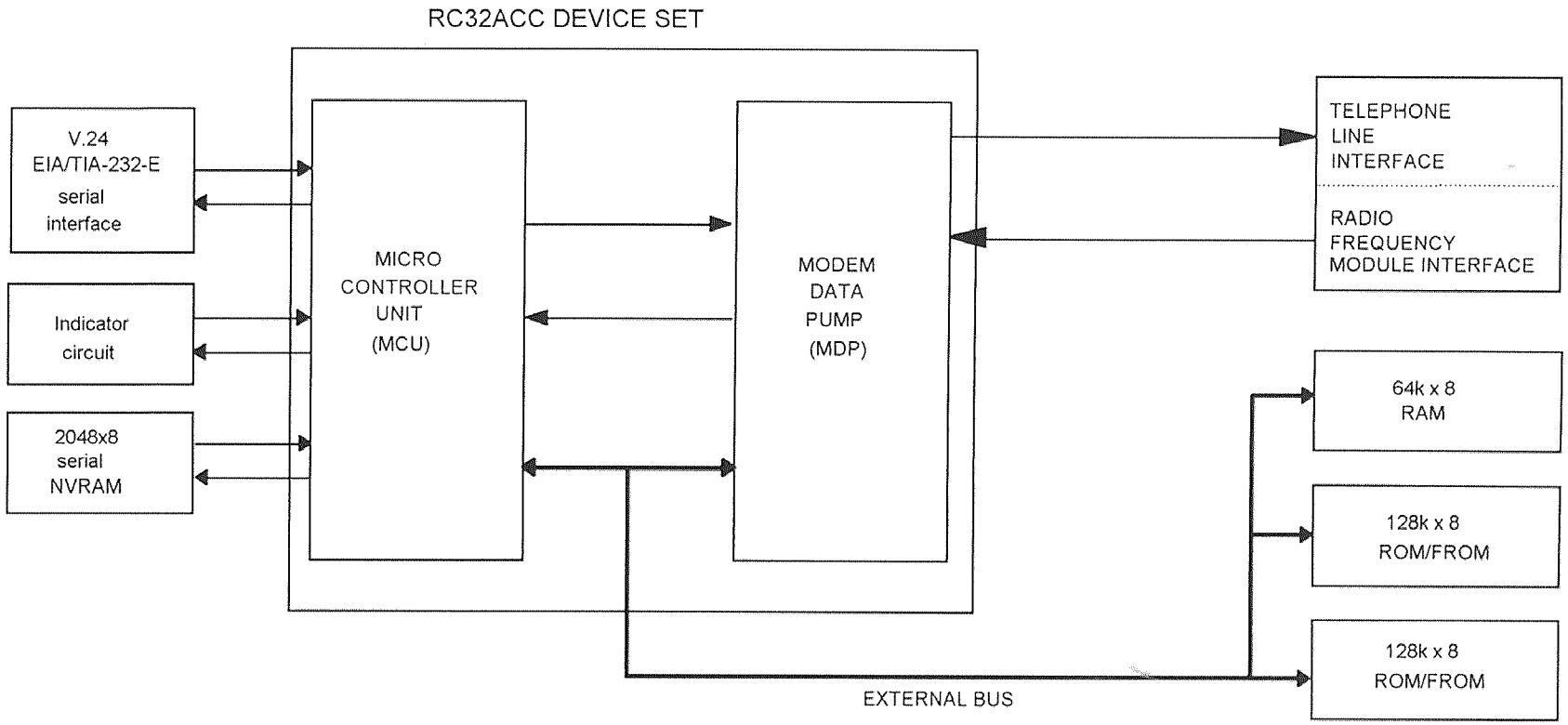

Figure 3.3 The Interconnection between RC32ACC Data Modem and its Peripherals

The command and data transfer between RC32ACC and RFM fall into two branches. One branch is the GMSK filtered data packets going into (or coming from) the APROC (see Figure 3.1). The other branch is the control and response packets with fixed format exchanged between RC32ACC and the 8x552 microcontroller via the standard serial bus. In order to ensure data integrity, the required correspondence or protocol between the two ends is quite complex (as this part is subject to software implementation, it will be discussed in more detail in next chapter).

The RC32ACC data modem also supports two power-reduction modes: sleep mode and stop mode. The modem enters the sleep mode if there is no host activity within a certain period of time. When the DTE writes a character to the modem, or when the system timer expires, sleep mode is terminated. Stop mode is entered if the input line ~STPMODE is asserted, and it is terminated when the same input returns to high. In the stop mode, all activities in the modem devices are frozen, and even less power is consumed as compared to the sleep mode. These two modes are essential for achieving low power consumption, as we discussed before.

3.4.3 The I^2C bus

Inter IC or I^2C bus is a simple bi-directional 2-wire bus introduced by Philips [6]. Due to its conceptual as well as practical simplicity, I^2C bus is now readily accepted by many electronics manufacturers. A typical I^2C bus configuration is shown in Figure 3.4. The I^2C bus uses only two wires (SDA and SCL) to transfer information among devices connected to it, where SDA is the data line, and SCL is the clock line. The I^2C bus ensures the minimum interconnections between the microcontroller and other modem components, which in turn gives the modem high reliability.

The I^2C bus has many other merits apart from its simplicity, and its main features include:

- Bi-directional (half-duplex) data transfer between master and slave,
- Arbitration between simultaneously masters without corruption of data on the bus,
- Serial clock synchronization over SCL line allows devices with different speed to communicate with each other through a single serial bus (SDA line). This feature is crucial in our design, in which the speed of DPROC, A/D RSSI and D/A power level control are not necessarily the same.

Once connected to I^2C bus, a device can be in any one of the four modes of operation: master transmitter, master receiver, slave transmitter, and slave receiver. It follows that there are two types of data transfer possible: from a master transmitter to a slave receiver, and from a slave transmitter to a master receiver. In the former case, the master transmits the slave address followed by data bytes to the slave, and the slave returns an acknowledge bit after the reception of each byte. In the latter case, the communication is again initiated by the master by sending a slave address. The slave returns an acknowledge bit followed by data bytes. The master responds to each received byte by returning an acknowledge bit, except for the last received byte, where a not-acknowledge is returned instead, and thus terminates the transmission. In both cases, the master is the supplier of bus synchronization (clock) pulses.

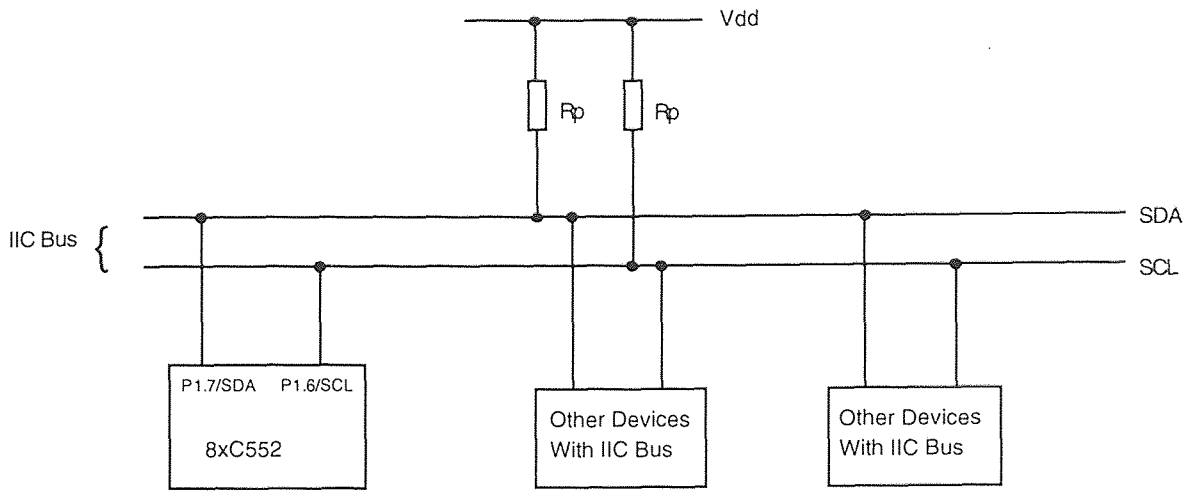

Figure 3.4 The IIC Bus Configuration

CHAPTER 4

SOFTWARE IMPLEMENTATION FOR THE EMBEDDED WIRELESS MODEM

4.1 Modular Programming Technique

Modular programming is a programming technique that divides the program to be developed into a set of interrelated function *modules* which are later linked together to form the complete program. From the point of view of control-flow, the same notion is sometimes referred to as *structured programming*.

Among many other benefits, the major advantage of modular programs is the ease of maintenance. In a large program that is prone to design flaws, it is reasonable to break the program into easy-to-manage modules, where each module is coded and tested individually. Errors in each module can thus be easily located, and debugging is relatively simpler. Another example of the ease-of-maintenance is that, during the process of developing a large program, staff turnovers may happen frequently, leaving unfinished programs which cannot be understood by anyone else except the programmer. This cannot generate serious impact on a modular design.

In order to make a modular program easily understandable, the logical relationship between modules should be kept simple, or equivalently, the interconnections between modules should be minimized, which can be done in the program structure design.

In summary, modular programming technique gives us more control and hence higher quality of a large program.

4.2 Software Implementation for the CDPD Standard

In this section, we focus mostly on the implementation of the protocol between the data modem and the microcontroller. Furthermore, as the software is developed for an industrial application in a particular (wireless) environment, analysis of the software's modularity, robustness, and power consumption will also be given.

4.2.1 The Serial I/O Protocol

In terms of technical jargons, the data modem is more customarily referred to as PDM (Packet Data Modem), and the microcontroller and other components are collectively referred to as RFM (Radio Frequency Module). These two terms will be used throughout the following discussion.

Information in the form of data packets is exchanged between PDM and RFM via the standard serial interface provided on both devices. Designed for general purposes, PDM is intended to be able to connect to any other RFM with a standard serial bus, and it does not support the I^2C bus interface. The packet format used is shown in Figure 4.1. The meaning of each field in Figure 4.1 is explained as follows [4]:

SOF (Start of Frame). 1 byte; SOF is the start of Text (STX) character, and the value is 02H. It is introduced to maintain frame synchronization.

Control Field. The Control Field is 2 bytes in length and contains the length (LEN) of the message and the Command Number (CMD)

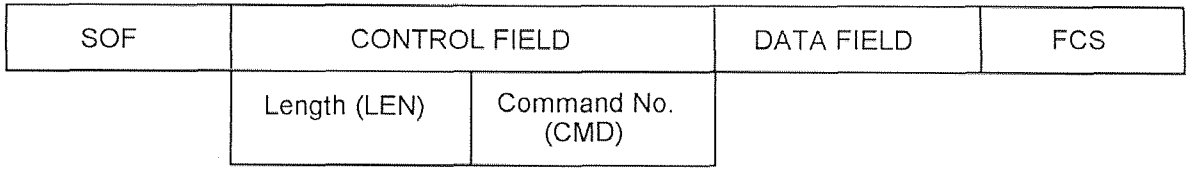

Figure 4.1 Serial Data Packet Format

Table 1. Commands between RC32ACC and Microcontroller

Command Name	Command Type	Command Number (CMD)	Length of Data Field (LEN)	Acknowledge Required (ACK)
Acknowledge (ACK)	Response	00	0	No
Radio Type Query/Response	Query/Response	40	2 or 3/4 or 5	No
Radio Reset (RES)	Command	41	0	Yes
Sleep Mode (SM)	Command	42	0	Yes
Wake Up (WAK)	Command	43	0	Yes
Radio Power Off (OFF)	Command	44	0	Yes
Radio Power On (ON)	Status	45	7/0	No
Output Power Level (PWR)	Command	50	1	Yes
Channel Numbers (FRQ)	Command	51	4	Yes
RSSI Update (RSU)	Status	53	1	Yes
RSSI Mode (RSSM)	Command	54	2	Yes
Radio Mode (RM)	Command	56	1 or 2	Yes
Packet Not Supported (NOS)	Response	FE	0	No
Negative Acknowledge (NAK)	Response	FF	0	No

Length (LEN). The LEN defines the number of bytes of the message in hexadecimal including the Control Field, the Data Field, and the FCS. It also includes any data transparency DLE characters (explained later) added into the data field.

Command Number (CMD). 1 byte. The CMD identifies the Command Number.

Data Field. The Data Field contains the message data. The maximum length is 60 bytes including inserted transparency bytes. In fact, in CDPD standard Release 1.1, the maximum length of a packet is much less than 60 bytes. Here, a longer data field is designated for future extensions.

FCS (Field Check Sum). 1 byte. FCS is calculated by the following addition and modulo operations:

1. Sums together all the bytes in the Control Field and the Data Field excluding the SOF,
2. Performs a modulo 256 operation on the sum to obtain an 8 bit result,
3. Takes the two's complement of the 8 bit result in step 2 and appends it to the end of the packet for transmission,
4. The receiving side should sum the Control Field, all the bytes in the Data Field, and the FCS. If the result is zero the packet should be assumed to be error free.

Note that if a byte value of 02H is present in the Control or Data Field, frame synchronization is lost on the receiver side. A so-called *byte stuffing* technique is therefore introduced to avoid this unwanted situation. More specifically, on the transmitter side, a DLE byte (10H) is inserted prior to every SOF byte or DLE byte in Control and Data Field (these DLE stuffing bytes are counted as valid data bytes and are included in the calculation of FCS); while on the receiver side, the DLE stuffing bytes are stripped and the original packet is recovered. A detected SOF character in the Data Field not preceded by a DLE character causes the receiver to start receiving a new packet. The receiver also starts a new packet if an SOF character is detected in the FCS field and the FCS is bad. It does not, however, start a new packet if an SOF character is detected in the FCS and the FCS is good. In such a case, any preceding DLE byte is not stripped from the Data Field.

Because the microprocessor may be interrupted by higher priority tasks when communicating with the data modem, error-free communication of data packets between the two is virtually impossible. For this reason, the protocol employs a mechanism similar to that used in computer networks, i.e., an acknowledge or response packet is returned to the sender after reception of each valid packet, a not-acknowledge packet is returned after reception of each bad packet, and invalid packets are ignored by the receiver. The validity of a data packet is checked using parity, framing, predetermined packet sizes, and packet duration. If there are parity errors, packet length is greater than the maximum length, the entire packet is not received within a reasonable period of time (generally 100 ms), or an FCS error occurs, then the packet is

invalid and is not responded by the receiver. A packet that is not invalid can be either bad or valid. A packet is determined to be bad if it has a good FCS and any of the following circumstances exists:

1. It has a number in the Command Field that is not supported. The receiver responds with the Packet Not Supported (NOS) command.
2. There is a mismatch between the Length and Command fields. The receiver responds with a Negative Acknowledge (NAK) command. The NAK packet is intended to make communications more efficient since the NAK usually arrives earlier than the 100 ms time-out required by the retry after a missing ACK.
3. There is an unrecognized byte in the Data Field. The receiver responds with a Negative Acknowledge (NAK) command.

A packet is valid if it is neither invalid nor bad, and it will be acknowledged by sending an ACK. An acknowledge must be received within 100 ms after the end of the packet. Otherwise the packet is assumed to be lost and a retry is attempted.

The implementation of this protocol is a complex issue because of a large number of possibilities involved. The main program, of which the block diagram is shown in Figure 4.2, is repeated endlessly. On power up, it defaults to the sleep mode and waits for incoming of the first command. This main program is repeated each time a command packet is received, and it uses the UART interrupt service routine to receive and check the validity of the packet, sends positive or negative acknowledge back to the data modem accordingly, and further calls different routines to process different commands.

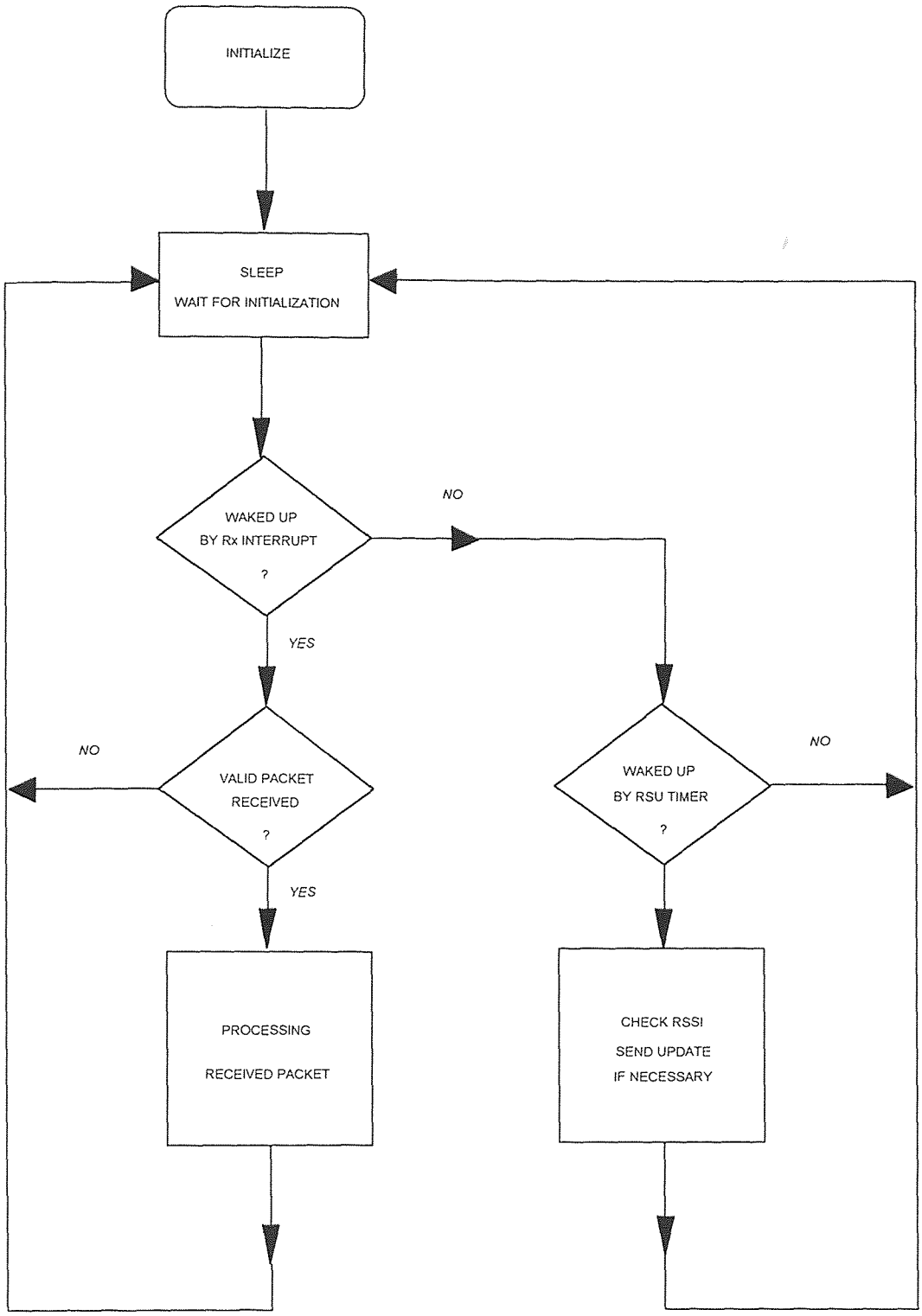

Figure 4.2 Block Diagram of the Main Program for CDPD Standard

After it has accomplished all the tasks, the main program forces the microcontroller to enter sleep mode to conserve power, and waits for the incoming of the next command.

Table 1 lists the commands transferred between PDM and RFM. Among them, the Radio Type Query is sent by PDM at the beginning to determine the capability and version of RFM, and RFM replies this query with the Radio Type Response to identify itself to PDM. The Radio Mode command is sent from PDM to RFM after the Type Query/Response sequence, and is used to set the RFM to desired network mode. Basically, this command is reserved for future use, and has only the CDPD mode in current standard release.

The Radio Reset command is used by the PDM to reset the RFM to a state that output power is selected, the transmitter is off, and status bits are cleared. It is used to regain normal operation when an error state is declared and cannot be resolved by software itself.

The Sleep Mode/Wake Up command pair is used during power saving protocol. When a connection has been established between base and mobile station, packet data is transmitted and received via the transmitter-DPROC-APROC-RC32ACC-host path, and generally there is no communication between PDM and the microcontroller. In this case, the microcontroller is forced into the sleep mode to conserve power. Whenever necessary, the microcontroller is waked up by PDM to resume normal operation.

When there is neither data nor voice transmission between mobile and base station, PDM sends the Radio Power OFF command to power down RFM, hence conserves

power to the maximum extent. The Radio Power On command, in the opposite, turns RFM back on when the host is ready to transmit or receive data packets.

The Channel Numbers (FRQ) command informs the RFM of the current channel number to be used. It is sent to the RFM whenever a current base station is chosen by the PDM, and/or when the PDM has been instructed by the base station to use a new channel. Similarly, the Output Power Level command is sent from PDM to instruct RFM to change its output power. Currently, eight Class I power levels ranging from -6 dBW to -22 dBW are used by mobile stations operating in CDPD mode.

The RSSI Update (RSU) is a somewhat special command: firstly, it is the only command that is sent from RFM to PDM; and secondly, if not acknowledged by PDM, it is resent with a new measurement. RSSI update is reported to PDM autonomously, with a reporting interval preset by the RSSI Mode command.

All queries, responses, and commands described so far are implemented in a modular fashion. But for simplicity, they are not discussed in detail here. Interested readers can find the source code in Appendix A.

4.2.2 Analysis of the CDPD Software

The software for CDPD is developed using the modular programming technique. It is composed of many independent modules and each of which performs a relatively simpler task. For example, there is a dedicated module responsible for the transmission and reception of command packets, and a module responsible for obtaining RSSI signals. These modules are microcontroller-dependent, meaning that when the

microcontroller needs to be changed - a common situation for industrial applications - these modules must also be recoded (sometimes these kinds of modules may be referred to as software *device drivers*). In case that microcontroller is changed, modular design shows one of its benefits: only those microcontroller-dependent modules need to be recoded, while other modules are kept intact. Modular software also lends itself to future extensions. As changes are constantly made to the newly-born CDPD standard, this feature is highly desirable.

Although not obvious, our software implementation is indeed not only modular, but also robust - any unexpected or erroneous state (these states may be entered since interrupt can occur at any point of the program) can eventually be resolved because the microcontroller is forced to sleep after something goes wrong. When a new packet begins at waking up, normal operation is regained. Assuming interrupt at each point and then eliminating possible error loops, the program is coded to ensure robustness. Interested readers can see Appendix A for more details.

4.3 Software Implementation for AMPS

For the AMPS subsystem, a set of library functions (modules) is developed. Among modules developed by the author, the most important ones are the Timer and Extract modules. In the following we shall provide a brief review of them in turn.

The EIA standard for AMPS specifies many timing requirements for signaling purposes. For example, in initialization mode of operation, a mobile station must scan and tune to the strongest dedicated control channel. Then, within 3 seconds, it must

receive a system parameter message. If no message is received within 3 seconds, it must tune to the second strongest channel and try again. Another example is that during the idle task, for every 46.3 ms, a mobile station must execute each of the following three subtasks: response to overhead information, page match, and order. From the above two examples, it is found that the timing requirements need to be implemented as "watchdog" timers in the microcontroller. In fact, many other events in the EIA standard need to be timed. They do not necessarily start at the same time, nor do they stop simultaneously. Actually, at least three timers are needed. However, there are only two hardware timers in an 8x552 microcontroller, which obviously cannot meet the demand. The only solution is so-called *software timer*. Each software timer has a memory location which stores the current time. Each time a hardware timer interrupt routine is entered, this memory location is subtracted by the time elapsed between the two interrupts. Multiple software timers share a same hardware timer, and provide the timing for different events. These software timers have the accuracy of approximately 0.1 ms, and are implemented using assembly language to optimize speed.

Another important set of functions is the Extract functions. They are designed to extract certain bit or bits from the received message word. We can do the same work using functions provided by the C language, but their execution takes too much time; and in an application where power consumption is a major concern, this is intolerable. The main strategy used in our design is to pass known information (such as which byte does the bit to be extracted lie in, how many times the target byte should be shifted) to the Extract routines, instead of letting the routine calculate them again. The source code

in Appendix B is again written in assembly to optimize speed, and is several times faster than routines coded in C language (See Chapter 5 for performance comparisons).

CHAPTER 5

RESULTS AND CONCLUSION

5.1 Simulation and Emulation Results

The software modules designed are first simulated using the simulator by Keil Software. The powerful Keil tool set, including C compiler, assembler and debugger, is optimized for the 8051 family. C programming language is used in most of the modules we have implemented so far. Programs written in C are easier to maintain than those coded using assembly, and can readily be adapted to different platforms. However, there are a few exceptions: the Timer module and Extract module discussed in the last chapter are two of them - they are coded in assembly language to optimize speed. Although coded in assembly, both of them have interfaces to C, i.e., they can be called just like C subroutines. The C/assembly interface, which combines the merits of both languages together, is accomplished by using special directives provided by the Keil C compiler and assembler. According to the simulation, the Timer module spends approximately 11% of the CPU time on its timer interrupt service routine. The corresponding C program, however, takes almost 90% of the CPU time on the service routine, which is unacceptable. Similarly, the performance of our Extract functions is much improved over that of the same functions coded in C language, and a comparison is shown in Table 2. The main reason for this improvement is that, unlike C library functions, our Extract functions are "specialized" ones, with known information passed to them as parameters.

Table 2. Comparison of Extract Functions Coded in C and Assembly Languages

Operation \ Language	Assembly program (Unit: Instruction Cycle)	C program (Unit: Instruction Cycle)
Extract 4 bits within a byte boundry	72	240*
Extract 4 bits across a byte boundry	57	240*
Extract 1 bit	44	240*

*Average value

For the CDPD software, simulation as well as real-time emulation has been accomplished successfully. A Nohau In-Circuit Emulator and its supporting software, an RC32ACC data modem, and a notebook computer with PCMCIA bus interface have been used as tools for emulation. At the beginning of the emulation process, AT commands are sent from the notebook computer to instruct RC32ACC to CDPD communication mode, and then real CDPD data packets are received from a nearby base station located in northern New Jersey. We have intentionally interrupted the software at every point to test its capability of regaining normal operation, and the results have shown that our design is indeed robust. Furthermore, by putting the microcontroller into sleep whenever possible, our software reduces the power consumption to the largest extent.

5.2 Conclusion and Future Research

This thesis has discussed the design philosophy, design considerations, and implementation issues of an embedded wireless modem conforming to both Advanced Mobile Phone System (AMPS) and Cellular Digital Packet Data (CDPD) standards. In summary, the contributions of this thesis are:

1. A high-level review of the CDPD standard is presented. The relationship between AMPS and CDPD is fully explained and analyzed.
2. The design philosophy for embedded systems is proposed and explained.

3. The hardware architecture of an embedded wireless modem that intends to implement both AMPS and CDPD functionality is presented. And this architecture has further been shown to be consistent with our design philosophy.
4. The methodology used in software development for the wireless modem is discussed. Under this methodology, a complete set of software modules for the CDPD standard has been developed and tested. Furthermore, analysis of the software's modularity, robustness, and the ability to conserve power is also given.

The wireless modem discussed here will be commercially available in the near future. Currently, the software for the AMPS subsystem is still under development, and the coordination of CDPD and AMPS software modules leaves considerable work for the designers.

In the current cellular market, competition is stiff, and new features and extensions are constantly added on the existing CDPD standard to satisfy various needs. Therefore, our future research effort will be concentrated on these new features, and the software design will be modified and enhanced accordingly.

APPENDIX A

SOURCE CODE FOR CDPD STANDARD

```
/*-----------------------------------------------
MAIN.H
-----------------------------------------------*/
#define FOSC    12000000
#define BAUDRATE 9600
#define AUTOLOAD 256-FOSC/3686400      /* 3686400=384*BAUDRATE */

#define STX 0x02
#define DLE 0x10

#define TYP 0x40              /* @ */
#define RES 0x41              /* A */
#define SM  0x42              /* B */
#define WAK 0x43              /* C */
#define OFF 0x44              /* D */
#define ON  0x45              /* E */
#define PWR 0x50              /* P */
#define FRQ 0x51              /* Q */
#define RSU 0x53              /* S */
#define RSSM 0x54             /* T */
#define RM  0x56              /* V */

#define ACK 0x00
#define NOS 0xFE
#define NAK 0xFF

/* The follwoing are the packet length after striping off the transparency bytes */

#define ACK_LEN       0x03
#define NAK_LEN       0x03
#define NOS_LEN       0x03

#define RES_LEN       0x03
#define SM_LEN        0x03
#define WAK_LEN       0x03

#define TYP_QRY_LEN   0x05    /* No extension byte */
#define TYP_RESP_LEN  0x07    /* No extension byte */

#define OFF_LEN       0x03
#define ON_LEN        0x0A
#define ON_RESP_LEN   0x03

#define PWR_LEN       0x04
#define FRQ_LEN       0x07
#define RM_LEN        0x04    /* No extension byte */

#define RSU_LEN       0x04
#define RSSM_LEN      0x05

/* The following are some bytes used in TYP and RM packets */

#define VPDM          0xFF    /* Real value TBD */
#define VRFM          0xFF    /* Real value TBD */

#define TYP_RESP_BYTE1  0x03    /* Real value TBD */
#define TYP_RESP_BYTE2  0x00    /* Real value TBD */
```

```
#define TYP_RESP_BYTE3  0x02      /* Real value TBD */

#define TYP_QRY_BYTE    0x02      /* Real value TBD */
#define TYP_QRY_BYTE_MASK 0x8F    /* Mask bits 4--6 */

#define RM_BYTE         0x02      /* Real value TBD */
#define RM_BYTE_MASK    0x8F      /* Mask bits 4--6 */

void main (void);
void UART_Isr (void);
```

```
/*-------------------------------------------
MAIN.C PROGRAM FOR THE COMMUNICATION WITH CDPD MODEM
-------------------------------------------*/

#pragma DEBUG CODE

#include <reg552.h>

#include <stdio.h>

#include "main.h"

#include "rxtx.h"

#include "rsu.h"

#include "timers.h"

#include "response.h"

#include "power.h"

#include "stuff.h"

char data packet[63];
char data rx_count=0;
char data tx_count=0;
bit rx_int=0;
bit rsu_int=0;
bit valid_packet=0;
bit invalid_packet=0;

void main(void) {

 /*****     INITIALIZE     *****/
 Initialize:
 RFM_Initialize();
 TIMER_Initialize();
 UART_Initialize();


 /*****     RFM_Sleep     *****/

 Sleep:
 PCON |=0x01;

 /***** If it's serial receiver interrupt *****/

 if (rx_int) {

    TIMER_Set(0,90);                    /* Set an 90ms timer at beginning,  */
                                        /* allow for possible inaccuracy of */
                                        /* software timer (the standard is  */
                                        /* 100ms)                           */

    while (!valid_packet) {             /* Wait for the packet to complete, */
```

```
    if (invalid_packet) {        /* If an invalid_packet received,  */

      REN = 0;                   /* Disable the receiver interrupt  */
      invalid_packet=0;          /* Reset invalid_packet flag       */
      rx_count=0;
      rx_int=0;

      TIMER_Set (0,10);          /* Wait for 10ms to complete the   */
      while (!TIMER_TimeUp(0));  /* invalid packet                  */

      REN = 1;                   /* if 10ms expired, then enable    */
      goto Sleep;                /* serial interrupt and sleep again */
    }

if (TIMER_TimeUp(0)) {           /* Check if time expired           */
      rx_count=0;
      rx_int=0;
      goto Sleep;                /* then goto sleep                 */
    }

}

REN = 0;                         /* Disable receiver                */
valid_packet=0;                  /* Reset valid_packet flag         */
rx_count=0;
rx_int=0;

Strip_Stuffing();



switch(packet[2]) {

   case ACK:
      if (packet[1]!=ACK_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      break;

   case TYP:          /*TYP*/
      TYP_Response();
      break;

   case RES:          /*RES*/
      if (packet[1]!=RES_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      else {
        Ack_Nak_Nos_PowerOn(ACK);
        goto Initialize;
      }
      break;

   case SM:           /*SM*/
      if (packet[1]!=SM_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      else {
        Ack_Nak_Nos_PowerOn(ACK);
```

```
        REN=1;
        goto Sleep;
      }
      break;

  case WAK:          /*WAK*/
      if (packet[1]!=WAK_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      else
        Ack_Nak_Nos_PowerOn(ACK);  /*Already waked up*/
      break;

  case OFF:          /*OFF*/
      if (packet[1]!=OFF_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      else {
        Ack_Nak_Nos_PowerOn(ACK);
        REN=1;
        PCON |=0x02;          /*Turn off RFM (power down mode)*/
      }
      break;

  case ON:           /*ON*/
      if (packet[1]!=ON_LEN)
        Ack_Nak_Nos_PowerOn(NAK);
      else {
        Ack_Nak_Nos_PowerOn(ON);  /*Already turned on*/
      }
      break;

  case PWR:          /*PWR*/
      Power_Set();
      break;

  case FRQ:          /*FRQ*/
      /*Check FRQ packet, response with ACK or NAK, */
      /*then call Channel_Set (up,down);          */
      break;

  case RSSM:         /*RSSM*/
      RSSI_Mode_Set();
      break;

  case RM:           /*RM*/
      if ( (packet[1]!=RM_LEN) || (RM_BYTE!=(packet[3]&RM_BYTE_MASK)) )
        Ack_Nak_Nos_PowerOn(NAK);
      else
        Ack_Nak_Nos_PowerOn(ACK);
      break;

  default:
      Ack_Nak_Nos_PowerOn(NOS);
}


REN = 1;          /* Enable receiver                    */
```

```
    goto Sleep;         /* Goto sleep again after processing packet    */
}

/***** If it's RSU timer interrupt *****/

else if (rsu_int) {     /* If 20ms time is up                          */

    REN=0;              /* Disable receiver when prepare RSU packet    */
    rsu_int=0;
    RSSI_Mode_Check();  /* Check if it's time to send an RSSI update,  */
                        /* or if 2dB difference is found (normal mode) */
                        /* and send an update if answer is positive    */

    REN=1;              /* Enable receiver again                       */
    goto Sleep;         /* Goto sleep after sending RSSI_Update packet */
}

/*****  If it's other interrupts  *****/
else
    goto Sleep;
}
```

```
/*---------------------------------------------------------------------
POWER.C
---------------------------------------------------------------------*/
#include <reg552.h>

#include "main.h"
#include "power.h"
#include "response.h"

char code TxPowerLookUp[7] ;      /* The values contained in the look-up table */
                          /* are ten times the actual power values     */
                                        /* Level 0 is the highest power level  */
                                  /* Level 7 is the lowest power level   */


void RFM_Initialize(void) {
 PWMP=PRESCL;                 /* Initialize the prescaler   */
 PWM0 = TxPowerLookUp[0];      /* Full output power selected */

 /************************/
 /****turn off tx here****/
 /************************/
}



void Power_Set (void) {

 extern char data packet[];
 char level;

 level=packet[3]-0x03;

 if ( (level<0) || (level>7) || (packet[1]!=PWR_LEN) )
   Ack_Nak_Nos_PowerOn(NAK);
 else {
   PWMP=PRESCL;                  /* Initialize the prescaler */
   PWM0 = TxPowerLookUp[level];   /* Calculate the value in PWM0 */
   Ack_Nak_Nos_PowerOn(ACK);
 }
}

/*---------------------------------------------------------------------
RESPONSE.C
---------------------------------------------------------------------*/
#include <reg552.h>

#include "main.h"
#include "response.h"
#include "stuff.h"

extern char data tx_count;
extern char data packet[];


void TYP_Response (void) {
```

```
  if ( (packet[1]!=TYP_QRY_LEN) || (TYP_QRY_BYTE!=(packet[4]&TYP_QRY_BYTE_MASK)) )
      Ack_Nak_Nos_PowerOn(NAK);

  else {
      packet[0]=STX;
      packet[1]=TYP_RESP_LEN;
      packet[2]=TYP;
      packet[3]=VRFM;
      packet[4]=TYP_RESP_BYTE1;
      packet[5]=TYP_RESP_BYTE2;
      packet[6]=TYP_RESP_BYTE3;

      Insert_Stuffing_FCS();

      tx_count=0;
      TB8=1;
      S0BUF=packet[0];
      while (tx_count<=packet[1]);
      tx_count=0;
  }
}


void Ack_Nak_Nos_PowerOn (char annp) {

      packet[0]=STX;
      packet[1]=ACK_LEN;
      packet[2]=annp;

      tx_count=0;
      TB8=1;
      S0BUF=packet[0];
      while (tx_count<=packet[1]);
      tx_count=0;
}
```

```
/*-------------------------------------------------------------------
RSSI.C
-------------------------------------------------------------------*/
#include <intrins.h>
#include <reg552.h>
#include "rssi.h"

char RADIO_GetRSSI (void) {

 ADCON=0x00;
 ADCON=ADCON | 0x08;
 while (!(ADCON&0x10));
 return (ADCH);
}
```

```
/*---------------------------------------------------------------------------
RSU.C
---------------------------------------------------------------------------*/
#include <reg552.h>


#include "main.h"
#include "rsu.h"
#include "stuff.h"
#include "rssi.h"
#include "response.h"


char data rssi_mode;
int  data rssi_period;          /* The period to report RSSI signal in periodic mode */
int  data rssi_count;

char  data rssi_new;
char  data rssi_old;

bit scan_complete=0;



void RSSI_Mode_Set(void) {

      extern char data packet[63];

      bit nak_flag=0;

      if (packet[1]!=RSSM_LEN)
          nak_flag=1;

      switch (packet[3] & 0x0F) {          /* Bit 0--3 is the RSSI mode          */

          case 0x00:
          case 0x01:
          case 0x03:
          case 0x07:
              rssi_mode=packet[3] & 0x0F;
              break;
          default:
              nak_flag=1;
      }

      switch (packet[3] & 0xF0) {          /* Bit 4--7 is the units of RSSI period */

                                  /* Calculate how many units (a unit is  */
                                  /* 20ms) to send RSSI_update, in scan   */
                                  /* mode (only once), or periodically in */
                                  /* periodic mode                        */
          case 0x00:
              rssi_period=packet[4];
              break;

          case 0x10:
```

```
            rssi_period=packet[4]*5;
            break;

        case 0x20:
            rssi_period=packet[4]*50;
            break;

        default:
            nak_flag=1;
    }


    if (nak_flag)
        Ack_Nak_Nos_PowerOn(NAK);

    else {
        rssi_count=0;
        scan_complete=0;
        rssi_old=RADIO_GetRSSI();
        RSSI_Update(rssi_old);
        Ack_Nak_Nos_PowerOn(ACK);
    }
}


void RSSI_Mode_Check (void) {

    rssi_new=RADIO_GetRSSI();
    rssi_count++;

    switch (rssi_mode) {

     case 0x00:                 /* Normal mode */
      if (rssi_count==50) {     /* Update RSSI every second */
       RSSI_Update(rssi_new);
       rssi_count=0;
      }

      else if ( (rssi_new-rssi_old)>2 ||(rssi_old-rssi_new)>2 ) /* Or when 2 dB change */
       RSSI_Update(rssi_new);
      break;

     case 0x01:                 /* Periodic mode */
      if (rssi_count==rssi_period) { /* Update RSSI when rssi_period*20ms has elapsed */
       RSSI_Update(rssi_new);
       rssi_count=0;
      }
      break;

     case 0x03:                 /* Scan mode */
      if (!scan_complete && rssi_count==rssi_period) { /*Update *once* after rssi_period*20ms */
       RSSI_Update(rssi_new);
       rssi_count=0;
       scan_complete=1;
      }
      break;
```

```
        case 0x07:
                        break;

      }

     rssi_old=rssi_new;
}



void RSSI_Update (char rssi) {
      extern char data tx_count;

      packet[0]=STX;
      packet[1]=RSU_LEN;
      packet[2]=RSU;
      packet[3]=rssi;

      Insert_Stuffing_FCS();

      tx_count=0;
      TB8=1;
      S0BUF=packet[0];
      while (tx_count<=packet[1]);
      tx_count=0;
}
```

```
/*---------------------------------------------------------------------
RXTX.C
---------------------------------------------------------------------*/
#include <reg552.h>

#include "main.h"
#include "rxtx.h"


void UART_Initialize(void) {
   EA   = 1;                    /* Each interrupt individually enabled     */
   ET1  = 0;                    /* Disable timer 1 interrupt               */
   TMOD |= 0x20;               /* TMOD: timer 1, mode 2, 8-bit reload     */
   TH1  = AUTOLOAD;             /* TH1:  reload value for 9600 baud        */
   TR1  = 1;                /* TR1:  timer 1 run                       */

   ES0  = 1;                        /* Enable UART interrupt                   */
   S0CON = 0xd0;               /* SCON: UART, 9-bit; enable reception     */
   PCON &= 0x7F;               /* Set SMOD to 0                           */
   PS0  = 1;                /* Set serial interrupt priority to high   */

 }


/*SERIAL PORT INTERRUPT SERVICE ROUTINE*/

void UART_Isr (void) interrupt 4 {

   extern char data packet[63];
   extern char data rx_count;
   extern char data tx_count;
   extern bit rx_int;
   extern bit valid_packet;
   extern bit invalid_packet;

   char i,check_sum;
   bit parity;

   /*****   If it's receiver interrupt     *****/

   if (RI) {
      RI = 0;
      rx_int = 1;

      /***** Move the received byte to RAM *****/

      ACC = S0BUF;
      parity = P;
      packet[rx_count]=S0BUF;



      /*****     Check for parity error    *****/

      if (parity!=RB8) {
            invalid_packet=1;
      }
```

```
/*****   Check if the 1st byte is STX   *****/

if (rx_count==0) {
      if (packet[0]!=STX){
            invalid_packet=1;
      }
}



/*****    Check if the length greater than 63    *****/

else if (rx_count==1 && packet[1] > 63) {
        invalid_packet=1;
}



/*****     Check for end or start of packet   *****/

else {
        if (packet[rx_count]==STX) {
                if (rx_count!=packet[1]) {
                      if (packet[rx_count-1]!=DLE) {
                            rx_count=0;
                      }
                }
                else {
                      check_sum=0;

                      for (i=1;i<=packet[1];i++)
                            check_sum+=packet[i];

                      if (check_sum!=0) {
                            rx_count=0;
                      }
                      else {
                            valid_packet=1;
                      }
                }
        }
        else if (rx_count==packet[1]) {
                check_sum=0;
                for (i=1;i<=packet[1];i++)
                      check_sum+=packet[i];
                if (check_sum!=0)
                      invalid_packet=1;
                else
                      valid_packet=1;
       }

}

/*****          Return          *****/
```

```
    rx_count++;

  }



  /***** If it's transmitting interrupt *****/

  else {
    TI=0;

    tx_count++;                    /* packet[0] has already been sent */
    if (tx_count<=packet[1]) {
      ACC=packet[tx_count];
      parity=P;
      TB8=parity;
      S0BUF=packet[tx_count];
    }
  }
}
```

```
/*-------------------------------------------------------------------
STUFF.C
-------------------------------------------------------------------*/
#include "main.h"
#include "stuff.h"

extern char data packet[63];

void Strip_Stuffing (void) {
  char i=2;
  char j;

  while (i<packet[1]) {
    if (packet[i]==DLE) {
      for (j=i;j<packet[1];j++) {
        packet[j]=packet[j+1];
      }
      packet[1]--;
    }
    i++;
  }
}


void Insert_Stuffing_FCS (void) {
  char i=2;
  char j;

  while (i<packet[1]) {
    if (packet[i]==DLE || packet[i]==STX) {
      for (j=packet[1];j>=i;j--) {
        packet[j+1]=packet[j];
      }
      packet[i]=DLE;
      packet[1]++;
      i+=2;
    }

    else
      i++;
  }



  j=0;                    /* Use j to calculate check-sum */
  for (i=1;i<packet[1];i++)
    j+=packet[i];


  i=packet[1];
  packet[i]=~j+1;         /* ~j+1 gives the FCS          */
}
```

APPENDIX B

SOURCE CODE FOR AMPS AND OTHER LIBRARY FUNCTIONS

```
/*---------------------------------------------------------------------------
TIMERS.C
---------------------------------------------------------------------------*/

#pragma DEBUG CODE SRC

#include <reg552.h>

#include "main.h"

#include "timers.h"

extern bit rsu_int;

/* TIMER_Initialize */

void TIMER_Initialize (void) {

 TR0=0;              /* Timer 0 is now idle */

 TMOD|=0x02;          /* Set Timer0 mode to mode2 */
 EA=1;               /* Global interrupt enable  */
 ET0=1;              /* Enable timer 0 interrupt */
 TH0=LDVAL;           /* Load timer 0 high byte with autoload value   */
 TL0=LDVAL;           /* Load timer 0 low byte with autoload value    */


 #pragma asm

 PUSH PSW
 MOV PSW,#010H

 ;Set the three software timers to 0

 MOV R0,#0H    ;R0--low  byte of software timer 0
 MOV R1,#0H    ;R1--high byte of software timer 0
 MOV R2,#0H    ;R2--low  byte of software timer 1
 MOV R3,#0H    ;R3--high byte of software timer 1
 MOV R4,#0H    ;R4--low  byte of software timer 2
 MOV R5,#0H    ;R5--high byte of software timer 2

 POP PSW
 #pragma endasm

 TR0=1;              /* Timer 0 is now running */
}

/* Timer 0 interrupt service routine */

void TIMER_Isr(void) interrupt 1 {    /* The interrupt number for timer 1 is 3 */
                                      /* Use reg. bank 2, for low-priority interrupts */

 static char data count1=0;
 static char data count2=0;

 count1++;                        /* Counts for the total of interrupts generated.  */
                                      /* When interrupt occured 4 times, decrement      */
```

```
                                        /* each active software timer by 1. Interrupt    */
                                        /* occurs every 0.25ms, thus active timers are   */
                                        /* decremented by 1 every 1ms                    */

if (count1==4) {
 count2++;
 /* The following assembly decrements each non-zero software timer by 1 */

 #pragma asm
       PUSH PSW
       MOV PSW,#010H

            DEC R0
            MOV A,R0
            INC A
            JNZ SECOND
            DEC R1
            MOV A,R1
            INC A
            JNZ SECOND
            INC R0
            INC R1

 SECOND:  DEC R2
            MOV A,R2
            INC A
            JNZ THIRD
            DEC R3
            MOV A,R3
            INC A
            JNZ THIRD
            INC R2
            INC R3

 THIRD:   DEC R4
            MOV A,R4
            INC A
            JNZ EXIT
            DEC R5
            MOV A,R5
            INC A
            JNZ EXIT
            INC R4
            INC R5

 EXIT:    POP PSW
 #pragma endasm

 if (count2==20) {
  rsu_int=1;
  count2=0;
 }
 count1=0;
}
}

/* TIMER_Set */
```

```
void TIMER_Set (unsigned char timernumber, unsigned int time) {
 #pragma asm
     PUSH PSW   ;Save PSW onto stack

     MOV A,R4   ;Save the parameter "time" onto stack
          PUSH ACC   ;R4--high-order byte
          MOV A,R5   ;R5--low-order byte
          PUSH ACC

          MOV A,R7   ;Save the parameter "timernumber" onto stack
          PUSH ACC

          MOV PSW,#010H  ;Switch from register bank 0 to 2

          POP ACC    ;R7 of register bank 2 now
          MOV R7,A   ;contains parameter "timernumber"

          CJNE R7,#0,NEXT1   ;Branch to software timer other than timer 0

          POP ACC    ;Set software timer 0
          MOV R0,A
          POP ACC
          MOV R1,A
          SJMP END1

 NEXT1: CJNE R7,#1,LAST1   ;Branch to software timer other than 0 or 1

          POP ACC    ;Set software timer 1
          MOV R2,A
          POP ACC
          MOV R3,A
          SJMP END1

 LAST1: POP ACC    ;Set software timer 2
          MOV R4,A
          POP ACC
          MOV R5,A

  END1: POP PSW    ;Restore PSW

  #pragma endasm
}

/* TIMER_TimeUp */

bit TIMER_TimeUp (unsigned char timernumber) {

  #pragma asm
     PUSH PSW   ;Save PSW onto stack

          MOV A,R7   ;Save the parameter "timernumber" onto stack
      PUSH ACC

          MOV PSW,#010H   ;Switch from register bank 0 to 2

          POP ACC    ;R7 of register bank 2 now
```

```
        MOV R7,A   ;contains parameter "timernumber"

    CJNE R7,#0,NEXT2   ;Branch to software timer other than timer 0

        MOV A,R0   ;Check if software timer 0 is timeup
    JNZ EXIT0
    MOV A,R1
    JNZ EXIT0
    SJMP EXIT1

NEXT2: CJNE R7,#1,LAST2   ;Branch to software timer other than 0 or 1

        MOV A,R2   ;Check if software timer 1 is timeup
    JNZ EXIT0
    MOV A,R3
    JNZ EXIT0
        SJMP EXIT1

LAST2:
        MOV A,R4   ;Check if software timer 2 is timeup
    JNZ EXIT0
    MOV A,R5
    JNZ EXIT0

EXIT1:  POP PSW    ;Restore PSW
    SETB C     ;If timeup, set the carry bit as return value
    SJMP END2

EXIT0: POP PSW    ;Restore PSW
    CLR C      ;If not timeup, clear the carry bit as return value

 END2: NOP

 #pragma endasm

}

/* TIMER_Time */

unsigned int TIMER_Time (unsigned char timernumber) {

 #pragma asm
    PUSH PSW   ;Save PSW onto stack

    MOV A,R7   ;Save the parameter "timernumber" onto stack
        PUSH ACC

        MOV PSW,#010H   ;Switch register bank from 0 to 2

        POP ACC    ;ACC now contains parameter "timernumber"

    CJNE A,#0,CONT1   ;Branch to software timer other than timer 0

        MOV A,R1   ;Move the high byte of software timer 0 into R6
        MOV R6,A
        MOV A,R0   ;Move the low byte of software timer 0 into R7
        MOV R7,A
```

```
        SJMP RETURN

CONT1: CJNE A,#1,CONT2    ;Branch to software timer other than 0 or 1

        MOV A,R3    ;Move the high byte of software timer 1 into R6
        MOV R6,A
        MOV A,R2    ;Move the low byte of software timer 1 into R7
        MOV R7,A
        SJMP RETURN

CONT2: MOV A,R5    ;Move the high byte of software timer 2 into R6
        MOV R6,A
        MOV A,R4    ;Move the low byte of software timer 2 into R7
        MOV R7,A

RETURN:
        PUSH ACC    ;Save R7 of register bank 2 onto stack
        MOV A,R6    ;Save R6 of register bank 2 onto stack
        PUSH ACC

        DEC SP     ;Restore PSW, switch back to original register bank
        DEC SP
        POP PSW

        INC SP
        INC SP
    INC SP

        POP ACC     ;R6 of the original register bank now
        MOV R6,A    ;contains high byte of return value
        POP ACC     ;R7 of the original register bank now
        MOV R7,A    ;contains low byte of return value

    POP ACC     ;get rid of the useless byte on stack

 #pragma endasm
}
```

```
/*---------------------------------------------------------------------------
EXTRACT.C
---------------------------------------------------------------------------*/
#pragma DEBUG SRC CODE

#include <reg552.h>

bit exbit (unsigned long msg, char tgt_byte, char tgt_bit)
{
#pragma asm

                MOV DPTR,#?_exbit?BYTE+04H
                MOVX A,@DPTR

EXBIT_In_Byte3:     CJNE A,#3,EXBIT_Not_In_Byte3
                MOV A,R4
                SJMP EXBIT_Cont

EXBIT_Not_In_Byte3:  CJNE A,#2,EXBIT_Not_In_Byte2
                MOV A,R5
                SJMP EXBIT_Cont

EXBIT_Not_In_Byte2:  CJNE A,#1,EXBIT_Not_In_Byte1
                MOV A,R6
                SJMP EXBIT_Cont

EXBIT_Not_In_Byte1:  MOV A,R7

EXBIT_Cont:         PUSH ACC

                INC DPTR
                MOVX A,@DPTR

EXBIT_Bit7:         CJNE A,#7,EXBIT_Not_Bit7
                POP ACC
                ORL C,ACC.7
                SJMP EXBIT_End

EXBIT_Not_Bit7:     CJNE A,#6,EXBIT_Not_Bit6
                POP ACC
                ORL C,ACC.6
                SJMP EXBIT_End

EXBIT_Not_Bit6:     CJNE A,#5,EXBIT_Not_Bit5
                POP ACC
                ORL C,ACC.5
                SJMP EXBIT_End

EXBIT_Not_Bit5:     CJNE A,#4,EXBIT_Not_Bit4
                POP ACC
                ORL C,ACC.4
                SJMP EXBIT_End

EXBIT_Not_Bit4:     CJNE A,#3,EXBIT_Not_Bit3
                POP ACC
                ORL C,ACC.3
                SJMP EXBIT_End
```

```
EXBIT_Not_Bit3:     CJNE A,#2,EXBIT_Not_Bit2
                    POP ACC
                    ORL C,ACC.2
                    SJMP EXBIT_End

EXBIT_Not_Bit2:     CJNE A,#1,EXBIT_Not_Bit1
                    POP ACC
                    ORL C,ACC.1
                    SJMP EXBIT_End

EXBIT_Not_Bit1:     POP ACC
                    ORL C,ACC.0


EXBIT_End:          NOP

#pragma endasm
}


unsigned char exone (unsigned long msg, char tgt_byte, char ror_num, char mask);

void main (void)
{
char x;
unsigned long Temp_Test = 0x02345458;

x = exone (Temp_Test,3,3,0x1F);  /*extract 5 bits starting from bit 27)*/
x = exone (Temp_Test,2,2,0x0F);  /*extract 4 bits starting from bit 18)*/
x = exone (Temp_Test,1,5,0x03);  /*extract 2 bits starting from bit 13)*/
x = exone (Temp_Test,0,0,0x1F);  /*extract 5 bits starting from bit 02)*/

}

unsigned char exone (unsigned long msg, char tgt_byte, char ror_num, char mask)
{
#pragma asm

;
;the following section moves the target byte into accumulator
;

                    MOV R1,?_exone?BYTE+04H

                    CJNE R1,#3,Not_Byte3
                    MOV A,R4                ;target byte is byte 3
                    SJMP Zero

Not_Byte3:          CJNE R1,#2,Not_Byte2
                    MOV A,R5                ;target byte is byte 2
                    SJMP Zero

Not_Byte2:          CJNE R1,#1,Not_Byte1
                    MOV A,R6                ;target byte is byte 1
                    SJMP Zero
```

```
Not_Byte1:      MOV A,R7                ;target byte is byte 0

;
;the following section decides if ror_num is zero
;

Zero:       PUSH ACC
            MOV A,?_exone?BYTE+05H      ;right rotate number
            MOV R0,A
            JNZ Shift           ;if ror_num not zero

            POP ACC             ;if ror_num is zero
            SJMP Msk

;
;the following section shifts target byte by ror_num, AND with mask and return through R7
;

Shift:    POP ACC
Loop:       RR A
            DJNZ R0,Loop

Msk:        ANL A,?_exone?BYTE+06H      ;AND with the mask

            MOV R7,A            ;return value through R7


#pragma endasm
}


unsigned char extwo (unsigned long msg, char tgt_high, char ror_num, char mask1, char rol_num, char mask2);

void main (void)
{
char x;
unsigned long Temp_Test1 = 0x02345458;
unsigned long Temp_Test2 = 0x52765420;

x = extwo (Temp_Test1,2,3,0x1F,5,0x60);  /*extract 7 bits starting from bit 11)*/
x = extwo (Temp_Test2,3,5,0x07,3,0x38);  /*extract 6 bits starting from bit 21)*/
}

unsigned char extwo (unsigned long msg, char tgt_high, char ror_num, char mask1, char rol_num, char mask2)
{
#pragma asm

            MOV R0,?_extwo?BYTE+05H     ;right rotate number

;
;the following section moves two target bytes into accumulator and stack
;

            MOV R1,?_extwo?BYTE+04H

            CJNE R1,#3,Not_Byte32
            MOV A,R4                ;target byte is byte 3 & 2
```

```
                PUSH ACC
                MOV A,R5
                SJMP Loop1

Not_Byte32:     CJNE R1,#2,Not_Byte21
                MOV A,R5                ;target byte is byte 2 & 1
                PUSH ACC
                MOV A,R6
                SJMP Loop1

Not_Byte21:     MOV A,R6                ;target byte is byte 1 & 0
                PUSH ACC
                MOV A,R7


;
;the following section shifts the low-order byte
;


Loop1:          RR A
                DJNZ R0,Loop1

                ANL A,?_extwo?BYTE+06H      ;AND with the mask
                MOV R2,A                    ;move to a temporary register


;
;the following section shifts the high-order byte
;
        POP ACC
                MOV R0,?_extwo?BYTE+07H     ;left rotate number

Loop2:          RL A
                DJNZ R0,Loop2

                ANL A,?_extwo?BYTE+08H      ;AND with the second mask


;
;the following section merge the two bytes into one and return through R7
;

                ORL A,R2                    ;the final result is now in A

                MOV R7,A                    ;return value through R7

#pragma endasm
}
```

REFERENCES

1. K. Feher, "Modems for Emerging Digital Cellular-Mobile Radio System," *IEEE Transaction on Vehicular Technology*, vol. 40, pp. 355-365, May 1991

2. D.J. Goodman, "Trends in Cellular and Cordless Communications," *IEEE Communications Magazine*, vol. 29, pp. 31-39, June 1991

3. W.C.Y. Lee, *Mobile Cellular Telecommunications*, 2nd ed., McGraw-Hill, New York, NY,1995

4. "RC32ACC Wireless Data Modem with V.32 bis Wireline Support," data sheet, Rockwell Inc., June 1994

5. "Mobile Station-Land Station Compatibility Specification, EIA/TIA-553," Engineering Department, Electronic Industries Association, September, 1989

6. "IC20: 80C51-Based 8-bit Microcontrollers," data handbook, Philips Semiconductors, 1995