

University of Waterloo  
Faculty of Engineering

Design Implications of a Serial Protocol in an  
Embedded System

Research In Motion Limited  
Waterloo, Ontario

Gavin J. Hurlbut  
92012471  
3B Electrical Engineering

Originally written January 6, 1997

September 11, 2002

# Contents

<b>Summary</b>	<b>iv</b>
<b>Conclusions</b>	<b>v</b>
<b>Recommendations</b>	<b>vi</b>
<b>Contribution</b>	<b>vii</b>
<b>1 Background</b>	<b>1</b>
1.1 Company Profile . . . . .	1
1.2 Digital Packet Radio . . . . .	1
1.3 OSI Reference Model . . . . .	5
<b>2 Serial Protocol Description</b>	<b>7</b>
2.1 Error Detection . . . . .	8
2.2 Data Framing . . . . .	9
<b>3 Design Restrictions</b>	<b>10</b>
3.1 Embedded System Restrictions . . . . .	10
3.2 Real-Time Characteristics . . . . .	11
<b>4 Optimizations and Compromises Needed</b>	<b>13</b>
4.1 Limited Queuing of Packets . . . . .	13
4.2 Interrupt Handling . . . . .	14
4.3 Single Queuing Path . . . . .	16
<b>5 Debugging in an Embedded System</b>	<b>17</b>
5.1 Separate Debugging Stream . . . . .	17
5.2 Simulation of the Embedded System . . . . .	18
<b>A Glossary</b>	<b>20</b>
<b>Bibliography</b>	<b>21</b>

# List of Figures

1.1	Mobitex Network Architecture . . . . .	3
1.2	DataTAC Network Architecture . . . . .	5
1.3	OSI Reference Model . . . . .	6
2.1	NCL Packet Structure . . . . .	8

# List of Tables

2.1 NCL Protocol Data Escaping . . . . .	9
--	---

# Summary

Although most programmers have plenty of experiences working with platforms with more than adequate resources, designing within the restrictions of an embedded system requires more careful thought. Implementing a serial protocol in an embedded system poses several problems that would not necessarily be encountered in personal-computer based designs.

Most embedded systems impose similar restrictions on software designs, with the most common being limited memory (both for data storage and code storage), slow processor speeds and limited input and output capabilities (both parallel and serial). Perhaps the most difficult restriction imposed by embedded systems is the inherent difficulty of providing debugging support.

To overcome memory limitations, it is often necessary to implement small, limited queues of packets to be sent by the serial protocol. It is crucial that the number of buffers be sufficient to minimize queue overruns such that no packets are lost. Additionally, strict controls over the size of the stack of each task in the embedded system's real-time operating system are required.

Since embedded systems platforms are often on the slow side, care must be taken to streamline the code being written. If a task is to be performed in real-time, it is sometimes necessary to code some sections in assembly language to make the code fast enough to meet the requirements.

Interrupt handlers require even more care, and often must be split into two sections (one that is performed at the time of signalling, and one part that runs at a later time). Another sometimes necessary step is to allow other interrupt sources to override the one currently being handled. There are potentially dangerous side effects to doing this, so great care must be taken when doing this.

As embedded systems generally have no easy facility for debugging code, it is in the best interest of the designer to build in debugging facilities such as a debugging stream that is sent out a serial port. Another beneficial development aid is to use an in-circuit emulator or a custom simulation environment to allow easier code development and testing.

# Conclusions

As RAM is generally the most precious of commodities in an embedded system, many of the methods used to successfully implement a serial protocol in embedded systems concern proper use of the limited RAM available, whether in the form of stack space or in the number or size of buffers.

The other valuable entity in an embedded system is time. Since these systems are often powered by slow processors, it is ever more important that all code be made as streamlined as possible. This is most crucial when dealing with interrupt handlers as these can seriously affect system performance.

Implementing serial protocols in embedded systems can take much more thought and care than is immediately evident. While it is possible to create a viable implementation without much regard given to the complexities and subtleties inherent in an embedded system, such an implementation is nearly always next to impossible to debug and maintain.

With sufficient forethought and wise design choices, it is often simple to create a piece of code that is not only fairly simple to debug, but also is almost a joy to maintain in the future as the number of “odd little features” would be kept to a minimum.

# Recommendations

It would be foolish for any embedded systems designer to ignore the implications of the platform for which they design. To this end, any future projects that include a similar serial protocol should be designed with care given to ensure that the code is simultaneously easy to debug and to maintain.

The most important consideration to be made is to ensure that all interrupt handlers allow real-time operation to occur, whether the handlers are for serial port interrupts, timer interrupts or DSP interrupts. Without real-time operation, it is not likely that a radio modem (or many other similar devices) will work correctly.

If the decision is made to unmask interrupt sources during the execution of an interrupt handler, great care must be taken as it will be nearly impossible to track elusive timing problems or stack overflows within interrupt handlers.

All future embedded systems should also be designed with an eye set towards the ease of debugging, perhaps including a debugging stream to a PC, or by simulation on a PC, or both. This is an incredible time saver for both development purposes and when bugs surface at a later date.

# Contribution

The group I worked in consisted of two full-time software designers and one co-op student (myself). Our group was tasked with generating the firmware for radio modems based on the DataTAC network.

My tasks included :

- implementation of a miniature file system to contain configuration and calibration values
- implementation of the NCL (Natural Command Language) for communications between a radio modem and a host computer
- creating regression test scripts to verify proper operation of the NCL implementation
- automatic generation of documentation from source code comments

This report is concerned with decisions necessary to perform the second of these tasks, that is in implementing the NCL protocol within the constraints created by the radio modem platforms.

# Chapter 1

## Background

### 1.1 Company Profile

Research In Motion (RIM) was founded in 1984, and has earned itself an international reputation for developing high performance RF technologies for narrowband data communications [Res96]. RIM's wireless technology has set new standards for battery life, transceiver performance, physical size and ease of use.

In 1988, RIM began developing software and hardware for wireless networks such as Mobitex and DataTAC, recognizing the huge potential that this market presented. Today, RIM products are sold world-wide through distributors, network operators, OEM partners, and software bundling agreements with companies such as Ericsson, IBM, and Megahertz/U.S. Robotics.

### 1.2 Digital Packet Radio

#### Introduction

An ever-expanding market is emerging for mobile data communications products. Many people are finding uses for hand held and vehicle-mounted radio communications devices, whether for personal communications or for inventory control.

Although paging networks have existed for quite some time, many users have desired the ability to have two-way communications on the move. The most commonly used response to this need is a digital packet radio system. Although digital packet radio networks have also been around for a while (since the advent of Amateur Radio's packet radio system), it has not been until recently that digital packet radio devices have become affordable and widely used.

There are several different digital packet radio networks in use, this report is primarily concerned with the Mobitex and DataTAC networks (by Ericsson and Motorola respectively). These two networks are the most widely used currently, so they are the most important networks to understand.

## Mobitex

Mobitex is a radio-based digital cellular communications network designed by Ericsson. It is based on packet switching technology and provides real-time, two-way communications for mobile users and for users at a fixed location. The Mobitex system is similar in concept to cellular mobile radio. [Eri96]

The communications links in the Mobitex network (for both mobile and fixed nodes) are based on public and open standards. This makes the integration of remote services and operations with central data processing services an relatively simple task. Some of the features of Mobitex are :

- wireless data communication and messaging
- packet-switched transmission
- roaming and store-and-forward functions
- open and closed user groups
- access to public data networks

Mobitex is currently being used in many fields including transportation, public safety, point-of-sale, and field sales and service. It uses encompass not only wireless messaging, database access and mobile offices but also for remote measurement. The most basic Mobitex application is as a replacement for voice-based private mobile radio dispatch systems, providing instantaneous two-way communication between vehicles and the central dispatching office. Advanced Mobitex applications also allow the host system to be integrated with the central data processing system to provide one-step data entry from order to invoice.

A Mobitex network is organized as a three-level hierarchy (see Figure 1.1): radio base stations (RBS), area exchanges (MOX) and main exchanges (MHX). At the top level there is also a Network Control Centre (NCC).

Coverage in a Mobitex network is provided by overlapping radio cells, each served by an intelligent base station, with users communicating with the closest such base station. With the use of an intelligently distributed routing scheme, a Mobitex node need only route data packets to the lowest network node common to the sender and receiver. This allows the base station to handle all local traffic between mobile terminals; passing only billing information up to higher levels. Base stations also co-operate to provide roaming between radio cells.

The area (MOX) and main exchanges (MHX) handle switching and routing and provide connection points for fixed terminals. The usual interface used by the Mobitex exchanges is X.25 (which is the ISO/CCITT standard for public packet-switched data networks). Billing information is also processed by the exchanges before being forwarded to the Network Control Centre.

Mobitex radio modems are available for portable, mobile and integrated applications from several manufacturers including Research In Motion and Ericsson. These radio modems support several different protocols to interface them to DTE equipment including extended AT, X.28 or native MASC protocols and

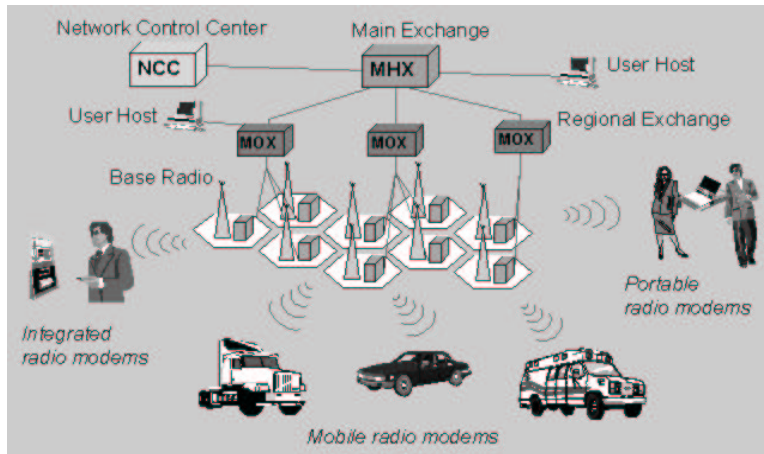


Figure 1.1: Mobitex Network Architecture

several more esoteric protocols. On fixed-side systems, the interface protocols typically include X.25, SNA 3270 and TCP/IP.

Mobitex networks in operation more than 20 countries on five continents including: Australia, Chile, France, Germany, Korea, Belgium, the Netherlands, the United Kingdom, the United States, Singapore, Finland, Norway, Sweden, Poland, and Canada (through Rogers Cantel). Although operated by many different network operators, all of these networks use the same protocols and operate according to the same specifications. Mobitex is a global standard, administered by the Mobitex Operators Association (MOA). Under MOA's leadership, European Mobitex operators have now introduced international roaming, to allow Mobitex users to continue to send and receive data outside their home countries.

### DataTAC

The DataTAC public network system was designed by Motorola's Wireless Data Group [Mot96a]. Motorola has had over 15 years' experience in the design and implementation of wireless data networks. DataTAC is currently the dominant standard in public data networks as it has more total subscribers than any other network. DataTAC networks have been installed around the world, in North America, Europe and in the Pacific Rim of Asia.

The DataTAC network is based on the concept of single frequency reuse. For each frequency channel, there can be several basestations within a given coverage area. This differs from a standard cellular system in that the cellular system has only one basestation audible on a particular frequency in any location. Thus the DataTAC system is not strictly a digital cellular system, although the concept is similar.

As with the Mobitex network, the DataTAC network has many possible

applications. Among them are :

- fleet dispatch and management
- remote database inquiries
- messaging
- wireless file transfer
- personal security

Through the use of an advanced RF protocol known as RD-LAP (Radio Data Link Access Procedure), the DataTAC network provides the highest data transmission rate available (either 19.2 kbps or 9.6 kbps depending on the country). This allows for fewer delays and faster response times for the user of the network, and for network operators, greater network capacity allowing more customers to be serviced.

The DataTAC network features extensive error detection and correction schemes to drastically decrease the probability of an undetected error caused by radio distortion to less than one in nine million. This helps ensure higher effective network throughput as there are fewer message re-transmittals necessary.

To allow for easier development of both new DataTAC conformant hardware and applications, Motorola has released many of the protocols as open standards. This facilitates the development of DataTAC products ranging from personal digital assistants (PDAs) to mobile data terminals and wireless modems. Thus the subscriber can choose the proper device to match the particular application at hand. There are several differing message formats and delivery options available, further allowing a developer to tailor their products to the specific application.

The DataTAC network infrastructure was designed to be a distributed and easily scalable architecture. This gives network operators the ability to upgrade their network as the number of subscribers increases without requiring massive replanning. The key components of the DataTAC infrastructure are :

- Network Management Center (NMC)
- Area Communications Controller (ACC)
- Data System Station (DSS)

All network administration, operation and maintenance functions are performed by the NMC. It is the control center of a DataTAC network and is based on an advanced client-server model.

The ACC consists of three components: the Radio Network Gateway (RNG), the Radio Network Controller (RNC) and the Communications Hub. The ACC is responsible for routing and switching all of the messages, and for providing a communications link between host computers and remote base stations. This communications is done using such protocols as TCP/IP and X.25 to allow

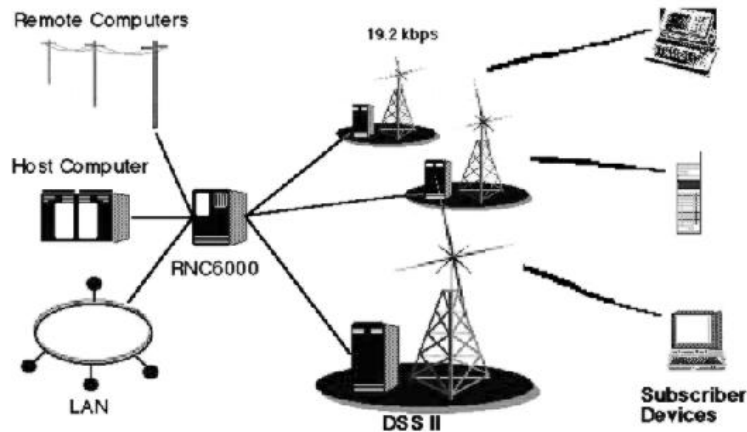


Figure 1.2: DataTAC Network Architecture  
Source: [Mot96a]

for standard communications tools to be used. The accounting and billing information for the users within an area is maintained by the ACC, along with control over roaming functions and authorizing devices that are attempting to use the network.

The DSS (or base station) apparatus is located at various sites that will provide maximal coverage to the subscribers. The function of the DSS is to modulate the messages coming from the network onto RF channels for transmittal to the subscriber units. A network operator must take care to employ sufficient DSS units to allow virtually seamless roaming from one base station to another, thus maximizing coverage for subscribers.

### 1.3 OSI Reference Model

The OSI Reference Model consists of seven layers of protocols that have differing functions in a digital networking system. As it is one of the best known reference models, it is useful to include a brief synopsis of it in this report as it will make the discussion considerably easier. The purpose of the model is to define logical building blocks to standardize computer intercommunication protocols. The seven layers are as follows (also shown in Figure 1.3) :

- Layer 1 — Physical Layer
- Layer 2 — Data-Link Layer
- Layer 3 — Network Layer
- Layer 4 — Transport Layer

- Layer 5 — Session Layer
- Layer 6 — Presentation Layer
- Layer 7 — Application Layer

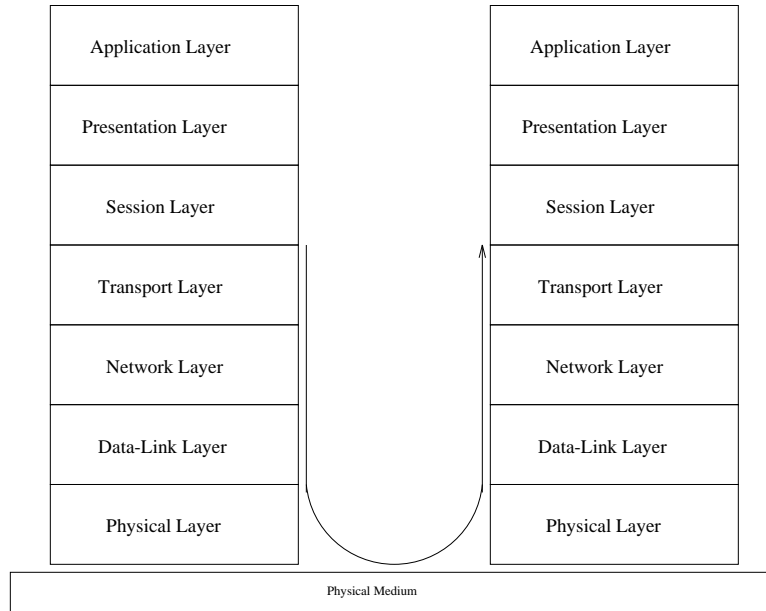


Figure 1.3: OSI Reference Model

The emphasis of this report is on Layer 2 — The Data-Link Layer, and on the implementation of a specific data-link protocol in an embedded system environment.

The Data-Link Layer has the responsibility of presenting the Network Layer with an error-free connection. This is done by extracting frames of data from the bit stream presented by the Physical Layer. To ensure an error-free connection, the Data-Link Layer must take care of retransmittals of frames that are deemed in error (generally by some sort of checksum or error-correction code), rejecting possible duplicate frames (if required) and possibly acknowledging incoming frames.

## Chapter 2

# Serial Protocol Description

The serial protocol that this report is concerned with is the NCL protocol which was developed to be used as the DTE—RPM communications protocol in DataTAC products. The physical medium is designed to be a RS-232 serial link, although it could be implemented with a different physical layer if that were required by an application.

The NCL protocol is a full duplex serial protocol with independent input and output streams. The protocol calls for a command–response architecture such that each command sent from the DTE (generally a personal computer) to the radio modem gets a response from the radio modem as soon as the contents of the command have been processed. Since the protocol is full duplex, the response being sent to the DTE is not necessarily in any way related to the command being sent from the DTE.

Each command is independent of each other such that the completion of a command (and the response thereto) is not necessary before the reception of the next command packet. This is important as some command packets take longer than others to complete (e.g. transmitting an RF packet will take longer than a status request). This necessitates queuing of the incoming commands such that all commands will be responded to. It also means that the responses may be sent to the DTE in a different order than in which the commands were received. To help distinguish which command a response corresponds to, there is a command identifier (known as the SDU tag) sent in the header of each command packet, and is returned in the corresponding response packet.

In addition to the nearly synchronous command/response packets, there are also asynchronous event packets sent from the radio modem to the DTE periodically. These event packets are generated whenever there is anything special to report (i.e. memory in the radio is full, etc.) as the DTE needs to be informed.

The structure of the serial protocol is shown in Figure 2.1. Particular note should be taken of the error detection and framing portions of the stream's definition.

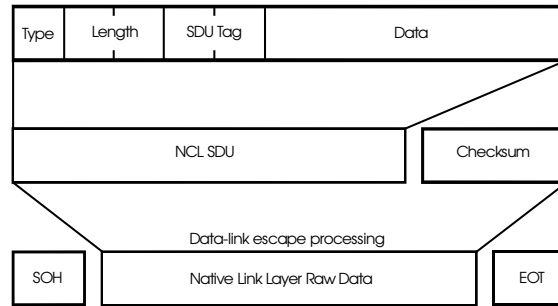


Figure 2.1: NCL Packet Structure  
Source: [Mot95]

## 2.1 Error Detection

As any data-link layer must ensure error-free data flow, serial protocols tend to use one of two methods of error control. The first is error detection, and the second is the use of error correction codes such as a BCH or Reed-Solomon code. The use of error correction codes provides a better protection against errors in the data stream, however at the price of transmitting the additional overhead of the ECC data.

For an error detection scheme, generally either a checksum or a CRC (cyclic redundancy code) is applied to the data, and is transmitted along with the data to be verified at the destination. This allows the destination entity to detect if there are any errors in the data as received, but requires the source entity to resend the entire packet of data if there is even a single bit error in the received packet.

The relative advantage of error *detection* methods over error correction methods is that the amount of extra data required to be sent is generally far smaller. On a fairly error-free connection, an error detection code is preferable to an error correction code as it requires less bandwidth to send the same data at the same rate. However, on fairly noisy (and error-prone) connections, error correction codes are preferable as they require fewer packet retransmittals as there is provision to automatically and transparently correct a finite number of errant bits.

The NCL protocol makes use of a rather simplistic form of error detection. Following the header and data payload, a checksum is included to ensure that the packet has arrived correctly at the other end of the serial link. This provides only a limited level of error control, but generally, an RS-232 serial line is nearly a perfect channel, so the lack of error correction capabilities is acceptable.

## 2.2 Data Framing

To allow for software flow control (i.e. XON/XOFF), and to delineate the beginning and end of the packets, there are special control characters used. This requires the escaping of these control characters when they occur in the data stream of a packet.

The beginning of a packet is indicated by an SOH character (ASCII value  $1_{16}$ ). This allows both a radio modem and a PC to ignore all garbage data on the serial link and determine when a new packet is arriving. To indicate the end of the packet an EOT (ASCII value  $4_{16}$ ) is sent. Thus the only valid data coming over the serial link is bounded by an SOH/EOT pair.

Unfortunately, the SOH, EOT, XON and XOFF characters can occur in the data stream. To ensure that they are interpreted as data rather than as control characters, they are escaped by adding a DLE character (ASCII value  $10_{16}$ ) and adding  $20_{16}$  to the offending character. The if the DLE character exists in the data stream, it must also be escaped. The resulting character translations are given in Table 2.1.

Control Character	ASCII Code	Escaped Character Sequence
SOH	\$01	\$10 \$21
EOT	\$04	\$10 \$24
DLE	\$10	\$10 \$30
XON	\$11	\$10 \$31
XOFF	\$13	\$10 \$33

Table 2.1: NCL Protocol Data Escaping

This escaping of data causes the data stream to be transparent to the layers above the data-link layer. This allows the upper layers to send whatever data required while maintaining the control features without the necessity of using extra hardware control lines.

## Chapter 3

# Design Restrictions

### 3.1 Embedded System Restrictions

Such a protocol is fairly simple to support in a system with virtually limitless memory and resources (such as a personal computer with multiple megabytes of memory), but the task of implementing it in an embedded system is somewhat more difficult. To understand this, it will be necessary to outline some of the restrictions imposed by embedded systems designs.

Perhaps the most important restriction imposed by an embedded system design is that the memory (especially RAM) is limited to a finite, and generally fairly small, amount. This is one of the most crucial limitations on a program as RAM must be used for all of the system data, and also for the stack(s) in use. As nearly all radio modems are coded around a real-time multithreaded operating system, the amount of RAM space available to the protocol interpreter is even more limited as the available RAM must be allocated not only to the protocol interpreter, but also to other system tasks and to the stacks corresponding to each task. Thus, any protocol interpreter must necessarily be as memory-lean as possible.

As many embedded system designs are used as battery-powered products, care must be taken to avoid making hardware requirements that are too power-hungry. An LED, for instance, although likely a very useful indicator when included in a product, will typically draw 10 to 20 mA when turned on, so if an LED is to be used in a battery-powered embedded system, the designer must take care to minimize the amortized current draw by means of flashing the LED at low duty cycles or other similar methods. Battery life is an often neglected issue when designing a product, with the results sometimes being a mad rush at the last minute to squeeze those last few milliamps of current draw out of the design. If it is taken into account from the beginning, the final stages of product design can be made a lot simpler.

Another sometimes problematic restriction in embedded system design is that the system is *embedded*. That is to say that there is no easy way to debug

or change the code once it has been programmed into the system without some special pre-planned method being included in the system. In most embedded systems, the code is nearly immutable, so the system designer needs to add functionality to accommodate for easier debugging, whether by adding a debugging stream that is fed to a separate processor, or by developing with the help of an in-circuit emulator rather than using an actual processor. Unlike on a personal computer there is usually no way to simply run a debugger on the code.

The serial port hardware included in many embedded system microprocessors often restricts the implementation of any serial protocol. Quite often the serial ports are severely lacking in capability, missing hardware control lines and lacking FIFO buffers on output (and more importantly input) data streams. This can pose problems when dealing with serial protocols as it makes the probability of dropped characters higher. A lack of hardware control lines (e.g. RTS, CTS, DTR, and DSR) can make implementation of serial protocols difficult at best.

In addition to these restrictions, the design is further restricted by being run under a multitasking real-time operating system.

## 3.2 Real-Time Characteristics

The embedded system in use in the radio modems is controlled by a real-time multitasking operating system, and there are further restrictions on any task running within the system.

Fortunately, in the area of RF communications, the real-time requirements are often that of a soft real-time system. That is to say that the external events do not have to be immediately dealt with in all cases. The design can therefore queue events and deal with them in some manner that can be optimized for speed, code size or even perceived importance to the end user.

To have a secure multitasking system, the memory used by one task should be isolated from all of the other tasks except where the data must be shared between tasks. This protects the inner data structures of each task from being corrupted by other tasks, and thus offers some degree of certainty that a given task's data is not corrupted. In some systems, the hardware provides some level of memory paging protection, which is the preferred method of protecting data. Where hardware protection is not available, the design of the tasks must attempt to isolate their own data as much as possible, although a catastrophic system failure or malicious code can still cause data destruction.

As the hardware may or may not include FIFO buffers for its serial ports, it becomes quite important that the operating system ensures that there are no serial port overruns. Characters must be removed from the serial port registers before the next one overruns it. This is usually more of a concern in the receiver half of the serial port as transmitting is totally under the control of the operating system, whereas the operating system has little or no control over when the system on the other end of the serial link sends characters. To ensure reception of all of the characters it is often necessary to streamline the serial port handling

code, and perhaps other interrupt or exception handling code such that no serial interrupts are lost.

To further ensure that serial interrupts are not lost, the serial interrupt handler task should be given a higher priority than the average task in the operating system. In many operating systems this is easily handled by using a hardware interrupt handler routine which typically has the highest possible priorities in the scheduler as it will preempt any running process while performing its duties. This will help ensure that the characters are pulled out of the serial port receiver buffers quickly, and possibly placed into a software-based serial port buffer for future processing.

The task (or subtask) that processes the received serial stream can then run at a lower task priority as the characters have not been lost, and the operating system can wait for an entire command packet to be received before processing the packets to allow the protocol command interpreter to be implemented in a perhaps more optimal manner. This also allows for better system performance as other possibly more important tasks to execute between the received characters, with the packets being processed in a burst of execution. As long as the software buffer is large enough to allow the system to empty it before it overflows, this type of software FIFO can be quite powerful.

## Chapter 4

# Optimizations and Compromises Needed

In order to make an effective Packet Assembler/Disassembler in a Real-Time Embedded System, it is necessary to make many optimizations and compromises in the design of the software. Some of these have potentially dangerous side effects, so the decision to implement some of these should not be taken lightly.

### 4.1 Limited Queuing of Packets

Since, in an embedded system, memory is often the most precious commodity, one of the first compromises generally taken is to limit the length of the queues of the packets to a finite size. This will decrease the amount of RAM tied up by the protocol interpreters. As the packet size is often rather large, (in the NCL protocol, the serial packets are up to 2048 bytes in length, and the radio packets are typically up to 512 bytes long), it is often a good idea to limit the number of packets to a small number such as four or five.

One of the consequences of making such a decision can be that the radio may not be able to keep up with the network under high network load as the packets may be received fast enough to fill the buffers before the radio modem has had a chance to parse them or pass them on to the DTE. This is especially more important as the radio networks switch to higher bit rates. Typical radio networks today operate at 4800 baud or perhaps 9600 baud, so currently most radio modems must be able to process the packets within only one or two milliseconds (one byte time), and the DTE bit rates are usually comparable to the radio network bit rates. This means that at current network speeds there should be few problems with buffer overruns, but in the near future, this will become progressively more of an important design issue.

## 4.2 Interrupt Handling

In most radio modem designs, there will be multiple interrupt sources, whether they be generated internally to the processor (such as a timer), or by external sources (such as a serial port). Deciding how to prioritize these interrupt sources is an important step to a working radio modem. Some of the priorities are pre-determined by the processor's manufacturers (typically the internally generated interrupts are pre-prioritized) leaving only some of them to the designer of the product. The method of masking out other interrupts while servicing another is yet another facet of the interrupt handling that is sometimes left to the designer. Some processor manufacturers make it simpler (not to mention less flexible) by enforcing a simple built-in priority and masking scheme such as the one found in Motorola's MC68HC11 microprocessor, whereas others leave most of the work up to the designer.

Since the radio modem is essentially an adapter between a radio network and, in most cases, a serial port on a personal computer, there are two main sources of interrupts to be considered: serial interrupts, and radio receiver interrupts. It is possible to in the most part neglect serial and radio transmit interrupts when considering asynchronous interrupts as they are under the control of the transmitting task(s) and as such are possible to be predicted. Although the serial interrupts are quite important to the radio modem's correct operation, the radio receive interrupts are crucial. Without the proper handling of these interrupts, a radio modem becomes practically pointless and useless.

One of the most important considerations for the system to be real-time is to ensure that no interrupts are missed. To this end, the interrupt handler routines should be made as lean as possible such that each interrupt is dealt with in the minimum amount of time possible, especially in a system that utilizes edge-triggered interrupts as opposed to level-triggered interrupts. If an interrupt handler takes too much time to execute, the result will likely be that other interrupts are missed.

There are two main methods of overcoming this problem: splitting the interrupt handler into two parts, and unmasking other interrupts in the offending interrupt handler. Both of these methods have their own particular problems and advantages, and are often both used within the same system for differing types of interrupt sources.

### Split the Interrupt Handler

Splitting the interrupt handler into two portions — a fast handler (that is vectored to on receiving an interrupt), and a slow handler (that is called from the operating system or a task periodically) — is a scheme that is often used for serial interrupt handlers. The fast handler in this case would take the characters from the serial port hardware (usually a UART), and place it in a buffer, then relinquish control of the processor. The slow handler would, as required, pull characters from the buffer and build packets from them, which is potentially time consuming as the completed packets must then be processed. The slow

handler can be implemented as a fairly low priority task in the system as the majority of the work to be done is after a complete packet has been received, which can be after receiving on the order of tens to hundreds of bytes from the DTE.

The disadvantages of using this method are somewhat obvious. There is the necessity of having a serial buffer which is at least large enough to hold a packet, or the slow handler must perform extraneous flow control or perhaps process the packets before they are completely received. This increases the complexity of the slow handler considerably. Additionally, the flow of code is obfuscated somewhat by severing the interrupt handler, making debugging a bit more difficult.

### **Unmask Interrupts**

The other method of ensuring that interrupts are not missed is to unmask the other interrupt sources while servicing an interrupt. This is a method that can be used with great success in a very large interrupt handler that does not lend itself well to being split up and is not time sensitive (as it will likely be interrupted). An example of this could be the handler for interrupts generated by the radio receiver. When an interrupt arrives from the radio receiver, it would most likely signify an incoming packet of data from the radio. After the data has been read from the radio receiver equipment (whether through a DSP or ASIC), some other interrupts can be unmasked while the received data is parsed. If this is not done and if the interrupt handler is kept as one thread of execution, chances are the handler will potentially take several milliseconds to deal with the received data, and in that time several serial characters or even timer interrupts could have been missed.

While unmasking the interrupts sounds like a rather simplistic procedure, it poses some rather important problems. If not done in a very careful and methodical manner, the act of unmasking interrupts can potentially cause stack overflows if, as is often the case in embedded systems designs, the allocated stack space is small (on the order of tens to hundreds of bytes). As each vectored call to an interrupt handler pushes the entire register context onto the stack (ranging in size from 9 bytes for Motorola's MC68HC11 microcontroller to about 40 bytes for Intel's i80188EX), it is easy to see how allowing nested interrupts could quickly fill or even overflow the limited stack space.

One plausible way to work around this problem would be to increase the stack space of each task (as it is not possible to predict which task will be running when an asynchronous event such as an interrupt occurs) such that it will be large enough to accommodate enough levels of interrupt handler nesting. However, this solution is often precluded by the low availability of RAM in the embedded system.

A perhaps more elegant solution would be to limit the degree of nesting allowed such that after a given point, interrupts will be masked regardless of how important they may be. This would give a known upper limit to the amount of extra stack space (above the normal requirements of the task's thread of

execution) is required in each task. This also has the additional side effect of allowing more control over the system in addition to allowing dynamic tracking of the the frequency of how deeply nested the interrupts are at any given point in time.

### 4.3 Single Queuing Path

In a radio modem, there can be several sources of data packets to be transmitted. Along with the data that the user wishes to transmit or receive, there will be the additional packets that must be sent and received by the radio code itself to maintain the connection. There is a great potential for confusion as to the source (or destination) of a particular packet. As there are two connections to be maintained (DTE to RPM, and RPM to RF Network), there is an even higher degree of confusion possible.

To remove some of the possible confusion, a single queuing path for each connection should be used. All packets that are to be sent out the connection whether from the radio code (internally generated) or from a user-controlled data source should be funnelled into the same queue with perhaps higher priority given to packets that are necessary to keep the connection operating correctly. This ensures that the code that dequeues packets and transmits them out the connection will be streamlined with only one thread of execution, rather than allowing many entry points to the transmitting routine creating a potentially difficult to maintain mess of code.

This method is the natural method of implementing a receive queue as the incoming stream *is* a single path anyways. At some point along the line, the transmit process needs to be a single path as only one packet can be transmitted at any point in time. Using a single queuing path from further up in the design removes the necessity of using an awkward synchronization method to ensure only one packet will be transmitted at a time.

## Chapter 5

# Debugging in an Embedded System

One of the most arduous tasks in the process of developing any embedded system product is debugging the code. As an embedded system does not always have a debugger available in the development environment, the developer must often find more ingenious methods of debugging the code.

### 5.1 Separate Debugging Stream

One of the most powerful debugging tools for an embedded system is a separate debugging stream sent to a host PC. This is analogous to the *printf* debugging method often used by C programmers. This method requires the embedded system to have an otherwise unused serial port that can be dedicated to the debugging stream (unless the developer chooses to somehow multiplex the debugging stream into the existing serial traffic).

With an appropriate decoding program written for the host PC, the debug stream can easily be compressed such that the radio modem need only transmit a few bytes (say a source file identifier and the line number of code) to represent a far longer string. With a few simple extensions to allow data values to be sent to the PC as well, the debugging stream can easily provide *printf*-like capabilities with only a small bandwidth requirement easily served by a serial line operating at 19200 or 9600 baud.

As the radio modem is implemented under a multi-tasking system, it may be a wise choice to differentiate between the output of the differing tasks in the debug output. This would make tracking inter-process problems a fair amount easier. If using a host PC with colour capabilities, using different colours for the tasks can make visual identification of the task a simple matter. Other methods of differentiating the tasks could be to display the task number on each output line, or perhaps to use separate windows for each task.

## 5.2 Simulation of the Embedded System

One of the most beneficial debugging aids for any embedded system is a simulated environment. Whether such an environment takes the form of a MS-DOS based simulator program or an in-circuit emulator, simulating the target embedded system is usually a time-saving endeavour.

Often the act of downloading code into the embedded system (especially in the early stages of development) can become a tedious if not risky procedure. The results of downloading incorrect code can sometimes be as drastic as requiring the ROM (usually FLASH ROM or EEPROM) to be physically removed from the embedded system and erased externally.

Having a simulation environment allows the developer to code and debug without the hassle of downloading the code into the “real thing”. Often the development of the embedded system hardware is done in parallel with much of the software development, so without a simulation environment, there could well be no debugging performed until the hardware were to be completed. There are usually far more PCs available during the development process than embedded system platforms, so having a PC-based simulation environment can allow a larger number of concurrent developers than could otherwise be utilized. This, in turn, allows for faster completion of the development of the final product, an aim that is in the best interests of most if not any research and development company.



# Appendix A

## Glossary

ASIC	Application-Specific Integrated Circuit
DCE	Data Circuit-Terminating Equipment (modem, printer, etc.)
DSP	Digital Signal Processor
DTE	Data Terminal Equipment (usually a computer)
MASC	Mobile Asynchronous Communications protocol
NCL	Native Control Language
RPM	Radio Packet Modem

# Bibliography

- [Eri96] Ericsson. *WIRELESS DATA — Dedicated wireless data network solutions*. Via URL <http://www.ericsson.com/EPI/BR/2mobitex.html>, 1996.
- [Fou82] R. J. Foulger. *Programming Embedded Microprocessors*. NCC Publications, Manchester, UK, 1982.
- [Mag93] Steve Maguire. *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*. Microsoft Press, Redmond, WA, USA, 1993.
- [MHR83] B. L. Meek, P. M. Heath, and N. J. Rushby, editors. *Guide to Good Programming Practice*. Ellis Horwood Limited, Chichester, UK, second edition, 1983.
- [Mot93] Motorola Wireless Data Group. *Mobile Asynchronous Communications (MASC) Interface R1.0 Reference Manual*. Via URL [http://www.mot.com/MIMS/WDG/TechSupport/pdf\\_docs/20.pdf/20.pdf](http://www.mot.com/MIMS/WDG/TechSupport/pdf_docs/20.pdf/20.pdf), July 1993.
- [Mot95] Motorola Wireless Data Group. *DataTAC Open Protocol Specifications: Native Control Language, Release 1.2*. Via URL [http://www.mot.com/MIMS/WDG/TechSupport/pdf\\_docs/8.pdf/8.pdf](http://www.mot.com/MIMS/WDG/TechSupport/pdf_docs/8.pdf/8.pdf), November 1995.
- [Mot96a] Motorola Wireless Data Group. *DataTAC Features and Benefits*. Via URL <http://www.mot.com/MIMS/WDG/products/datatac/features.html>, 1996.
- [Mot96b] Motorola Wireless Data Group. *Wireless Data Protocols*. Via URL <http://www.mot.com/MIMS/WDG/TechSupport/html.docs/protocols.html>, 1996.
- [Res96] Research In Motion, Ltd. *About RIM*. Via URL <http://www.rim.net/rim.htm>, 1996.
- [Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1989.