ANNOUNCEMENT OF CREATION OF FREE SOFTWARE TO READ WINLINK AND DISCUSSION OF HOW TO AVOID FUTURE TWO-DECADE DEBACLES

DATE: 08/30/2019
Gordon L. Gibby MD KX4Z

Introduction

Multiple filers with the FCC have made assertions that WINLINK cannot, or effectively cannot be read due to "effective encryption" or other supposed hindrances which have frequently been claimed to be violations of FCC Part 97.[1] [2] [3] [4] [5] Some pretty accusatory language has been used, which some writers may eventually wish to retract. It appears some language has already disappeared. The "encrypted" terminology prompted my first experiment, which many *never quite understood*, leading to some amazingly inaccurate comments. Subsequently John Huggins KX40 used brute force cut-and-paste on over-the-air captured packets to demonstrate decompression of a WINLINK email. This further report now details the successful creation of c-code software, running on a lowly 3B Raspberry pi, that can capture, rearrange, and decompress WINLINK FC EM messages (their only protocol). After that disclosure I discuss what could be done to avoid such an interminable argument about the next digital system to be developed in amateur radio.

A bit of history...

1 https://www.fcc.gov/ecfs/filing/1040761965970
2 https://ecfsapi.fcc.gov/file/10429199250117/FCC Letter Reply to Comments RM 11831.pdf
3 https://ecfsapi.fcc.gov/file/10503186789866/Rebuttal to Comments from Jame Whedbee.docx
4 https://ecfsapi.fcc.gov/file/1050452798372/Rebuttal to Comments from Philip Karn.docx
5 https://ecfsapi.fcc.gov/file/1052998397105/Dear FCC letter.docx

In actual fact, as previously explained, the WINLINK FC EM file transfer system is merely another protocol in the suite that Jean-Paul Roubelat F6FBB began sometime between 1986 and 1999, and the technical details governing such protocols have been available since ca. 1999 and are still available today for anyone to read, and comprehend. [6] There *never was*, and there *still isn't* any "encryption." That term has a specific meaning, and it isn't here. The improper usage of that term has been called out by a world expert.[7] There is just the same type of **data compression** that is used all around us, on our computers, music playing devices, internet browsers and just about anything that depends on storing or transmitting large amounts of data compared to the storage of speed available. *Who has not downloaded a compressed application, or zipped file?*

Also in actual fact, multiple different groups ***have done precisely the necessary reading and comprehension, to accomplish real code development***, that may not have been done by so many. The successful groups included:

- The Winlink Development Team[8] (source code generally not available)
- Paclink- Unix-Winlink Developers[9] [10] (appears to have source code)
- John Wiseman's G8BPQ software[11] (including all source code)
- PAT developers[12] (all open source and available[13])

It is quite notable that the PAT developers even went farther and created their own **new protocol**, FD transfer. [14] This development team chose to use end-to-end compression just as Jean-Paul Roubelat and the WINLINK team, but to use gzip instead of lzhuf. Both of these routines are ancient, and time-tested. Their code is fully open source and could be used by anyone to learn how to read any portion of WINLINK.

With all of this information constantly available for any person skilled in the art to utilize in order to build any desired reader or monitor system for WINLINK, it is surprising to me that years have gone by in which various persons complained that such reading was not possible, but did not construct their own system to suit their own desires.

Although I practice Anesthesiology, in my very early research days (30 years ago) I dabbled in some C-programming, and have done a small amount of Arduino programming. Approximately on August 1, I decided to re-study C-programming and to attempt to create the an initial reader software for WINLINK PACTOR messages, in the FC EM protocol, based on John Huggins's cut-and-paste success at decoding WINLINK messages. [15] [16] .

6 FBB (F6FBB) https://en.wikipedia.org/wiki/FBB_(F6FBB)
7 https://ecfsapi.fcc.gov/file/10513525129724/rm11831-rebuttal-to-rappaport.pdf
8 https://downloads.winlink.org/
9 http://paclink-unix.sourceforge.net/
10 https://sourceforge.net/projects/paclink-unix/
11 http://www.cantab.net/users/john.wiseman/Documents/Downloads.html
12 http://getpat.io/
13 https://github.com/la5nta/pat
14 https://github.com/la5nta/wl2k-go#gzip-experiment
15 https://ecfsapi.fcc.gov/file/108140794324824/KX4O_Dem onstration_OTA_Decoding_Addendum.pdf
16 https://ecfsapi.fcc.gov/file/1073182572879/KX4O_Demonstration_OTA_Winlink_Decoding.pdf

<u>I completed that task on the evening of August 25</u>, after 25 days of off-hours work, and successfully decoded an over-the-air transmitted message. ALL of the source code was provided on a national amateur radio forum.[17] I will soon make it available at https://www.qsl.net/nf4rc/ also. This project developed a simple, but working, c-coded reader system consisting of three separate executables, that together read a WINLINK (system) PACTOR (technique) message using FC EM compressed protocol.

The WINLINK message that was read was:

```
MID: TDJFWWUKTYEO
Date: 2019/08/25 22:32
From: SMTP:docvacuumtubes@gmail.com
To: KX4Z
Subject: Test13
Mbo: SMTP
Body: 130

12345767890
12345767890
12345767890
12345767890
12345767890
12345767890
12345767890
12345767890
12345767890
12345767890
```

The last error that had to be corrected in order to decompress this properly was to toss of the correct number of binary characters after reading the initial 0x02, <size-of-logical-packet>, initial packet, which followed the capture of the beginning of the data by detecting the 0x00,0x30,0x00 initial start of the file, which was preceded by a FA EM transfer proposal. Anyone who has studied Jean-Paul Roubelat's long-published work will understand those comments. Both I and Huggins have published lengthy explanations to teach how all this works.[18] [19] Once the code properly handled the startup point in the initial packet, the decode worked perfectly. These matters are further discussed in the section below that explains the code implementing the protocol that Jean-Paul Roubelat first pioneered more than 20 years ago. Every team since had likely had to grasp the same information that I had to. The GO group in particular rewrote *literally everything* open source, into an entire new cross-platform language. It is hard to imagine how this would not be helpful.

This free code is provided in good faith. It is my first successful solution to the decompression of WINLINK FC EM transmissions. It may even have software *bugs* as it was created by an amateur programmer. It works. If users are not satisfied with some portion or some performance --- it is up to

17 https://forums.qrz.com/index.php?threads/decode-off-the-air-winlink-message-request-for-programming-help.668470/page-10#post-5174896
18 https://ecfsapi.fcc.gov/file/10808597817982/ExParteCommunicationAug8.pdf
19 https://ecfsapi.fcc.gov/file/108140794324824/KX4O_Demonstration_OTA_Decoding_Addendum.pdf

YOU to now improve upon it. Amateur radio is not a spectator sport where we made demands on others, it is a collaborative group fulfilling the goals of Part 97.1 while enjoying radio and allowing others to likewise enjoy radio. ***We do not demand that others prove a "need" for their choices; we allow for others to choose which areas or techniques they want to explore.***

My estimate is that a person skilled in this art, and knowledgeable about the past developments, would have been able to create what I did, **in two working days**. I had to sit with a copy of Kernighan & Ritchie[20] *permanently at my side*, spending hours and hours discovering and correcting simple mistakes, using generic Windows Notepad and other simple editors that don't do *anything* to assist a programmer. At one point I discovered I could load my code into the Arduino free editor system, which at least matched up braces, to check for particular errors.

Two decades of arguing have gone past over the creation of just such a reading system.
Was it worth it, over only two days' of work?

Like any initial development by a novice in the field, this reading system and software are certainly not even close to what an expert would be able to optimize. I had to use three PACTOR modems, and one of them is roughly 30 years. I have no idea if they are in calibration, and my radios are old, and used and one of them is particularly inaccurate at frequencies. I had a hum problem from my signal, and large problems with signal level. My code is *highly inefficient* due to my inexperience, and littered with `printf()` statements from my debugging process (I don't even know how to use pre-processor directives). It put out a screen dump of 100 kilobytes just to capture a few minutes of text into the correct format as part of my debugging! If the SCS corporation had not come to my aid to show me how to read the serial port of the modem, I would never have succeeded. They have graciously agreed to allow their code to be widely used. Worse, the commonly published routines for `lzhuf` are ***processor dependent*** -- different platforms use different bit sizes for the c **`int.`** My code would not work for at least two days because of that problem, until I discovered Dan Planet had addressed it in 2009[21] and pointed to a machine-independent source code for lzhuf.[22] The version I provide here should work on a much wider variety of processors.

Limitations

This effort made no evaluation or optimization of methods to improve the PACTOR capture. There was no receiver diversity. There was no advantage taken of "repeat" packets, which could add significant improvement.[23] **If the monitor has a better signal to noise ratio than the intended recipient, they will be blessed with excess useful packets** -- but my novice effort ignores them, or worse. The development was done as modules of code interfacing with disk files to allow me a chance to succeed. It could be easily stitched all together (removing unnecessary intermediate files) and would produce the output in real time -- even on the $35 Raspberry Pi 3B, it operates in the blink of an eye. Certainly no research has been done yet into taking advantage of the sliding window features of lzhuf which possibly could allow re-lock after lost information. It does not yet handle multiple queued FC EM messages, but that would be an easy improvement. It does not yet handle gzip, utilized by the

20 https://en.wikipedia.org/wiki/The_C_Programming_Language
21 http://www.danplanet.com/blog/2009/11/09/winlink-1988/
22 https://people.cs.umu.se/isak/Snippets/lzhuf.c
23 Study this manual, apparently produced in 2012: https://www.p4dragon.com/download/Update_Info_DR7X00_Version_1_17_English.pdf

PAT group, who have furthered Roubelat's work, past what WINLINK created. All of this is work that could easily be started by any interested persons.

The WINLINK development team utilized exactly the same "logical packet" handling for all modes -- whether PACTOR or soundcard. **This allowed them to put all packet handling work at a higher level and not have different compression algorithm in every newly-developed modes.** The PAT group apparently felt similarly, (a fact I have yet to see acknowledged by the detractors of WINLINK) but moved to the *gzip* compression routine. My logical packet handling routine will work with *any* of the WINLINK techniques. All that has to be changed is to arrange for access to received radio packets, and store them in some similar format to what my term.c code does, and the remainder of the software should operate "as is."

EXPLANATION OF THE SOFTWARE

DISCLAIMER: Except for the portions of this software that were generously made available for use by SCS, all of this was written by a non-professional programmer, in the simplest fashion possible. Any professional would easily see much better ways to write all of this. The goal wasn't to produce professional polished code; the goal was to get something that worked, with as simple a layout as possible, with as much explanation as possible, that could then be used by any other interested persons.

Compiler: For all my work I used the publicly available gcc compiler. `gcc <source file> -o <desired compiled file name>` The experimenter will likely wish to learn how to use `hexdump`. (`hexdump -C <filename>`)

TERM.C

The first piece of software allows the pactor modem to be used to capture packets off the air and store them in a file (`/home/pi/capturefile`).

It is built on simple terminal software generously made available by SCS. The author wises the code to pass into public domain.[24] That portion of the software establishes the necessary baudrate of the pactor modem and establishes the keyboard & monitor connection to the pactor modem as a normal terminal emulator. Normally the modem will respond to the terminal with:

```
cmd:
```

This allows the user to give the pactor modem commands. The ones that I used were:

```
PMON HEX 1
PMON VERBOSE 3
PMON START
```

Following that the software will simply capture all text to the hard-coded filename /home/pi/capturefile. If the User inputs an exclamation point

!

followed by <ENTER> on their keyboard, the software will cease capture, close the capturefile and exit, placing a line

```
#EOF
```

at the end of the file. Abnormal terminations may end up with this not being placed.

After the reader has studied the SCS document on PMON, it should be easy to recognize when the packet counter is incrementing steadily and there are no missing packets. You can easily spot the FC

24 Personal communication, Peter Mack.

EM line for yourself, and soon you will be spotting the 00,30,00 that marks the beginning of the data file.

There is a persistent myth circulating that if you miss even one bit of the radio file, all of your decoding is for naught. Not only is this incorrect, but published work from Huggins demonstrated it to be false. The lzhuf algorithm stores the tree of the compression routine with the data. All works well until you have a missing packet, at which point you will miss a portion of the tree being constructed and subsequent portions of the message do not decode properly. I am not an expert on this, but because lzhuf is a sliding protocol, it is possible that there may be a recovery method after a sufficient amount of rebuilding. I simply don't know. But in any event, one way to deal with this issue is to use a diversity receiver network, which apparently has been done for scores of years by military and is even being done now to some extent by amateur radio contest managers.

Here is a segment of a captured transmission:

```
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  4.4, CRC: 52D2, FRCNT: 1, FRNR: 47
###PAYLOAD1: LEN: 59, TYPE: 0
###PAYLOAD2:
: KX4Z UURYXHAQS2AF 208 NF4AC@winlink.org //WL2K Test 40m P
###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  2.8, CRC: 34CF, FRCNT: 2, FRNR: 48
###PAYLOAD1: LEN: 59, TYPE: 0
###PAYLOAD2:
ACTOR send from EOC
;PM: KX4Z PI37QJTMHOG2 303 W4UFL@winlin
###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  2.2, CRC: 5F56, FRCNT: 3, FRNR: 49
###PAYLOAD1: LEN: 59, TYPE: 0
###PAYLOAD2:
k.org Re: //WL2K //p test send
FC EM UURYXHAQS2AF 237 208 0
###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  1.9, CRC: 4B8B, FRCNT: 0, FRNR: 50
###PAYLOAD1: LEN: 36, TYPE: 0
###PAYLOAD2:

FC EM PI37QJTMHOG2 363 303 0
F> C6

###PAYLOAD_END
```

```
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  2.4, CRC: 141F, FRCNT: 1, FRNR: 51
###PAYLOAD1: LEN: 0, TYPE: 0
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  1.9, CRC: 6480, FRCNT: 2, FRNR: 52
###PAYLOAD1: LEN: 0, TYPE: 0
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  7.1, CRC: 8A18, FRCNT: 1, FRNR: 53
###PAYLOAD1: LEN: 0, TYPE: 6
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  9.5, CRC: B883, FRCNT: 2, FRNR: 54
###PAYLOAD1: LEN: 0, TYPE: 6
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF: 10.0, CRC: A90A, FRCNT: 3, FRNR: 55
###PAYLOAD1: LEN: 0, TYPE: 6
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  9.6, CRC: 2D61, FRCNT: 0, FRNR: 56
###PAYLOAD1: LEN: 6, TYPE: 7
###PAYLOAD2:
FS YH

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  9.7, CRC: 8A18, FRCNT: 1, FRNR: 57
###PAYLOAD1: LEN: 0, TYPE: 6
```

```
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  4.8, CRC: BD9E, FRCNT: 1, FRNR: 58
###PAYLOAD1: LEN: 58, TYPE: 8
###PAYLOAD2:
2F,57,4C,32,4B,20,54,65,73,74,20,34,30,6D,20,50,41,43,54,4F,52,20,73,65,6E,64,20,66,72,6F,6D,20,4
5,4F,43,00,30,00,02,D0,E6,94,ED,00,00,00,EC,F5,7A,1C,6D,67,87,03,BD,E5,F2,75

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  2.8, CRC: 7B46, FRCNT: 2, FRNR: 59
###PAYLOAD1: LEN: 57, TYPE: 8
###PAYLOAD2:
39,BD,DE,FE,FA,EF,A5,99,CB,61,FB,40,3E,38,58,37,1B,CD,EF,17,77,63,C4,AE,A7,61,52,6F,0E,6E,
FE,A5,B5,AD,BF,FB,F7,F9,A5,57,B5,27,81,37,9F,2E,57,82,86,28,6E,71,65,3B,91,7C,ED

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  2.1, CRC: B345, FRCNT: 3, FRNR: 60
###PAYLOAD1: LEN: 55, TYPE: 8
###PAYLOAD2:
3F,0E,60,3F,C7,56,51,5E,7E,BE,1C,C9,2A,08,DF,56,37,5E,F7,2D,6E,22,0E,BD,EC,E7,9A,64,B6,F2,1
E,EF,B3,A8,1B,37,5C,6E,DF,C7,D8,AC,FE,DA,C7,9F,F5,4D,AE,65,04,B2,73,65,5A

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  2.2, CRC: 1B15, FRCNT: 0, FRNR: 61
###PAYLOAD1: LEN: 57, TYPE: 8
###PAYLOAD2:
F6,EF,CA,C6,6C,54,49,FE,57,62,70,FD,16,26,02,65,53,D9,71,CF,6C,A5,E7,42,36,DD,75,5C,2A,61,01
,24,86,8D,C8,73,A9,5C,02,09,3F,BA,24,07,8C,6A,2A,60,A9,23,95,16,4E,E8,91,86,9E

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  2.1, CRC: 090A, FRCNT: 1, FRNR: 62
###PAYLOAD1: LEN: 23, TYPE: 8
###PAYLOAD2:
05,17,4C,43,BF,E8,FA,00,CB,D1,7E,3E,10,00,E5,59,7C,18,05,2C,30,04,0B

###PAYLOAD_END
```

```
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 0, LSB: 0, dF:  2.1, CRC: 2684, FRCNT: 2, FRNR: 63
###PAYLOAD1: LEN: 0, TYPE: 0
###PAYLOAD2:

###PAYLOAD_END
þú
###PLISTEN: Level: 3:
###STATUS: SL: 3, CYC: 0, RQ: 0, REV: 1, LSB: 0, dF:  2.2, CRC: 7509, FRCNT: 3, FRNR: 64
###PAYLOAD1: LEN: 0, TYPE: 0
###PAYLOAD2:

###PAYLOAD_END
```

READCAPTURE.C

This program is the real guts of the system. It works its way serially through the `/home/pi/capturefile` and finds an FC EM (protocol C, encapsulated message) transfer proposal, and if found, captures the all-important unencoded file size from the following ASCII characters [PMON HEX 1 setting on the pactor modem]

An example of what the proposal might look like:

`FC EM WTLHRAMSQAFB 1826 983 0`

Version `readcapture825a.c` is enclosed in an appendix (first working version of the day August 25th).

The code opens the hardcoded filename `/home/pi/capturefile` for reading, then utilizes a nested set of loops to read in ASCII lines, character by character, to a maximum of 1923 lines or an EOF (end of file) marker. Obviously, this limit is arbitrary and can be replaced by a competent programmer. Lines from the PMON output are terminated (13,10) with an ASCII character having the decimal value 10, so the software uses this to detect end of line.

Each line that is found is first searched for "#EOF", which `term.c` placed on a normal exit. If that is reached and an FC EM proposal hasn't been found, the program exits with an error statement.

Each line is then searched for the ASCII characters "FC EM" and if found, steps past the following message identifier and then captures the size of the unencoded file, and converts it to long int format, and prints it out, stored in the software as variable UCFS. The flag "finishedfirstinput" (C `int`) is set to 1 to be used to get me out of this section of code.

Note: I had only simple editors with which to write all of this, so keeping track of matching braces was quite difficult, so braces are given alphabetic characters in comments, such as //Braces X to help me avoid software errors.

The WINLINK software, following very closely the conventions Jean-Paul Roubelat created, compresses the text to be transmitted using `lzhuf_1` software, prepends the unencoded file size and possibly a cyclic redundancy character check, followed by all of the file. That stream of characters is then divided into what I chose to term "*logical packets*." The size of each logical packet can be up to 255 characters, and each logical packet is prepended with the binary character 02, followed by one byte (0-255) giving the size of that logical packet.

Once the above has been accomplished by the WINLINK software, the resulting logical packets (the last of which may be somewhat shorter than the earlier ones) is then handed off to whatever protocol for actual radio transmission is going to be used. That protocol may be pactor, Winmor, Ardop, AX.25, or VARA or whatever is developed next. The radio transmission protocol then creates its own subdivisions of the binary information, which I chose to term "*radio packets*" and send them. Thus there are two different subdivisions of the data to be properly handled.

A file is opened into which the reconfigured binary characters will be placed to be later decompressed by a suitable executable from lzhuf_1. The first four binary characters (1 byte, 0-255) must contain the size of the unencoded file in little-endian format where (for example, in decimal) 1,256 is stored as "6521" -- the little end of the number goes into the first position. This is done for computer architecture historical reasons and has been that way for apparently > 30 years in some applications.

The next characters of the file depend on the particular form of lzhuf_1 that is chosen. WINLINK appears to use a version that expects an additional 4 characters which may be a cyclic redundancy check. I did not choose to deal with that, so I simply inserted 4bytes of binary 0. A better version of this software would likely handle this more appropriately and take advantage of the error checking opportunity.

Having the outputfile in order, and ready to receive characters, the program now turns back to the input file and proceeds to read all of the characters that have something to do with the binary compressed file into a long char array

```
char      pyload2array[10000];
```

A professional version of this software would far more likely dynamically allocate needed memory rather than this simple expedient of creating a fixed-size array, but this is NOT a professional piece of software.

The goal of this section of code is merely to capture all of the binary transmitted characters, which are transmitted in groups of "radio packets" and in the PMON output, are prepended by

```
###PAYLOAD2:
```

and other characters. It will be the job of a later section of the code to strip out the delimiters of the logical packets to get the data into final format to send to lzhuf_1

Therefore, a loop is utilized to read in a line of the PMON Output and hunt for any line that begins with ###PAYLOAD2: (discarding and moving past those which do not). If a ###PAYLOAD2 is found, the program reads in the NEXT line, and hunts for the characters that mark the beginning of a compressed binary file:

`00,30,00` [recall the PMON HEX 1 produces ASCII hexadecimal 2-character versions of binary characters, separated by commas]

If the 00,30,00 is found, an internal flag begofmesg is set to 1 so that software knows we have advanced beyond the beginning of the actual message, and begins to move characters from the input file into the output file. The first packet, and all subsequent packets, will begin with 02 followed by a one byte size-of-logical-packet, and all of this is store in the character array.

At this point the char array is completely filled with the logical packets of the binary compressed file.

A loop is then used to read in each character from the char array, find the 02 beginning of a logical packet, capture the next size-of-logical-packet byte (stored as an ASCII 2-character hex representation because I stuck with the PMON HEX 1 output). As a novice programmer, it took me several hours to figure out how to change that ASCII 2-character hexadecimal output into the software variable **`PacketChar.`** A professional programmer would chuckle at the mistakes I made trying to understand why C-library function `atoi()` didn't work the way I though it did.

The VERY FIRST logical packet has several characters that must be discarded, as John Huggins discovered. I do not yet know their function. It took me several iterations to find how to discard the proper number of characters from that very first packet and **cost me multiple days of effort** until I got that correct -- apparently the first 3 binary characters, each being 2 ASCII hexadecimals followed by a comma in PMON HEX 1 output. Recognize that you must know the size-of-logical-packet to know how many characters to move across because a "02," could be a real member of a packet, or it could be the start of the next logical packet -- only the **`PacketChar`** value helps you tell the difference.

All subsequent packets merely need to have the **`02`** packet starting character and the size-of-logical-packet stripped out.

This code is filled with debugging printf() statements to help me catch errors and a professional programmer would quickly use preprocessor directives to make this a ton better. ***It still runs blindingly fast even on a $35 raspberry pi!*** It is done in the blink of an eye in my experience. And it produces a lot of output to the screen. During development you can run it with `>screendump` to capture all the output to a file to study at leisure.

The output is stored in the hardcoded file name /home/pi/outputfile

LZHUFUNIV8.C

The code provided is from a source discovered by Dan Planet in 2009.   I added simple code so that it properly handles an 8-byte preamble, to be compatible with WINLINK procedures. However, i don't take any advantage of a cyclic redundancy check, which the WINLINK code might.

This code was required because the typical version of `lzhuf` on the Internet is not machine independent and doesn't work at all on a raspberry pi, which cost me several days of confusion.

Almost every version of this widespread routine works the same way.

```
lzhuf d file1 file 2
```
will decode file1 into file 2

```
lzhuf e file1 file2
```
will encode file1 into file2.

Most versions, if you simply type

```
lzhuf
```

will give you instructions as to the proper commands.   Execute this file on /home/pi/capturefile and send the output to a suitable file name and you should be able to read the email -- if you have received the packets properly.   After a bit of practice, you will learn how to read the status lines and you will begin to understand when you are missing a packet or so forth.   As stated above, almost NOTHING of this has been optimized.   There is enormous room for improvement, and it is such simple code that any student should be able to begin to make significant improvements from whatever point they start.

How to Avoid Such a Ridiculous Debacle Next Time

Having now conclusively demonstrated that even a novice programmer could, in a relatively short period of time, create a method to read over the air WINLINK (**system**) transmissions, I think it is important to address how to avoid this astonishing 2-decade arguing match the next time someone comes up with new amateur radio communications system.

Orders of magnitude more time has been spent ARGUING over reading these packets, than it took to write the code to do it.

And I suspect that the number of people who will ever download, compile and use the code that I wrote could be counted on one hand. Yet I did it, because it needed to be done, to put that argument to rest, and for openness in amateur radio. Discussion on QRZ almost ceased. Why it wasn't done by all of the brilliant people who have argued back and forth these issues for two decades, is a complete mystery to me. Perhaps some of them will explain that.

My experiences in the discussion on the QRZ forum gave me a much more jaded view of amateur radio operators. Despite the fact that I was the person creating the software so fervently demanded, I was the recipient of innumerable attacks. One person wrote a particularly scathing assessment of my character and actions and sent it to my personal email. I can only imagine what it has been like for friends of mine who have been involved in this for decades. *No wonder so little got accomplished.* Simple questions were difficult to get people to answer, time after time. John Huggins created a thread begging for other programmers to help us work on this long-demanded software development -- and no one helped, despite so many people in this debate claiming their expertise.[25] (Nevertheless, there is a gentleman in my area who may better my effort.) Is arguing of more value that achieving requested goals?

Everyone appeared to be demanding that the "other guy" do the work. That doesn't get us anywhere.

So it appears to me that some sort of "bright line" of precisely what is required to properly document new **techniques** [26](e.g. PACTOR or ARDOP) and now possibly also methods for monitoring **systems** (e.g., WINLINK or PAT), to avoid recurring decades-long vicious controversies.

TECHNIQUES

The previous "bright line" from the FCC was that all digital emissions had to be technically specified.

Multiple commenters have not understood the difference between codes (Baudot or ASCII) and a technique (Clover, G-Tor, Pactor, WINMOR, ARDOP). It would be a significant mistake to erase the memory of the solid precedent preserved in 97.309(a)(4) that **proprietary techniques** are well

25 https://forums.qrz.com/index.php?threads/decode-off-the-air-winlink-message-request-for-programming-help.668470/

26 From Part 97.309(a)(4) the word "technique" is used: (4) An amateur station transmitting a RTTY or data emission using a digital code specified in this paragraph may use any **technique** whose technical characteristics have been documented publicly, such as CLOVER, G-TOR, or PacTOR, for the purpose of facilitating communications. [emphasis added]

accepted if their technical specification has been published.[27] They have served amateur radio quite well At least in the case of proprietary CLOVER, a reader has been created by Wavecom, solidly demonstrating the adequacy of the description. The case for PACTOR is similar in that regard, because a competitor literally has created a 3rd party reader. (Wavecom). How one can then argue that the description *isn't* sufficient, is curious.

For PACTOR, certain filers seem unaware that the German government demanded that all modems be able to monitor (read) the transmissions of others, and that a truly ancient WA8DED "hostmode" has always been present, which provides even PACTOR communications in binary readable form, waiting for the programmer to exploit.[28] (Channel 4 of the hostmode is said to be PACTOR.) There have been some astonishing claims of inability to read PACTOR transmissions that appear to badly need retraction.

<u>Specification/Monitoring of Techniques</u>

Ron Kolarik has published several different versions of his demands for technique monitoring capability, possibly as a result of the negotiations process, beginning with language that appeared to prohibit even the use of any new hardware of any sort to read a totally new protocol.[29] That particular demand would have dramatically damaged the historical development of multiple techniques. RTTY, SLOWSCAN, AX-25 and other techniques would have been badly stunted, if the developers were not allowed to use custom hardware suitable for their development. [30] [31] Thus, that is a particularly poor choice for a monitoring requirement.

Mr. Kolarik appears to have modified his stance in at least three instances that I recently recognized, at times even using language that appeared to recognize the need to "own" both hardware and software[32] (implicitly recognizing that there might be a cost to obtaining the ability to monitor certain techniques), but at other times adopting a harder line of demanding freely available source code. The latter requirement would doom any amateurs developing a new advance and then trying to make it more widely available for use by creating a company to produce the product, in our globally competitive world with little protection of any released software. [33] One estimate of the cost to the American economy is $225-600 Billion--by just 1 nation!-- annually.[34] That makes not understanding the importance of trade secrets in the modern world, somewhat difficult to fathom. For just one well-known example, a commercially manufactured sound-card interface has been purchased by countless amateurs and led to dramatic expansions of all manner of digital techniques. Refusing to allow

---

27 97.309(a) (4) An amateur station transmitting a RTTY or data emission using a digital code specified in this paragraph may use any technique whose technical characteristics have been documented publicly, such as CLOVER, G-TOR, or PacTOR, for the purpose of facilitating communications.

28 Personal communication, Hans-Peter Helfert, DL6MAA

29 Requested text for 97.309 did not even allow for dedicated hardware: "and the protocol used can be be monitored, in it's entirety, by 3rd parties, with freely available open source software, " https://ecfsapi.fcc.gov/file/100918881206/PETITION FOR RULEMAKING.pdf

30 https://forums.qrz.com/index.php?threads/capturing-winlink-fc-em-pactor-messages-over-the-air-decoding.670930/page-2#post-5175239 noting HAL proprietary FSK modem for RTTY with equivalent today's dollars price of $2800

31 An advertisement for early Slow Scan proprietary hardware: https://forums.qrz.com/index.php?threads/capturing-winlink-fc-em-pactor-messages-over-the-air-decoding.670930/page-3#post-5175316

32 https://ecfsapi.fcc.gov/file/1071758880862/Reply to Gibby comments.pdf

33 https://www.fbi.gov/investigate/white-collar-crime/piracy-ip-theft

34 https://money.cnn.com/2018/03/23/technology/china-us-trump-tariffs-ip-theft/index.html

intellectual property protection would gravely damage forward progress.   For what logical reason would such a drastic standard not later move to transceivers themselves, thus endangering the newer computer-operated DSP transceiver systems?

A More Workable Standard
Mr. Kolariks goal of adequate technical specification of digital modes is laudable, if his chosen standard is not so attractive.   The WINLINK team has already published a suggested "bright line" based on existing US law for determining when the technical specification is adequate:

" The specification shall contain a written description of the invention [technique], and of the manner and process of making and using it, in such full, clear, concise, and exact terms as to enable any person skilled in the art to which it pertains, or with which it is most nearly connected, to make and use the same, and shall set forth the best mode contemplated by the inventor [author] or joint inventor [author] of carrying out the invention [technique]." Properly adapted [with substitutions], it lends itself well to provide an appropriate standard of disclosure to be added to §97.309(a)(4). [35]

> It was clear by referring to the original legal text 35 U.S.C 112 (a)[36] that the substitutions suggested were given right there inside the brackets, just as the words "with substitutions" were placed in the brackets.

That sounds like a useful "bright line" for determining when a **technique** has been sufficiently *technically specified*.   Therefore I support that definition of "technical specification." See for example, the incredibly detailed 22-page technical specification for WINMOR.[37] and equally impressive documentation for ARDOP.[38]

Existence of A Monitor
Apparently the German government went much farther, demanding that each publicly-available device be able to *monitor* emissions from similar devices for the user.

That seems like a useful way to silence interminable arguments, and therefore I would support a requirement that devices, or software for a technique, additionally have an ability (or interface) that allows monitoring of signals created by other such creations.   For PACTOR it is obviously already done.[39] For other techniques, it would seem relatively easy to provide some documented interface. See for example the interface specification of ARDOP.[40] [41]

SYSTEMS
But the WINLINK development is not a **technique**, it is a message forwarding **system**. [42] Until Sunday evening 8/25/2019 I do not believe there was published source code/software to monitor the data

35 https://ecfsapi.fcc.gov/file/104190189011279/RM-11831%20ARSFI%20Comment.pdf
36 https://www.law.cornell.edu/uscode/text/35/112
37 http://www.arrl.org/files/file/WINMOR.pdf   Debut: 2009 TAPR conference
38 https://winlink.org/sites/default/files/downloads/ardop_docs_pdf.zip
39 https://www.scs-ptc.com/en/Downloads.html
40 https://www.winlink.org/sites/default/files/downloads/ardop_tnc_host_mode_interface_spec.pdf
41 https://winlink.org/sites/default/files/downloads/ardop_docs_pdf.zip
42 FCC definition: (32) Message forwarding system. A group of amateur stations participating in a voluntary, cooperative, interactive arrangement where communications are sent from the control operator of an originating station to the control operator of one or more destination stations by one or more forwarding stations.

content of this **system** over the air, if you were not the sender or recipient or gateway operator. [43] The Winlink Development Team's production of arguably the worlds very first distributed, networked, multi-frequency, multi-technique receiver removed much of the apparent demand for such an over-the-air monitoring solution, and is obviously an incredibly effective alternative. It has been proven astonishingly effective for improving user-compliance with FCC regulations, demolishing arguments to the contrary.[44]

Like many other gateway systems operators, I have always been able to review messages that move through my gateway,[45] but my knowledge of the intricacies of all the requirements was certainly not as precise as that of Mr, Kolarik, and despite good intentions and checking many times, I'm sure that I missed transmissions that are now being flagged as "objectionable." That system was admirable, but did not seem to allow me to see messages moving through other gateways.

Looking at the larger picture, ham radio involves several facets of humans. There is a need to grasp something of technology. (This is RADIO, not tennis.) There is a need to stick to the rules. There is a need to be "innocent." But there is also a need to "cooperate" and "collaborate." It is not a spectator sport. All of us have some responsibility to exert some "pressure" on folks who are causing "issues." I've not seen that be very effective.....apparently ker-chunking of repeaters goes on interminably and from what I read, stuff goes on 75 meter phone that isn't printable. EGREGIOUS problems have been dealt with by the FCC to be sure -- and those get published for all to know. But how for the average ham to deal with improper WINLINK emails, or ker-chunking on the repeater, or cursing or other problems on 75 meter phone? Or ram-rodding over someone's signal on 40 or 20 meters? Gee....for the average ham, there doesn't seem to be much one can do, unless it is truly blatant.

The people who were particularly upset about WINLINK..... needed some sort of READER of the system. As previously explained on the level of techniques, every PACTOR modem included all that was needed to read transmissions and **claims to the contrary should be retracted quickly and specifically.** Some sort of method for monitoring other techniques would seem to be possible to be added over a reasonable length of time, as discussed above.

The benefit? Had Ron and Janis been reading the various messages running through the gateways (as they likely have been in the past few months) **a lot of this might be have been dealt with years ago, stopping this decades-long argument.** The internet-accessible, networked, distributed receiver has been incredibly effective**,** dropping objectionable emails by 2 orders of magnitude in a very short period of time. WINLINK is objectively the cleanest portion of all of amateur radio now.

While Mr. Kolarik's petition seemed to address issues at the level of technique, it seems that at the level of systems, there is also a need for new developers to be helpful in the creation of monitoring techniques, as the WINLINK team did, using the alternative of the distributed receiver. [46] My suggestion however, is that similar to the proposal by ARSFI for the "bright line" for disclosure of *techniques*, a similar requirement for *systems* to provide technical details of their operation so that monitoring can be accomplished might be required. [This was ***very clearly accomplished by Roubelat***

43 Call signs should have been easily monitored, meeting the FCC regulations.
44 https://ecfsapi.fcc.gov/file/10822196770221/ReAnalysisOfWinlinkObjectionableMessages.pdf
45 https://winlink.org/content/sysops_message_monitor System Operator Message Monitor
46 Viewing good-faith and extraordinary alternatives such as the winlink distributed, networked massive remote receiver viewer with undeserved suspicion does not help amateur radio.

and WINLINK, since I succeeded in writing a monitoring application by using the references which have been repeatedly discussed by me. [47]]

However, in addition to that, I suggest that systems either provide

- a distributed receiving system available to amateur radio operators at all normal times[48], similar in operation to what the WINLINK group provided with their 21-day history and mechanism for reporting issues.

Creating that may be extremely difficult for many groups and therefore the alternative below is included:

or

- at least *one working example* of a working monitor, **such as what I have just finished and disclose in this writing,** in any computer language and any computing platform of their choosing (as I have done), and <u>providing either compiled or source code for that system</u>, at their choice. (I provided free source code.)

Developers may certainly provide both.

<u>Responsibilities of the Message Gateway Station Operator</u>
There is a remaining matter dealing with responsibility for message content that passes through a message system forwarding (or originating, in the case of white-listed smtp: users) gateway. For repeaters, the FCC has made it clear that the liability of the control operator of the repeater is limited:

> 97.205 Repeater Station
> (g) The control operator of a repeater that retransmits inadvertently communications that violate the rules in this part is not accountable for the violative communications.

For message gateway operators/trustees, the situation is somewhat more murky, with ARRL Extra Class License manual giving unreferenced teaching that the amateur activating the gateway is considered the control operator.[49] I have previously asked the FCC for clarification of this matter,[50] since the ARRL published view properly deals with some, but not all, aspects of the situation. **What is quite clear is that beyond taking action when notified, the owner of the message gateway's liability should be limited for inadvertently transmitted communication that was obviously the responsibility of another amateur** (either by directly causing the transmission, or by allowing it by white-listing the creator and then retrieving their writing[51]). <u>Therefore I would ask the Commission to</u>

47 https://ecfsapi.fcc.gov/file/10808597817982/ExParteCommunicationAug8.pdf
48 Should the Internet fail, we will have much larger problems than whether or not we can reach the WINLINK distributed receiver.
49 This would not have been written without a good reason.
50 See detailed discussion in "Area 3" on page 5ff of: https://ecfsapi.fcc.gov/file/10730701023399/ResponseToRappaportJuly24Filing.pdf
51 There have been astonishgly ignorant comments about the operation of the WINLINK white-listing system. One key fact to note is that the ONLY person who can allow or retrieve or receive any communication from a white-listed internet address....is the amateur who white-listed it. Others simply cannot. This obviously explains who bears responsibility.

explicitly limit the liability of message gateway stations' operators. In view of the astonishing current documented WINLINK operators' collective compliance with FCC regulations (for which there is not any equal in all of amateur radio) this is a quite reasonable request.

Conclusion

I have now provided *free source code* that handles all of the decoding of all WINLINK system-level compression, demonstrated specifically for the PACTOR technique. **There should be no further argument as to the monitorability of WINLINK PACTOR --- it now all about radio technique, and optimization.** Years of productive research for interested amateurs, undergraduate, graduate students and engineering faculty can now be pursued, very productive work in power level control, diversity receiver systems, and further..... if can now move forward past these argument.....if amateur radio will rise to the challenge.

APPENDIX: term.c

```
/* SCS simple terminal
   Copyright 2005-2012 SCS GmbH & Co. KG, Hanau, Germany
   written by Peter Mack (peter.mack@scs-ptc.com)

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <linux/serial.h>

#include "lock.h"

// for use with a patched kernel module please comment out the following line
#define P4DRAGON // set the special baudrate

#define BAUDRATE B38400          // custom divisor only works if baud is set to 38400 !
//#define BAUDRATE          B115200         // this is the proper baudrate for the SCS PTC-IIIusb and a
good choice for the PTC-II modem series
```

```
char serdev[] = "/dev/ttyUSB0";
int       ser;
int       run;

void sigHandler (int sig)
{
   if (sig == SIGINT || sig == SIGTERM || sig == SIGQUIT || sig == SIGKILL || sig == SIGHUP)
   {
         run = 0;
   }
}

int main (void)
{
         fd_set  readfds, testfds;
         ssize_t rd;
         char buf[256];
         struct termios  new_termios, old_termios;
         struct serial_struct sstruct;
      FILE *fp1;   // file pointer for capture file


         printf ("SimpleTerm\n(c) 2005-2012 SCS GmbH & Co. KG, Hanau, Germany\npress CTRL-C
to end program\n");

         if (lock_device (serdev) < 0)
         {
                 // error
                 fprintf (stderr, "Could not lock %s\n", serdev);
                 return EXIT_FAILURE;
         }

         if ((ser = open (serdev, O_RDWR | O_NOCTTY)) == -1)
         {
                 // Error
                 fprintf (stderr, "Could not open %s\n", serdev);
                 unlock_device (serdev);
                 return EXIT_FAILURE;
         }

         /* save current port status */
         tcgetattr(ser, &old_termios);

         new_termios = old_termios;

         new_termios.c_cc[VTIME] = 0;
         new_termios.c_cc[VMIN] = 1;
```

```
        new_termios.c_iflag = 0;
        new_termios.c_iflag |= IGNBRK;
        new_termios.c_oflag = 0;
        new_termios.c_lflag = 0;
        new_termios.c_cflag |= (CS8 | CRTSCTS | CREAD | CLOCAL);
        new_termios.c_cflag &= ~(CSTOPB | PARENB | PARODD | HUPCL);

        cfsetispeed(&new_termios, BAUDRATE);
        cfsetospeed(&new_termios, BAUDRATE);

        tcsetattr(ser, TCSANOW, &new_termios);

#ifdef P4DRAGON
// special P4dragon handling (if not done by the kernel module!)

        // get serial_struct
        if (ioctl(ser, TIOCGSERIAL, &sstruct) < 0)
        {
           printf("Error: could not get comm ioctl\n");
           close (ser);
           unlock_device (serdev);
           exit(0);
        }

        // set custom divisor to get 829440 baud
        sstruct.custom_divisor = 29;
        sstruct.flags |= ASYNC_SPD_CUST;

        // set serial_struct
        if (ioctl(ser, TIOCSSERIAL, &sstruct) < 0)
        {
           printf("Error: could not set custom comm baud divisor\n");
           close (ser);
           unlock_device (serdev);
           exit(0);
        }
// end P4dragon handling
#endif

        signal (SIGALRM, sigHandler);
        signal (SIGINT, sigHandler);
        signal (SIGTERM, sigHandler);
        signal (SIGQUIT, sigHandler);
        signal (SIGKILL, sigHandler);
        signal (SIGHUP, sigHandler);

        FD_ZERO (&readfds);
```

```
   FD_SET (ser, &readfds);
   FD_SET (0, &readfds);

   write (ser, "\r", 1);

printf("\n\n\n TYPE !  (EXCLAMATION POINT)  TO EXIT CLEANLY \n\n\n");


   run = 1;

fp1 = fopen("/home/pi/capturefile", "a");
if (fp1==NULL) printf("Unable to open capture file /home/pi/capturefile \n");


   while (run)
   {
            testfds = readfds;
            if (select (FD_SETSIZE, &testfds, NULL, NULL, NULL) < 0)
            {
                     //fprintf (stderr, "\nSelect error!\n");
                     //return EXIT_FAILURE;
                     goto EXIT;
            }

            if (FD_ISSET (ser, &testfds))
            {
                     rd = read (ser, buf, 256);
                     write (1, buf, rd);
            buf[rd]=0;  // null terminate the string
            if (rd>0)    // Don't write out if nothing to write
                {
                 if (fputs(buf, fp1) != 0) printf ("Writing to file failed \n");
                }

            }

            if (FD_ISSET (0, &testfds))
            {
                     rd = read (0, buf, 256);
                     buf[rd - 1] = '\r';
                     write (ser, buf, rd);
            if (strchr(buf, 0x21) != NULL) // exlamation point hex 21
            {
             printf("Finishing capture \n");
             fclose(fp1);  // close the file we were writing text to
             run=0;  // this will cause the loop to move to EXIT:
             }
```

```
                }
        }

EXIT:
        close (ser);
        unlock_device (serdev);

        printf ("\n");

        return EXIT_SUCCESS;
}
```

APPENDIX: readcapture825a.c

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <termios.h>
#include <unistd.h>
#include <ctype.h>



void main()
{

FILE   *finputfile, *foutputfile;  //both in /home/pi
char    inputbuffer[1925];
int     linefillcounter;  // Counter for characters of one line read in from input file
int     outcounter;   // Counter for characters written out to the output file
char    *FCEMpointer;  // pointer to the start of fcempointer
char    *UncompressedFileSize; // pointer to the start of the uncompressed file size in ASCII

int     finishedfirstinput;  // track whether we have fouund the FC EM
long    UCFS, tempUCFS;    // long version of uncompressed file size
char LittleEndianString[20];  // string to get the 8 starting characters


size_t  asciisize;  // characters in the ascii representation of the uncompressed file size
char    FCbuffer[35];   // buffer for the FC EM line
char    FCEMinbinary[10]="FC EM";  // beginning of FC EM proposal
char    *Datapointer;   // pointer to pound-sign 02
char    buf2[10] ;  // buffer to grab things like   "84" from comma-delimited ascii hex
int     bytesread;  // counter to track how many characters we have read from this packet
int     PacketChar;  // #of characters in this packet, found right after the 0x02
int     endofline;  // set to 1 when we reach end of line from a packet
int     DataElement;  // an 8-bit data element to send to lzhuf
int     Left,Right;  // individual portions of the Data Element
char    *PAYLOAD2;   // pointer to the beginning of a PAYLOAD2 string


char    payload2array[10000];   // array to store the ASCII CHARS ("02,FA,BG....") of all payload 2's
int     LogicalPacketNr;       // counter to know which logical packet we are working on
int      logicalpacketend;      // 1 = we reached the end of this logical packet
int    RadioPacketNr;         // counter to know which radio packet we are working on
int    radiopacketend;          // 1 = we reached the end of this radio packet
int    radiofileend;            // 1 = we reached the end of the radio file
int      totalbytes;            // number of bytes packed into payload2array
int    begofmsg;                // 1 = we have found the 00 30 00 and are past that
                    // the importance being that in the first half of the program, from
                    // that point on, we are just going to shove all of the payload2
                    // message payloads onto the payload2array[] -- and let the second
                    // half figure it out how to strip out the 02's etc.
int    payload2index;
int      totalbytesread;        // total bytes out of array that have been read (where we are in the array)
int    nextonepayload2;         // way to keep track of the next one actually being the payload for 2


//  We will be reading in PMON output lines.
//  The maximum size of those lines occurs in Payload 2
//  The longest DATA field of a PACTOR III is 1276 characters in long-path mode.
//  One character = 1 byte transferred.
//  PMON adds a line start  #
//  and presents data as 2-character hexadecimals with a comma.
//  I think one hexadecimal is 2 bytes, but to be VERY safe, I'll assume using one hex characters
//  (stored as ASCII viewable hex characters)
//  for each of up to 1276 character, (638 2-character hex) plus a comma
//  for each of those 638.
//  Making 1276 +  638 +  add in 10 more = approx 1925

finishedfirstinput = 0  ;   // set the task marker for finding FC EM

finputfile = fopen("/home/pi/capturefile", "r");
if(finputfile==NULL) printf("Unable to open /home/pi/capturefile \n");

while (finishedfirstinput!= 1)   // Loop to handle lines until we find FC EM
{  // BRACES A

   // Read in a line
   linefillcounter = 0 ;

   do
     {
      inputbuffer[linefillcounter] = getc(finputfile);  //GET ONE CHAR
      if(inputbuffer[linefillcounter]==EOF)
             // this is not seeming to pick up the end of a file.....
          {
           printf("Error or reached end of file \n");
           linefillcounter=1923; // set to let us OUT
           finishedfirstinput= 1; // set to get us OUT
          }

      linefillcounter++;
     } while (  (inputbuffer[linefillcounter -1] != 10 ) && (linefillcounter<1923) );
      // SCS terminates line with 0d 0a    I search for the 0d.

   // AT THIS POINT: we have one line into inputbuffer[]


   inputbuffer[linefillcounter]=0 ;    // be CERTAIN it got terminated....
   // print out the line
   printf("Line:  %s \n",inputbuffer);

   // Check to make sure we haven't reached   #EOF
   if( strstr(inputbuffer,"#EOF") != NULL)
   {  // Braces X -- reached the end of the file before we ever found FCEM
     printf("Reached #EOF before finding FC transfer proposal ");
     fclose(finputfile);
     exit(0);

   }  // Braces X

   // Now look for FC EM

// Use this line if using PMON HEX 1 (ASCII readable output)
// I think it will actually also work for PMON HEX 2
// Because the FC proposal is sent in readable ASCII and thus even in
// binary mode it will be readable by human eyes.
   FCEMpointer =  strstr(inputbuffer,"FC EM");

// Use this line if using PMON HEX 2 (Binary output)
//   FCEMpointer  = strstr(inputbuffer, FCEMinbinary);
   if(FCEMpointer!=NULL)
     { //  BRACES H
     strncpy(FCbuffer,FCEMpointer,27);
     FCbuffer[27]=0;   // strncpy does not apparenlty 0 terminate...

     printf("\n\n\nFOUND FC EM:   %s \n",FCbuffer);
     finishedfirstinput=1;  // mark that we are done with the first chore
     // Because we FOUND the FC EM
     // Example:   FC EM WTLHRAMSQAFB 1826 983 0
     // Now find the start of the uncompressed file size.
      UncompressedFileSize  =( strchr(FCEMpointer+8, 0x20)) +1;
           // start somewhere inside the message ID
           // step 1 character past the space after the message ID
           // now we are pointing to the start of the ascii formated uncompressed file size
      asciisize = strspn(UncompressedFileSize, "0123456789");
        *(UncompressedFileSize + asciisize) = 0;  // null terminate it
        printf("Uncompressed File Size (ASCII) =   %s \n", UncompressedFileSize);
        UCFS = atol(UncompressedFileSize);
        printf("Int version of uncompressed file size = %ld \n",UCFS);
       }  // BRACES H -- end of if statement on FCEMpointer



} // BRACES A  End of while loop to read all lines until we find the FC EM


// Keep these open for now because we're advancing forward
//fclose(finputfile);
//printf("Capture File closed \n");

// First four bytes of the file are little endian (small side first) 4-byte
// unencoded file size in binary [0-255]
// see:
https://ecfsapi.fcc.gov/file/108140794324824/KX4O_Demonstration_OTA_Decoding_Addendum.pdf


// UCFS is already a long int.


tempUCFS = UCFS ;  // local long int version to be destroyed


// open a file to begin to write to, and make the
// REMEMBER:  file is actually made of 0-255 base 256 binary numbers
///          BYTES
// REMEMBER:  PMON Hex 1ngives us comma delimited, *hexadecimal representations*
// of binary numbers in the received binary.
// each individiual 0-255 binary character gets represented as TWO HEX
// ASCII-printable characters.    When Huggins was looking at the files
// he was looking with a viewer that converts to 2-char-hex printable
// representations also.
// What I have to put into the file for lzhuf is REAL BINARY.
// So the first four bytes are little-endian base 256 file size
// first byte =   0-255
// second byte =  gets us up by 256's to 64 kb -- enuf for my purposes
// then six binary values of ZERO befoe the file begins.
// Here is how Jean-Paul's encoder wrote it out:
//       textsize = ftell(infile);
//       ptr = (char *)&textsize;
//       for (i = 0 ; i < sizeof(textsize) ; i++)
//            crc_fputc(ptr[i], infile);
//            if (fwrite(&textsize, sizeof textsize, 1, outfile) < 1)
//              Error(wterr);   /* output size of text */

//

LittleEndianString[0]= (tempUCFS & 0x0ff);
LittleEndianString[1]= (tempUCFS & 0x00ff00) >>8;
LittleEndianString[2]= (tempUCFS & 0xff0000) >>16;
// three characters is enough to contain 64kb!!!

foutputfile = fopen("/home/pi/outputfile", "a");
if(finputfile==NULL) printf("Unable to open /home/pi/outputfile \n");

// C strings are initialized to 0 in all places
for(outcounter=0; outcounter<8; outcounter++)
   {
   if( fputc( (int)LittleEndianString[outcounter], foutputfile) ==EOF )
       printf("Unable to write size to /home/pi/outputfile \n");
   } // end of the outcounter loop


/*---------------NOW START READING LINES,
             HUNTING FOR THE START OF DATA SEGMENT (begoffile-->1)
             and packing the payload2 array of data elements   ---------------*/

// Now go back to the input file and hunt for that start of file indicator
// There should be a PAYLOAD 2 packet that starts either with 0x02 or has a
// line #02, in hexidecimal (PMON Hex 1) or binary (PMONHex 2)

finishedfirstinput = 0;    // re-use this job tracker
begofmsg=0;                // haven't found the start of the message yet
radiofileend = 0;          // haven't reached the end of the radio captured file yet
totalbytes = 0;         // number of bytes stored in payload2 array
radiopacketend= 0;         // end of a PAYLOAD2 packet line

// At this point we have found the FC EM... and now we need to pack the 1-dim
// array from the remaining PAYLOAD 2 lines in the input file until we reach the
// end of the message.

while (radiofileend!= 1)   // G Loop to handle lines from radio capture file forward of FC EM
{  // G while loop hunting for 00 30 00 02 (PMON HEX 1)  or <something related> (PMON HEX 2)

      nextonepayload2 = 0;  // set it to NOT at PAYLOAD2 at the moment
   // Read in a line
   linefillcounter = 0 ;

//Note that the line are filled with ASCII CAPITALS  BF,AC,etc
// I'll code for hexadecimal, 2-character comma delimited input,
// which should have a /n end  [0]


   // READ IN ONE LINE
   do
      {  // H do loop
       inputbuffer[linefillcounter] = getc(finputfile);  // get 1 char
       if(inputbuffer[linefillcounter]==EOF)
            {
             printf("Error or reached end of file without finding #02 \n");
             linefillcounter=1923; // set to let us OUT
             radiofileend= 1; // set to get us OUT
            }
                   if(feof(finputfile)!=0)
            {
             printf("feof returned end of file \n");
             linefillcounter=1923; // set to let us OUT
             radiofileend= 1; // set to get us OUT
            }

       linefillcounter++;
      }  // H do-loop
     while (  (inputbuffer[linefillcounter -1] != 10 ) && (linefillcounter<1923) );
    // END OF READING IN ONE C - LINE -- scs terminates lines with 0d 0a  search for the 0d


   // TERMINATE STRING AND PRINT FOR DEBUG
   inputbuffer[linefillcounter]=0 ;    // be CERTAIN it got terminated....
   // print out the line
   if (radiofileend != 1)  printf("=====Line:  %s \n",inputbuffer);
   if( strstr(inputbuffer,"#EOF") != NULL)
        {
         radiofileend=1;  // we reached the end of the file!!
         printf(" ***********WE FOUND FILE END**************  ");
        }

   //DONT GO ANY FARTHER IF WE ARE AT RADIO FILE END
   if(radiofileend==0)
   {  // GG loop to only evaluate while not at the end of the file


   // -----------TEST FOR PAYLOAD 2 LINE
   //See if we have found a PAYLOAD2 line:
   // need char *PAYLOAD2
   printf("Test for PAYLOAD2.  totalbytes = %d \n",totalbytes);
```

```
PAYLOAD2 = strstr(inputbuffer,"###PAYLOAD2:");
// This is terminated with 0a 0d in the SCS printout....so now read in
// AGAIN in order to get the actual line....

if(PAYLOAD2!=NULL)   // we FOUND a payload 2 line so read in again...
          {  // ZA brace to read in and deal with payload 2 if nonempty
                linefillcounter = 0 ;
           do
             {
             inputbuffer[linefillcounter] = getc(finputfile);  //GET ONE CHAR
             if(inputbuffer[linefillcounter]==EOF)
               // this is not seeming to pick up the end of a file.....
                 {
                 printf("Error or reached end of file \n");
                 linefillcounter=1923; // set to let us OUT
                 finishedfirstinput= 1; // set to get us OUT
                 }

             linefillcounter++;
             } while (  (inputbuffer[linefillcounter -1] != 10 ) && (linefillcounter<1923) );
               // SCS terminates line with 0d 0a    I search for the 0d.
               // AT THIS POINT: we have a possibly non-empty PAYLOAD2 line into inputbuffer[]

              inputbuffer[linefillcounter]=0 ;    // be CERTAIN it got terminated....
               // print out the line
               printf("PAYLOAD2 Line:  %s \n",inputbuffer);



            // IF we found it, then we can look for the 00,30,00,02 the next time we come here....
            // if it isn't there, then we gingerly move one line forward and check until we find it
            printf("\n\n---------ZA Loop, nextonepayload2==1 %s \n\n",inputbuffer);

           // now search for the 00,30,00, if we haven't yet found the begofmsg=1

            // Iinitialize Datapointer so this will work on the times through
            // after the first one (the one where we found the message start

            Datapointer = inputbuffer;  // start it off at the beginning
            payload2index = 0;          // index of characters taken out inputbuffer and shoved


            if(begofmsg==0)
            {  // braces AA   hunting for the start of the message
               Datapointer = strstr(inputbuffer, "00,30,00,");
               if(Datapointer!= NULL)
                  { // P loop because we FOUND start of file, first packet
```

```
                    printf("We found the start of a Datafile \n");
                    Datapointer=Datapointer + 9;  // point to where the FIRST 02 should be
                    begofmsg=1;   // mark that we found the beginning of the message
                              // thus now we start squirreling away the bytes
                    printf("string starting at begin of message:  %s \n",Datapointer);

                    } // end of P loop having found the unique start of file

               }  // end of braces AA hunting for the start ofthe message

          if(begofmsg==1)  // we have already previously found the beginning of message
                        // so now we are just sending everything over except the 0d's if we find one
               {  // braces AB packing away bytes from a logical packet
                 // pack all bytes to the end of the radio packet line (character 13)  into payload2 array
                 printf("Loop for unpacking PAYLOAD2 bytes: totalbytes: %d \n",totalbytes);

                 while( *(Datapointer+payload2index) !=13 )
                 {
                  printf("Char: %c from PAYLOAD2 index: %d  into arrray at %d \
n",*(Datapointer+payload2index), payload2index, totalbytes );
                  payload2array[totalbytes]= *(Datapointer+payload2index);
                  payload2index++;
                  totalbytes++;

                 }
                 // now we are at the end
                 radiopacketend=1;  // so now return to reading in lines and hunting for PAYLOAD2



               }  // end of braces AB packing away bytes from a logical packet
          // now reset the payload2 flag

          } // end of ZA loop dealing with a PAYLOAD2 line

      } // end of GG loop handling packing away a line

}  // end of G loop reading in lines until we hit radiofile end




fclose(finputfile);
printf("Closed the input file  \n");

/* --------------WE HAVE THE 1-dim Array Filled -----------------------------------*/
printf("payload2array:   %s \n\n", payload2array); // print out the array
printf("\n\n-------------\n1-Dim Array is filled with totalbytes =  %d \n\n\n",totalbytes);
totalbytesread=0;

// The array should now have 02, SIZE, ......[size# of comma delimieted]...02...repeat

// we SHOULD be positioned right before a 02 in ASCII

while(totalbytesread < (totalbytes-3))
  { // BA loop to go through the entire array


      //first move to the start of a 02 logical packet
         Datapointer=strstr(& (payload2array[totalbytesread]),"02");
         // Now we sit at the start of a logical packet
      // assume for the moment that bytesread=0;

         Datapointer=Datapointer+3; // now advanced to the size character
      totalbytesread = totalbytesread +3;
         buf2[0] = *Datapointer;  // get the left character
         buf2[1] = 0;  // null terminate it
         printf("Read the first size char:  %s \n",buf2);
         buf2[0]=tolower(buf2[0]); // atoi doesn't handle CAPITALS
         sscanf(buf2, "%x", &Left);  // get left as an int
         printf(" Left = %d \n",Left);
         printf(" Left Buffer:  %s, Value  %d  \n", buf2, Left);
         Datapointer++;   // increment to 2nd of size character
         buf2[0]= *Datapointer;  // get the right character
         buf2[1]= 0;   // null terminate it
         Datapointer=Datapointer + 2;  //move so it points to the start of the next character
         buf2[0]=tolower(buf2[0]);
         sscanf(buf2, "%x", &Right);  // get the right character of bytes in this packet
         printf(" Right Buffer:  %s, Value %d  \n", buf2, Right);
      PacketChar =  (Left * 16) + Right;
      printf("Number of bytes in this logical packet =   %d \n",PacketChar );
      totalbytesread=totalbytesread+3;


    //Because we now have the entire message in memory, we will
    //definitely be able to find the entire packet

      endofline=0;
      for(bytesread= 0; (bytesread<PacketChar) && (endofline!=1); bytesread++)
            { // Q loop to read in bytes

             // if we are on the VERY FIRST PACKET....we are going to NOT SEND
             // the first SIX dual-character hexadecimals.
            //
            buf2[0] = *Datapointer;  // get left character
               //   if (buf2[0] == 13) endofline=1 ;  // little d - reached end of line 0d 0a
            buf2[1] = 0 ; // null terminate it
            buf2[0] = tolower(buf2[0]);  // because atoi doesn't handle UPPER CASE
            printf("Bytes read  %d  FirstChar: %s \n",bytesread, buf2);
            sscanf(buf2, "%x", &Left);  // get left as int
            Datapointer++;  // increment to 2nd char of data element
            totalbytesread=totalbytesread+3;
            buf2[0]=*Datapointer;   // get the right character
            //  if (buf2[0] == 13) endofline=1; // little d - reached end of line
            buf2[1]= 0;   // null terminate it
            buf2[0] = tolower(buf2[0]);
            printf("Bytes read  %d  SecondChar: %s \n",bytesread, buf2);
            sscanf(buf2, "%x", &Right);  // get the 2nd char as an int
            DataElement= (Left * 16) + Right;  // REVERSE IF LITTLE ENDIAN NEEDED!!!!
                    // Data Element is a binary [0-255]
            printf("DataElement to write =  %x \n\n",DataElement );

            Datapointer++;  // move to where the comma oughta be
            buf2[0]=*Datapointer;
            //     if( buf2[0] == 13 ) endofline=1;
            Datapointer++;  // now we are pointing to the start of the next 2-char rep of a byte
            totalbytesread=totalbytesread+3;
            // WRITE OUT THE DATA ELEMENT
                // 18 caused us to skip sets of hexadecimals -- should have only skipped 18 sets!!!
                // so change this to 5....
                if(bytesread>5)  // to keep us from writing out the first 6 doubles [3 characters -- ??
CRC??]
                {
                if( fputc((int)DataElement, foutputfile) ==EOF )
                printf("Unable to write DataElement to /home/pi/outputfile \n");
                else printf("wrote %d to output file \n",DataElement);
                }

            } // End of Q loop to deal with ONE LOGICAL PACKET.
              // Now we have to move to the next 02 and throw it out....


    }  // end of BA loop to write out the entire array


  printf("We printed out the entire file \n\n");



fclose(foutputfile);
```

```
printf("output file closed \n");


}  /*  end of main */
```

APPENDIX: lzhufuniv8.c

Machine independent version, designed to work with an 8-byte preamble.
This version runs properly on a raspberry pi 3B
First four bytes are little-endian file size in binary.
Second four bytes are ignored.   (Added by me for compatibility with WINLINK versions)

```
/**************************************************************
    lzhuf.c
    written by Haruyasu Yoshizaki 1988/11/20
    some minor changes 1989/04/06
    comments translated by Haruhiko Okumura 1989/04/07
    getbit and getbyte modified 1990/03/23 by Paul Edwards
      so that they would work on machines where integers are
      not necessarily 16 bits (although ANSI guarantees a
      minimum of 16).  This program has compiled and run with
      no errors under Turbo C 2.0, Power C, and SAS/C 4.5
      (running on an IBM mainframe under MVS/XA 2.2).  Could
      people please use YYYY/MM/DD date format so that everyone
      in the world can know what format the date is in?
    external storage of filesize changed 1990/04/18 by Paul Edwards to
      Intel's "little endian" rather than a machine-dependant style so
      that files produced on one machine with lzhuf can be decoded on
      any other.  "little endian" style was chosen since lzhuf
      originated on PC's, and therefore they should dictate the
      standard.
    initialization of something predicting spaces changed 1990/04/22 by
      Paul Edwards so that when the compressed file is taken somewhere
      else, it will decode properly, without changing ascii spaces to
      ebcdic spaces.  This was done by changing the ' ' (space literal)
      to 0x20 (which is the far most likely character to occur, if you
      don't know what environment it will be running on.
**************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

FILE  *infile, *outfile;
static unsigned long int  textsize = 0, codesize = 0, printcount = 0;

char wterr[] = "Can't write.";
```

```
static void Error(char *message)
{
   printf("\n%s\n", message);
   exit(EXIT_FAILURE);
}

/********** LZSS compression **********/

#define N       4096    /* buffer size */
#define F       60  /* lookahead buffer size */
#define THRESHOLD   2
#define NIL     N   /* leaf of tree */

unsigned char
      text_buf[N + F - 1];
static int    match_position, match_length,
      lson[N + 1], rson[N + 257], dad[N + 1];

static void InitTree(void)  /* initialize trees */
{
   int  i;

   for (i = N + 1; i <= N + 256; i++)
      rson[i] = NIL;        /* root */
   for (i = 0; i < N; i++)
      dad[i] = NIL;         /* node */
}

static void InsertNode(int r)  /* insert to tree */
{
   int  i, p, cmp;
   unsigned char  *key;
   unsigned c;

   cmp = 1;
   key = &text_buf[r];
   p = N + 1 + key[0];
   rson[r] = lson[r] = NIL;
   match_length = 0;
   for ( ; ; ) {
      if (cmp >= 0) {
         if (rson[p] != NIL)
            p = rson[p];
         else {
            rson[p] = r;
            dad[r] = p;
            return;
            }
        } else {
            if (lson[p] != NIL)
                p = lson[p];
            else {
                lson[p] = r;
                dad[r] = p;
                return;
            }
        }
        for (i = 1; i < F; i++)
            if ((cmp = key[i] - text_buf[p + i]) != 0)
                break;
        if (i > THRESHOLD) {
            if (i > match_length) {
                match_position = ((r - p) & (N - 1)) - 1;
                if ((match_length = i) >= F)
                    break;
            }
            if (i == match_length) {
                if ((c = ((r - p) & (N-1)) - 1) < (unsigned)match_position) {
                    match_position = c;
                }
            }
        }
    }
    dad[r] = dad[p];
    lson[r] = lson[p];
    rson[r] = rson[p];
    dad[lson[p]] = r;
    dad[rson[p]] = r;
    if (rson[dad[p]] == p)
        rson[dad[p]] = r;
    else
        lson[dad[p]] = r;
    dad[p] = NIL; /* remove p */
}

static void DeleteNode(int p)  /* remove from tree */
{
    int  q;

    if (dad[p] == NIL)
        return;         /* not registered */
    if (rson[p] == NIL)
        q = lson[p];
    else
    if (lson[p] == NIL)
        q = rson[p];
    else {
        q = lson[p];
        if (rson[q] != NIL) {
            do {
                q = rson[q];
            } while (rson[q] != NIL);
            rson[dad[q]] = lson[q];
            dad[lson[q]] = dad[q];
            lson[q] = lson[p];
            dad[lson[p]] = q;
        }
        rson[q] = rson[p];
        dad[rson[p]] = q;
    }
    dad[q] = dad[p];
    if (rson[dad[p]] == p)
        rson[dad[p]] = q;
    else
        lson[dad[p]] = q;
    dad[p] = NIL;
}

/* Huffman coding */

#define N_CHAR      (256 - THRESHOLD + F)
                /* kinds of characters (character code = 0..N_CHAR-1) */
#define T       (N_CHAR * 2 - 1)    /* size of table */
#define R       (T - 1)         /* position of root */
#define MAX_FREQ    0x8000      /* updates tree when the */
typedef unsigned char uchar;


/* table for encoding and decoding the upper 6 bits of position */

/* for encoding */
uchar p_len[64] = {
    0x03, 0x04, 0x04, 0x04, 0x05, 0x05, 0x05, 0x05,
    0x05, 0x05, 0x05, 0x05, 0x06, 0x06, 0x06, 0x06,
    0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
    0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
    0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
    0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
    0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
    0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08
};
```

```
uchar p_code[64] = {
   0x00, 0x20, 0x30, 0x40, 0x50, 0x58, 0x60, 0x68,
   0x70, 0x78, 0x80, 0x88, 0x90, 0x94, 0x98, 0x9C,
   0xA0, 0xA4, 0xA8, 0xAC, 0xB0, 0xB4, 0xB8, 0xBC,
   0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC, 0xCE,
   0xD0, 0xD2, 0xD4, 0xD6, 0xD8, 0xDA, 0xDC, 0xDE,
   0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE,
   0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
   0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
};

/* for decoding */
uchar d_code[256] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
   0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
   0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
   0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
   0x0C, 0x0C, 0x0C, 0x0C, 0x0D, 0x0D, 0x0D, 0x0D,
   0x0E, 0x0E, 0x0E, 0x0E, 0x0F, 0x0F, 0x0F, 0x0F,
   0x10, 0x10, 0x10, 0x10, 0x11, 0x11, 0x11, 0x11,
   0x12, 0x12, 0x12, 0x12, 0x13, 0x13, 0x13, 0x13,
   0x14, 0x14, 0x14, 0x14, 0x15, 0x15, 0x15, 0x15,
   0x16, 0x16, 0x16, 0x16, 0x17, 0x17, 0x17, 0x17,
   0x18, 0x18, 0x19, 0x19, 0x1A, 0x1A, 0x1B, 0x1B,
   0x1C, 0x1C, 0x1D, 0x1D, 0x1E, 0x1E, 0x1F, 0x1F,
   0x20, 0x20, 0x21, 0x21, 0x22, 0x22, 0x23, 0x23,
   0x24, 0x24, 0x25, 0x25, 0x26, 0x26, 0x27, 0x27,
   0x28, 0x28, 0x29, 0x29, 0x2A, 0x2A, 0x2B, 0x2B,
   0x2C, 0x2C, 0x2D, 0x2D, 0x2E, 0x2E, 0x2F, 0x2F,
   0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
   0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
};
```

```
uchar d_len[256] = {
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
};

unsigned freq[T + 1]; /* frequency table */

int prnt[T + N_CHAR]; /* pointers to parent nodes, except for the */
          /* elements [T..T + N_CHAR - 1] which are used to get */
          /* the positions of leaves corresponding to the codes. */

int son[T];   /* pointers to child nodes (son[], son[] + 1) */

unsigned getbuf = 0;
uchar getlen = 0;
```

```
static int GetBit(void)    /* get one bit */
{
   unsigned i;

   while (getlen <= 8) {
      if ((int)(i = getc(infile)) < 0) i = 0;
      getbuf |= i << (8 - getlen);
      getlen += 8;
   }
   i = getbuf;
   getbuf <<= 1;
   getlen--;
   return (int)((i & 0x8000) >> 15);
}

static int GetByte(void)   /* get one byte */
{
   unsigned i;

   while (getlen <= 8) {
      if ((int)(i = getc(infile)) < 0) i = 0;
      getbuf |= i << (8 - getlen);
      getlen += 8;
   }
   i = getbuf;
   getbuf <<= 8;
   getlen -= 8;
   return (int)((i & 0xff00) >> 8);
}

unsigned putbuf = 0;
uchar putlen = 0;

static void Putcode(int l, unsigned c)     /* output c bits of code */
{
   putbuf |= c >> putlen;
   if ((putlen += l) >= 8) {
      if (putc(putbuf >> 8, outfile) == EOF) {
         Error(wterr);
      }
      if ((putlen -= 8) >= 8) {
         if (putc(putbuf, outfile) == EOF) {
            Error(wterr);
         }
         codesize += 2;
         putlen -= 8;
         putbuf = c << (l - putlen);
```

```
        } else {
            putbuf <<= 8;
            codesize++;
        }
    }
}


/* initialization of tree */

static void StartHuff(void)
{
    int i, j;

    for (i = 0; i < N_CHAR; i++) {
        freq[i] = 1;
        son[i] = i + T;
        prnt[i + T] = i;
    }
    i = 0; j = N_CHAR;
    while (j <= R) {
        freq[j] = freq[i] + freq[i + 1];
        son[j] = i;
        prnt[i] = prnt[i + 1] = j;
        i += 2; j++;
    }
    freq[T] = 0xffff;
    prnt[R] = 0;
}


/* reconstruction of tree */

static void reconst(void)
{
    int i, j, k;
    unsigned f, l;

    /* collect leaf nodes in the first half of the table */
    /* and replace the freq by (freq + 1) / 2. */
    j = 0;
    for (i = 0; i < T; i++) {
        if (son[i] >= T) {
            freq[j] = (freq[i] + 1) / 2;
            son[j] = son[i];
            j++;
        }
    }
    /* begin constructing tree by connecting sons */
    for (i = 0, j = N_CHAR; j < T; i += 2, j++) {
        k = i + 1;
        f = freq[j] = freq[i] + freq[k];
        for (k = j - 1; f < freq[k]; k--);
        k++;
        l = (j - k) * 2;
        memmove(&freq[k + 1], &freq[k], l);
        freq[k] = f;
        memmove(&son[k + 1], &son[k], l);
        son[k] = i;
    }
    /* connect prnt */
    for (i = 0; i < T; i++) {
        if ((k = son[i]) >= T) {
            prnt[k] = i;
        } else {
            prnt[k] = prnt[k + 1] = i;
        }
    }
}


/* increment frequency of given code by one, and update tree */

static void update(int c)
{
    int i, j, k, l;

    if (freq[R] == MAX_FREQ) {
        reconst();
    }
    c = prnt[c + T];
    do {
        k = ++freq[c];

        /* if the order is disturbed, exchange nodes */
        if ((unsigned)k > freq[l = c + 1]) {
            while ((unsigned)k > freq[++l]);
            l--;
            freq[c] = freq[l];
            freq[l] = k;

            i = son[c];
            prnt[i] = l;
            if (i < T) prnt[i + 1] = l;
            j = son[l];
            son[l] = i;

            prnt[j] = c;
            if (j < T) prnt[j + 1] = c;
            son[c] = j;

            c = l;
        }
    } while ((c = prnt[c]) != 0); /* repeat up to root */
}

unsigned code, len;

static void EncodeChar(unsigned c)
{
    unsigned i;
    int j, k;

    i = 0;
    j = 0;
    k = prnt[c + T];

    /* travel from leaf to root */
    do {
        i >>= 1;

        /* if node's address is odd-numbered, choose bigger brother node */
        if (k & 1) i += 0x8000;

        j++;
    } while ((k = prnt[k]) != R);
    Putcode(j, i);
    code = i;
    len = j;
    update(c);
}

static void EncodePosition(unsigned c)
{
    unsigned i;

    /* output upper 6 bits by table lookup */
    i = c >> 6;
    Putcode(p_len[i], (unsigned)p_code[i] << 8);
    /* output lower 6 bits verbatim */
    Putcode(6, (c & 0x3f) << 10);
}

static void EncodeEnd(void)
{
    if (putlen) {
        if (putc(putbuf >> 8, outfile) == EOF) {
            Error(wterr);
        }
        codesize++;
    }
}

static int DecodeChar(void)
{
    unsigned c;

    c = son[R];

    /* travel from root to leaf, */
    /* choosing the smaller child node (son[]) if the read bit is 0, */
    /* the bigger (son[]+1} if 1 */
    while (c < T) {
        c += GetBit();
        c = son[c];
    }
    c -= T;
    update(c);
    return (int)c;
}

static int DecodePosition(void)
{
    unsigned i, j, c;

    /* recover upper 6 bits from table */
    i = GetByte();
    c = (unsigned)d_code[i] << 6;
    j = d_len[i];

    /* read lower 6 bits verbatim */
    j -= 2;
    while (j--) {
        i = (i << 1) + GetBit();
    }
    return (int)(c | (i & 0x3f));
```

```
}

/* compression */

static void Encode(void)  /* compression */
{
   int  i, c, len, r, s, last_match_length;
   printf("This version uses 8 bytes of startup characters. ");

   fseek(infile, 0L, 2);
   textsize = ftell(infile);
   fputc((int)((textsize & 0xff)),outfile);
   fputc((int)((textsize & 0xff00) >> 8),outfile);
   fputc((int)((textsize & 0xff0000L) >> 16),outfile);
   fputc((int)((textsize & 0xff000000L) >> 24),outfile);
   fputc(0,outfile);
   fputc(0,outfile);
   fputc(0,outfile);
   fputc(0,outfile);
   if (ferror(outfile))
      Error(wterr);   /* output size of text */
   if (textsize == 0)
      return;
   rewind(infile);
   textsize = 0;          /* rewind and re-read */
   StartHuff();
   InitTree();
   s = 0;
   r = N - F;
   for (i = s; i < r; i++)
      text_buf[i] = 0x20;
   for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
      text_buf[r + len] = (unsigned char)c;
   textsize = len;
   for (i = 1; i <= F; i++)
      InsertNode(r - i);
   InsertNode(r);
   do {
      if (match_length > len)
         match_length = len;
      if (match_length <= THRESHOLD) {
         match_length = 1;
         EncodeChar(text_buf[r]);
      } else {
         EncodeChar(255 - THRESHOLD + match_length);
         EncodePosition(match_position);
      }
```

```
        last_match_length = match_length;
        for (i = 0; i < last_match_length &&
                (c = getc(infile)) != EOF; i++) {
            DeleteNode(s);
            text_buf[s] = (unsigned char)c;
            if (s < F - 1)
                text_buf[s + N] = (unsigned char)c;
            s = (s + 1) & (N - 1);
            r = (r + 1) & (N - 1);
            InsertNode(r);
        }
        if ((textsize += i) > printcount) {
            printf("%12ld\r", textsize);
            printcount += 1024;
        }
        while (i++ < last_match_length) {
            DeleteNode(s);
            s = (s + 1) & (N - 1);
            r = (r + 1) & (N - 1);
            if (--len) InsertNode(r);
        }
    } while (len > 0);
    EncodeEnd();
    printf("In : %ld bytes\n", textsize);
    printf("Out: %ld bytes\n", codesize);
    printf("Out/In: %.3f\n", 1.0 * codesize / textsize);
}

static void Decode(void)  /* recover */
{
    int  i, j, k, r, c;
    unsigned long int  count;
    int  test;
    printf("This version expects 8 characters prior to text ");

    textsize = (fgetc(infile));
    textsize |= (fgetc(infile) << 8);
    textsize |= (fgetc(infile) << 16);
    textsize |= (fgetc(infile) << 24);
    test = fgetc(infile);
    test = fgetc(infile);
    test = fgetc(infile);
    test = fgetc(infile);

    if (ferror(infile))
        Error("Can't read");  /* read size of text */
    if (textsize == 0)
```

```
        return;
    StartHuff();
    for (i = 0; i < N - F; i++)
        text_buf[i] = 0x20;
    r = N - F;
    for (count = 0; count < textsize; ) {
        c = DecodeChar();
        if (c < 256) {
            if (putc(c, outfile) == EOF) {
                Error(wterr);
            }
            text_buf[r++] = (unsigned char)c;
            r &= (N - 1);
            count++;
        } else {
            i = (r - DecodePosition() - 1) & (N - 1);
            j = c - 255 + THRESHOLD;
            for (k = 0; k < j; k++) {
                c = text_buf[(i + k) & (N - 1)];
                if (putc(c, outfile) == EOF) {
                    Error(wterr);
                }
                text_buf[r++] = (unsigned char)c;
                r &= (N - 1);
                count++;
            }
        }
        if (count > printcount) {
            printf("%12ld\r", count);
            printcount += 1024;
        }
    }
    printf("%12ld\n", count);
}

int main(int argc, char *argv[])
{
    char  *s;

    if (argc != 4) {
        printf("'lzhuf e file1 file2' encodes file1 into file2.\n"
               "'lzhuf d file2 file1' decodes file2 into file1.\n");
        return EXIT_FAILURE;
    }
    if ((s = argv[1], s[1] || strpbrk(s, "DEde") == NULL)
     || (s = argv[2], (infile = fopen(s, "rb")) == NULL)
     || (s = argv[3], (outfile = fopen(s, "wb")) == NULL)) {
        printf("??? %s\n", s);
        return EXIT_FAILURE;
    }
    if (toupper(*argv[1]) == 'E')
        Encode();
    else
        Decode();
    fclose(infile);
    fclose(outfile);
    return EXIT_SUCCESS;
}
```