Scratching An Itch, With And Without Help

Hugh Blemings - hugh@linuxcare.com.au

18th December 2000

Abstract

This paper examines open source projects with particular reference to the different challenges that come from working with and without hardware vendor support. The principle "without" case examined is gnokii[1], the "with" case is the process of writing Linux kernel drivers for the Keyspan range of USB to serial adapters[2]. The reverse engineering to achieve the former and discussions with the vendor to do the latter will be presented. The gentle reader is warned that raw source code (often uncommented), schematics and hand written notes may be presented.

Contents

1 Introduction

This paper examines a number of open source projects with a particular reference to the different challenges that come from working with and without hardware vendor support.

Much of the focus will be on gnokii, the without case, as it is somewhat more interesting from a technical standpoint as a degree of reverse engineering and/or detective work was required.

[1] `http://gnokii.org`
[2] `http://www.linuxcare.com.au/hugh/keyspan`

A with case will also be discussed as it provides a nice example of how easy development can be when the vendor in question has a clue. Some ideas on how you can help them to be clueful are included too.

2 Scratching an itch

Scratching an itch has nothing to do with the personal hygiene of hackers or writers of open source software but rather describes how many Open Source projects come about. ERIC S. RAYMOND put it thus

> 'Every good work of software starts by scratching a developer's personal itch.'

...and this describes it pretty well. Much of the open source software that is out there comes about because the author needs it to get a particular task done and there is either no tool to do the job or existing offerings fail to meet the standards of the individual concerned.

On the upside this usually means you end up with a tool that is really pretty good -the author(s) are using on a daily basis and probably have similarly high standards to your own, they don't want the thing to crash, they want it to run on modest hardware and they never read the documentation so don't write any. The latter actually isn't so great but manuals are for wimps anyway...

The downside is that if you have a particularly unique or boring requirement it's quite possible no one has written a tool to address it.

3 Doing it without help - gnokii

gnokii provides tools for doing various things including modem emulation for most recent model Nokia GSM mobile phones.

Nokia have their own suite - "Nokia Cellular Data Suite" (NCDS) which provides this functionality on the Windows platform. They have a virtual serial driver that sits between the standard Windows serial layer and the phone and translates between Hayes AT style modem commands and the proprietary protocol used by their phones (the so called FBUS or MBUS protocols).

The "scratching an itch" angle here was that I had a Nokia 3810 which I quite regularly used with a Toshiba laptop running Windows. I eventually started running Linux on the laptop (I'd been running it for a couple of years on my desktop machine). Having done so I couldn't use the 3810 under Linux at all and had no means of mobile 'net access. The trembles from this were terrible.

The project initially supported the then current 3810/3110/8110 model series - another project had sprung up (but not generated any code) to support the (then) new 6110 series. After a short dialogue we combined the two figuring (correctly as it happened) that much of the higher level code would be common anyway. PAVEL JANIK who was based in Brno, Czech Republic became a co-author having contributed the majority of the 6110 series code.

3.1 Asking nicely for help

During the course of the gnokii project there were a couple of exchanges with people at Nokia. In the end I was able to make contact with the chap responsible for Nokia's own NCDS. He was quite positive about helping out, the intent being that Nokia's internal documentation on the FBUS protocol would be made available to us.

Unfortunately on 11 June 1999 it became clear it wasn't to be...

> ...been a while but anyway now I have some information. Generally idea is OK from our side. However the problem arises with the open source model. We can give you protocol specs but we couldn't let you distribute the implementation in source code format. This is due to the fact that even if we gave you only necessary protocol spec parts, it is very easy to re-engineer rest of the bus if you see the source, and thus bus gives access to functions like SIM-locks etc. in the phone which in turn poses high risk to us; if someone then implements software to mess around with these kind of "dangerous" functionality with our handsets and causes harm to him/herself or other people using the software, we are very likely to be held liable on that and that risk we cannot take as a company.
>
> So, tough issue, you'll get our support but with the limitations that you could distribute your software only in binary form, not in source code form. Let me see how do you see this.

Reading between these lines and from the context of earlier discussions my sense is that the legal team put the stoppers on any sort of collaboration. We discussed the idea of a binary only release on the mailing list and the consensus was “No way - we'll do it ourselves.”

As it turned out, Nokia's concerns were partially valid - someone did work out how to remove SIM locks on some phones using their protocol. The irony being that whether or not they gave us a hand with the protocol this would have happened anyway. Hey ho...

3.2 Working it out anyway

By the time we got the “sorry, not open source” response there had actually been several releases of gnokii and we were well underway in understanding the protocols in use. Their assistance still would have been of help as at this point we hadn't got data calls going or some of the more exotic features sorted out. To understand what we had to work out a little explanation of how the FBUS protocol and GSM phones operate will assist. Note that what follows is what we now know, how we came to know this is the subject of section 4.

Like most modern GSM phones, the 3110/3810/8110 series allow you to store phonebook entries, send and receive Short Message Service (SMS) messages, place and receive data and fax calls and so forth. These facilities are available remotely by the phone's serial interface using a cable that translates between RS-232 voltage levels and TTL/CMOS levels used by the phone. Signalling wise, the FBUS protocols run at a 115,200 baud, 8-N-1. There are a couple of different variants on the protocol itself depending on the phone model series in question. What follows relates to the 3110/3810/8110 series as these are the ones I was working on. Most of the actions over FBUS follow a command/response sort of model. The phone will also send some messages unsolicited in response to events occurring on the network, such as incoming calls, SMS messages etc.

Fax and data calls on a GSM network actually send binary data over the air interface using the Radio Link Protocol (RLP). Some munging of the data stream is done depending on the other end of the link. If the other end is an ISDN connection the data stays in the digital domain the whole time. If the other end is an analogue modem or fax then a device at the far end of the network modulates/demodulates the RLP data into the baseband signals sent to the codec at the end of the remote end's piece of copper.[3]

When run in data or fax mode (we don't support the latter) the phone streams raw RLP frames off the GSM network down the interface and similarly expects RLP frames to be sent to it. To get data calls going we had to write our own RLP implementation from the relevant ETSI specifications.[4]No small task but the open development model prevailed.[5]

4 A reverse engineering 101

What is reverse engineering ? In broad terms it is the process of finding out how something works in detail to the point where it can be replicated or fully understood without recourse to the engineering and specification documents used during its creation. For most practical cases it amounts to pulling the thing apart either physically or virtually, often (and preferably :) both.

Reverse engineering has become one of the hotter topics for Open Source development, particularly to the extent that it touches that most contentious of areas - intellectual property. Regrettably in many countries Big Business has successfully lobbied for it to be declared illegal or have created sufficient fuss that it appears to be illegal to the casual observer. Happily, like much of Europe, Australia has been largely free of such wrangling.

In the context of hardware systems, one of the most common forms of reverse engineering involves running the piece of equipment unmodified in its native environment and watching what is going on through suitable monitoring equipment. In the context of gnokii, this involved leaving the phone connected to a Windows 95 PC running Nokia's software and watching (or *sniffing)* what was sent over the cable between the two.

Another common, though for various reasons more controversial method of reverse engineering is the process of examining the software itself - disassembling the binary image into (typically) assembler source code and seeing what is happening from there. Often this approach is applied in tandem with sniffing to allow a more complete understanding of the system. In the case of gnokii I chose to leave the binaries alone as I figured (a) it would take longer to understand the mess of Windows system calls than to decode the protocol, (b) I didn't have a disassembler handy and (c) I have an intense dislike for Intel assembler.

[3]“GSM Switching, Services and Protocols” by Eberspacher and Vogel, Published by Wiley 1999 is an at times dry but informative reference on GSM. ISBN 0-471-98278-4.

[4]The ETSI website `http://www.etsi.org` has most of the standards available online. Unusually they're free PDF downloads.

[5]Thanks to Chris Kemp who did much of this code using a Delphi implementation put together by Serge Odinokov and the ETSI standards as a reference.

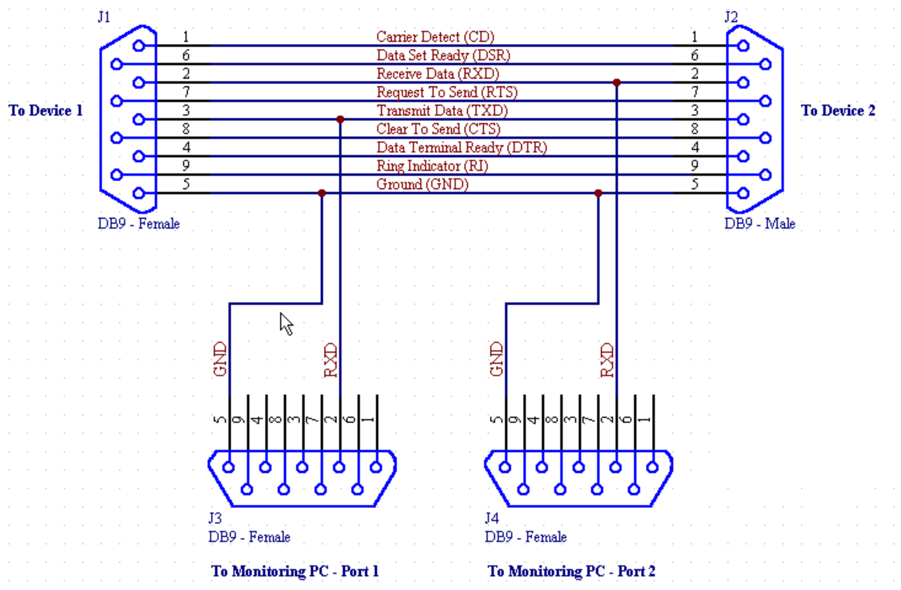

Figure 1: Simple serial “sniffer” cable.

In the next few sections I’ll outline the reverse engineering techniques applied to the 3810 series of phones under gnokii. What follows is by no means definitive but did the trick in this case.

4.1 Making a serial sniffer cable

Figure 1 shows a basic serial sniffer cable that makes use of the receive data inputs on two ports on the “sniffer” machine to monitor data emerging from the transmit data outputs on each of the two devices. Like “real” protocol analysers, a more involved cable will allow you to monitor handshaking signals and the like by feeding these signals into input handshaking pins on the “sniffer” machine.

4.2 Working out baud rates, character lengths etc.

Before any sort of reliable sniffing can take place you need to know the baud rate, character length, parity settings and so forth. With a bit of practice you can do this by feel to some degree - looking at the output from whatever software sniffer you are using and seeing if the data is consistent and basically looks “sane” will often tell you if you’ve got it right. If you want to get an idea of what a baud rate mismatch looks like connect two serial ports together through a null modem, set them to different rates and type...

If you have access to an oscilloscope you can take a look at the serial data stream in raw form and ascertain the bit time from there. Your baud rate will then be 1/t where t is the bit time in seconds (for 9600 baud, t will be around 104uS) Examining the data in this manner will allow you to make a good guess about character length (7/8/9 bits), parity etc. as well.[6].

4.3 Serial sniffing software

Once a suitable cable has been put together, some software is required to allow the data to be collected and displayed. At the time I was working on gnokii there weren’t any suitable open source tools around so some suitable ‘C’ code was cobbled together for the purpose.[7] This software would output data flowing in one direction

[6]Don’t assume the world revolves around PC baud rates which are ultimately derived from a 4.9152 or 9.8304MHz crystal which drives the PC UART’s 16x reference, particularly if the equipment in question isn’t normally interfaced with a PC. MIDI for example runs at 31.25kbaud which can’t be accurately derived from a conventional PC UART. If on examination with a CRO you’re getting bizzare bit times, it’s worth looking at the hardware in question and/or a schematic if possible. If there is a device you can identify as a UART on the board and it’s near a weirdass crystal then maybe it really is 650.2kbaud, 9 data bits, even parity as one bit of gear I worked on proved to be...

[7]There is now a rather more elegant piece of sniffing software available - take a look at Grahame Kelly’s serial_sniffer which he used to reverse engineer the protocol used by the Nikon Coolpix digital camera. `http://www.ibiblio.org/pub/Linux/apps/misc/serial_sniffer-0.5.tar.gz`

Byte number	Description
1	Synchronisation byte - 0x03 during RLP mode, 0x04 in “normal” mode
2	Message length (n) including type byte but not checksum
3	Message type
n	Message data (if any)
n+2	Checksum - exclusive OR of entire message

Table 1: Message format for 3110/3810/8110 protocol, phone to PC

enclosed in square brackets, data in the other in braces with the hexadecimal byte and ASCII character (if valid) in between them.

```
{01 }{02 }{4aJ}{16 }{5f_}[04 ][02 ][4aJ][0e ][42B][04 ][05 ][4bK][15 ][01 ][03 ]
[02 ][5f_]{01 }{02 }{4bK}{1d }{55U}{01 }{02 }{3f?}{17 }{2b+}[04 ][02 ][3f?][0f ]
[366][04 ][1b ][41A][16 ][02 ][00 ][00 ][00 ][00 ][00 ][00 ][a7 ][01 ][01 ][00 ]
[0c ][2b+][366][311][344][311][311][399][399][300][300][300][300][00 ][f9 ]{01 }
```

The primitive code I put together had a number of limitations, most notable was that it didn’t really address time alignment of the received data - thus it wasn’t clear from the output from the sniffer whether the PC or phone sent data first. This was a bit of a pain initially but it quickly became apparent what was going on.

4.4 Capturing data and pouring over printouts

Armed with a basic tool I set about capturing some data corresponding to the “idle” state of the phone to PC connection. As it turned out there is a heartbeat request that goes from the PC to the phone once every second or so to retrieve signal strength and battery status - without this the PC to phone link would behave oddly. This idle exchange contained sufficient information to determine the basic protocol whereupon I did some captures when retrieving and writing phone book entries. I also took a look at what happened at startup and found there was a sequence of commands used to set the link up.

Before printing anything out I did a minor tweak to the sniffer such that data from the PC was also displayed in bold which made it much easier to identify the flow of data then printed a few example dumps. Displayed like this a pattern quickly became apparent and the basic protocol fell into place (see Table 1).

4.5 Parsing code can be your friend

Armed with the information above a state machine based parser was put together that would look at incoming bytes and try and do something useful with them. Appendix A has a code snippet that shows the basic idea, cribbed from the fbus-3810 handler in gnokii. Some of the error checking has been removed for clarity.

By building the basic parser early in the piece it was possible to build the FB38_RX_DispatchMessage() function such that when it received a message that was understood it would be handled accordingly but for unknown message types they would be dumped to STDOUT in a similar hex-dump style to the original sniffer code.

```
To: MT43 SN11 CS51 [02 ][04 ]
Fr: MT46 SN17 CS79 [04 ][48H][6fo][6dm][65e][0c ][2b+][366][311][322][366]
                   [322][366][322][366][300][311][366]
```

This is what our (old) home phone number looked like when being retrieved from the phone by NCDS, the message to the phone is Message Type 0x43 with data fields 0x02 specifying SIM memory and location 0x04. In the response - Message Type 0x46 - the 0x04 byte is the length of the name field and 0x0c is the length of the number field. Rest is pretty self evident...

4.6 Release early, release often

The Open Source mantra of release early, release often actually works *really* nicely if you’re working on these types of reverse engineering problems... others can often see a pattern where you can’t as you’re too familiar with the pages upon pages of captures. Having others try your work is also important as it may show up subtle differences in a protocol relating to firmware versions and the like.

In the context of gnokii I put some basic 3810 code out pretty early and this quickly resulted in more functionality being added as people deciphered more messages and added further details to partially understood ones. The latter is an important point as if you're dealing with any sort of telecommunications gear there will almost inevitably be country or network specific data. As interest in the project grew a couple more people added their input and we quickly arrived at a pretty complete understanding of the protocol.

A second technique that proved useful was to make raw dumps available (in .gz form) for people to download - in the case of gnokii many people involved in the project didn't have NCDS so providing the dumps in a raw form with some markup to indicate what was happening allowed others to work on deciphering things.

5 Doing it with help - Keyspan drivers

Keyspan make a range of USB peripherals including some rather nice USB to serial adapters. The "itch" for this project was a result of moving to an Apple Powerbook as my main machine which lacked serial ports and hence any ability to work with gnokii. I'd looked at other USB to serial adapters on the market - the Keyspan ones simply seemed to be the best quality and price combination. Their subsequent willingness to help was a bonus to say the least!

Initially I just sent some email to their developer support address seeking information about their over the wire protocol and the like. After an initial exchange I got a response from their VP of Engineering who was quite upbeat about the idea of providing Linux support. My advice here is to be patient and remember that what has become second nature to you is still quite an unusual, sometimes even threatening to people unfamiliar with the Open Source community.

Keyspan had a number of questions which whilst easily addressed are worth reiterating here so that if you find yourself in similar circumstances you have a feel for a commercial organisation's perspective on Open Source development.

- *Who will provide support ?* Keyspan are very much customer focussed and were concerned that they wouldn't have the expertise in house to support the resultant drivers. Once I explained that the norm is for the author/maintainer to be the primary contact for support and that, in general, support is handled by the community as a whole this provided a good level of comfort for them.

- *How much input will be required from their development team ?* Would Keyspan have to provide someone full time, was it likely that changes would be needed to the device firmware to provide Linux compatibility - in both cases no but the latter side question gives a good indication of just how helpful they were prepared to be.

- *Proprietary information.* Keyspan very quickly "got it" and understood what Open Source was about and that in a situation where the source code is being made available to the public, a degree of nominally proprietary information will be exposed. I agreed to sign a common sense agreement that basically said I would not release any documentation supplied to me directly other than what would appear in the structure of the driver source itself.

- *Copyright notices.* The Keyspan devices make use of the Ez-USB chipset[8]and so the firmware binary must be included in the driver and uploaded on insertion. A static initialiser contains the firmware image that is produced from a raw binary file supplied by Keyspan and Keyspan requested a suitable copyright/license notice for this. This is not without precedent - a number of SCSI controllers work in a similar manner. Appendix B has a code snippet showing the Copyright/License notice and (partial) 'C' initialiser.

5.1 Writing the driver

Keyspan provided a comprehensive set of documentation for their USB to serial devices which made writing the driver a doddle. They also provided the firmware images for each of the devices that were supported and I was able to talk directly to their developers to get clarification on anything that wasn't clear. In fairly short order I had a basic driver built within the kernel's USB-serial framework that would detect which model Keyspan device was inserted and upload the appropriate firmware.

Things progressed from there to a point where basic character I/O was working up to a point and a reasonable overall framework existed for the driver code. At this point I got distracted with other things (read: work)

[8]Ez-USB is an 8031 core with a separate USB engine. Originally manufactured by Anchor Chips they're now a Cypress Semiconductor part. The devices allow 8031 firmware to be uploaded over USB whereupon the device "Renumerates" or disconnencts/reconnects from the USB bus with a new device ID.

and things went on hold for a while. In true Open Source tradition, Paul Mackerras, one of my colleagues at Linuxcare and kernel hacker extraordinaire more or less finished the driver off one weekend because he needed it working so he could drive his ISDN modem from his new Apple G4 cube. To his considerable credit he also took the time to go through the code with me and explain what he had done and why. Thanks Paulus :)

Some general observations/tricks that I found useful during what was my first major foray into kernel space.

- *You will crash the kernel, occasionally you'll crash it hard.* I eventually ended up doing much of the development on a SMP Intel box, often if you wedged one CPU you could still do a reboot and limp home with the second.
- *Big disks take longer to fsck if you have to do it several times an hour.* This is of course subjective but it's worth considering making a smaller partition or even re-using a smaller drive specifically to boot from when doing kernel development. The inevitable fscks are much quicker. I found NFS mounting the kernel tree was another useful approach providing you have a fast network.
- *A little hardware trick.* If you're doing USB development specifically, consider building a little switch box using a four pole switch to connect/disconnect the four USB lines. This reduces wear and tear on your USB connectors and saves you having to fiddle around at the back of your machine. As a minimum get a 2m extension cable so you at least simulate connect/disconnects from the comfort of the top of your desk.
- *Find a mentor.* I'm fortunate to work in an office with a number of kernel developers which makes it easy to ask questions and get advise on curly problems. You'll almost invariably find someone at your local usergroup or on a mailing list someplace who'll be happy to field the odd question or two.

6 Other notable hacks - more "withouts"

gnokii and the Keyspan drivers are used by a couple of thousand people or so between the two of them - not large in the context of a Linux user base of several million. What may come as a surprise is that many of the much higher profile applications, ports or drivers have in fact been done *without* help from the vendor concerned. In no particular order here are some of them,

- *Linux for Power Macintosh*[9] Essentially all the work done on porting Linux to Apple's PowerMacintosh has been done without technical assistance from Apple. In recent years Apple have assisted by way of loaning equipment, setting up a website and so forth which has been great. To this day though no technical information has been made available, it's all been reverse engineered by Paul Mackerras and a few other developers, initially from scratch then more recently by poking around in the open firmware descriptions of the system. Apple's recent release of the Darwin micro-kernel source code has been a positive step in facilitating this work.
- *Teaching TiVos PAL and Ethernet*[10] Tridge bought a Phillips TiVo in the US, brought it back to Australia and immediately pulled the cover off. Not long afterwards with much larking about he had managed to get the thing to output video using the PAL colour system. A few other suspects got involved at this point and there is now an ISA interface to allow an ethernet card to be used.
- *CueCat Barcode Readers*[11]CueCat produce a hardware/software combination that allows users of MS Windows to scan barcodes in literature and be transported to a corresponding website. These devices have been hacked pretty mercilessly to turn them into a generic barcode wand that operates under Linux. There has also been considerable, ahem, discussion and entreaties made by the manufacturer about intellectual property and such.
- *SMB protocol/Samba*[12]Samba provides interconnectivity between *nix and the MS Windows networking environment. Although Microsoft have been remarkably forthcoming with their support in some areas, much of the hard stuff has had to be reverse engineered through network traces.

[9]`http://linuxcare.com.au/paulus/kernels.html`
[10]`http://linuxcare.com.au/tridge/tivo-ethernet/`
[11]`http://www.flyingbuttmonkeys.com/foocat/`
[12]`http://samba.org/`

7 Conclusions ?

If you're a hardware vendor it should be obvious that choosing *not* to assist in the process of developing open source drivers or utilities to support your particular product will not (for most values of not) prevent them from being written, it will only slow the process somewhat.

It's pleasing to be to note at the end of 2000 that the most major hardware vendors are actively supporting Linux and Open Source software and are reaping commercial rewards in the form of positive publicity and sales into a growing and often quite brand-loyal market.

If you've got a new bit of gear that you can't get vendor support for you need to consider whether it is worthy of your time to try and reverse engineer the device (subject to local law of course). If there are other manufacturers of comparable gear who are more supportive of Open Source development you should consider voting with your feet and buying one of their widgets instead. If the device is unique or just plain cool enough to warrant it, then have a crack at it - it can be a lot of fun too :)

8 Thanks to

- Rusty, Paulus, dwg, and bcg (OzLabs)
- Eric and Stephen (Keyspan)
- Pavel, Pawel, Chris, Jano (gnokii)
- Lucy and Rachael (/home)

A Sample source code for 3810 fbus protocol parser

```
enum FB38_RX_States     RX_State;
int                     MessageLength;
u8                      MessageBuffer[FB38_MAX_RECEIVE_LENGTH];
u8                      MessageCSum;
int                     BufferCount;
u8                      CalculatedCSum;

    /* RX_State machine for receive handling.  Called once for each
       character received from the phone. */
void    FB38_RX_StateMachine(char rx_byte)
{
    static u8   current_message_type;

    switch (RX_State) {

                    /* Messages from the phone start with an 0x04 during
                       "normal" operation, 0x03 when in data/fax mode.  We
                       use this to "synchronise" with the incoming data
                       stream. */
        case FB38_RX_Sync:
                if (rx_byte == 0x04 || rx_byte == 0x03) {
                    current_message_type = rx_byte;
                    BufferCount = 0;
                    CalculatedCSum = rx_byte;
                    RX_State = FB38_RX_GetLength;
                }
                break;

                    /* Next byte is the length of the message including
                       the message type byte but not including the checksum. */
        case FB38_RX_GetLength:
                MessageLength = rx_byte;
                CalculatedCSum ^= rx_byte;
                RX_State = FB38_RX_GetMessage;
                break;

                    /* Get each byte of the message.  We deliberately
                       get one too many bytes so we get the checksum
                       here as well. */
        case FB38_RX_GetMessage:
                MessageBuffer[BufferCount] = rx_byte;
                BufferCount ++;

                    /* If this is the last byte, it's the checksum */
                if (BufferCount > MessageLength) {

                    MessageCSum = rx_byte;
                        /* Compare against calculated checksum. */
                    if (MessageCSum == CalculatedCSum) {
                        /* Got checksum, matches calculated one so
                           pass to appropriate dispatch handler
                           on the basis of current_message_type  */
                        if (current_message_type == 0x04) {
                            RX_State = FB38_RX_DispatchMessage();
                        }
                        else {
                            if (current_message_type == 0x03) {
                                RX_State = FB38_RX_HandleRLPMessage();
                            }
                                /* Hmm, unknown message type! */
                            else {
                                /* Error... */
                                RX_State = FB38_RX_Sync;
                            }
                        }
                    }
                        /* Checksum didn't match so ignore. */
                    else {
                        ChecksumFails ++;
                        RX_State = FB38_RX_Sync;
                    }

                }
                CalculatedCSum ^= rx_byte;
                break;

          default:
                RX_State = FB38_RX_Sync;
                break;
    }
}
```

B Header from the USA-19W firmware

A code snippet from the USA-19W kernel driver showing the license message and the initialiser for the firmware.

```
/* keyspan_usa19w_fw.h
   Generated from Keyspan firmware image Wed Jul 5 09:18:29 2000 EST This
   firmware is for the Keyspan USA-19W Serial Adaptor
   "The firmware contained herein as keyspan_usa19w_fw.h is Copyright
   (C) 1999-2000 Keyspan, A division of InnoSys Incorporated ("Keyspan"),
   as an unpublished work. This notice does not imply unrestricted or
   public access to this firmware which is a trade secret of Keyspan,
   and which may not be reproduced, used, sold or transferred to any
   third party without Keyspan's prior written consent. All Rights Reserved.
   This firmware may not be modified and may only be used with the
   Keyspan USA-19W Serial Adapter. Distribution and/or Modification
   of the keyspan.c driver which includes this firmware, in whole or
   in part, requires the inclusion of this statement."
*/
static const struct ezusb_hex_record keyspan_usa19w_firmware[] = {
   { 0x0000, 3, {0x02, 0x09, 0x60} },
   { 0x0003, 16, {0xe4, 0x90, 0x7f, 0x93, 0xf0, 0x90, 0x7f, 0x9c, 0x74,
     0x30, 0xf0, 0xe4, 0x90, 0x7f, 0x96, 0xf0} },
   { 0x0013, 16, {0x90, 0x7f, 0x94, 0xf0, 0x90, 0x7f, 0x9d, 0x74, 0xff,
     0xf0, 0xe4, 0x90, 0x7f, 0x97, 0xf0, 0x90} },
   .
   .
   .
   { 0xffff, 0, {0x00} }
};
```