One firmware to monitor 'em all.

Andrés Blanco Matias Eissler
Core Security Technologies

August 23, 2012

Abstract

In the last years mobile devices usage has turned massive. These devices, in general, follow the IEEE 802.11 standard for wireless connectivity. Broadcom is one of the most important semiconductor companies in the wireless and broadband communication business. Some of their WiFi solutions (BCM4325 & BCM4329 chipsets) are included in great part of the mobile devices market, including vendors like Apple, Samsung, Motorola, Sony, Nokia, LG, Asus and HTC. In this paper we describe the process of modification of the firmware program on these cards. The presented results could open new possibilities to the information security community such as access to baseband components without intervention of the operating system and the capabilities to store information within the network card's internal memory among others. As the reader explores the present work we go through the internals of the firmware program, our reverse engineering process and show, as a proof of concept, how to set these cards on monitor mode.

1 Introduction

Operating systems provide security by means of layered sets of levels of trust. However peripheral's firmware program code is executed on a separate physical CPU, with its own internal memory, thus firmware program has access to the system hardware without involving the operating system. Code executed with this access level has, regarding hardware interaction, higher privileges than those of an "administrator" or "root" user. Such access level can be obtained on most mobile devices, by simple modifications of unprotected (i.e. not signed) files.

Network card firmware code modification introduces several possibilities. Previous research works show advances in the field of firmware modification: direct access to other peripherals, such as video card GPU [Tri08], stealth agents that could act as a relay for pivoting attacks [Tri10], network card flash components used as persistence mechanisms to host firmware rootkits [Del10] and firewall bypassing schemes [Tri10]. Other possibilities were suggested: the so-called "attacks from below" [Tri10] which would allow not only new attack vectors against the operating system and network card driver modules but also information disclosure of cryptographic material on the host system [DPVL10]. Other works focused on directly accessing the hardware allowed connection with DMA controllers to gain control of modern operating systems [AD10] and even when

	2011		**2010**	
Company	**Units**	**Share (%)**	**Units**	**Share (%)**
Nokia	422,478.3	23.8	461,318.2	28.9
Samsung	313,904.2	17.7	281,065.8	17.6
Apple	89,263.2	5.0	46,598.3	2.9
LG Electronics	86,370.9	4.9	114,154.6	7.1
ZTE	56,881.8	3.2	29,686.0	1.9
RIM	51,541.9	2.9	49,651.6	3.1
HTC	43,266.9	2.4	24,688.4	1.5
Huawei	40,663.4	2.3	23,814.7	1.5
Motorola	40,269.0	2.3	38,553.7	2.4
Sony Ericsson	32,597.5	1.8	41,819.2	2.6
Others	597,326.9	33.7	485,452.0	30.4
Total	**1,774,564.1**	**100.0**	**1,596,802.4**	**100.0**

Table 1: Worldwide Mobile Device Sales to End Users by Vendor in 2011 (Thousands of Units) - Source: Gartner (February 2012)

the protection of an IO/MMU was present, vulnerabilities have been demonstrated and exercised to provide access to operating system memory [SLND10].

We will focus on mobile devices and WiFi technology where adoption has grown exponentially over the last years. These devices evolved from simple gadgets into complex pocket computers available to almost any person. Making use of every capability in these devices can provide the information security community with a portable tool and above all, a really interesting target for multi-stage attacks. It is important to mention that nowadays most uses of mobile devices rely on internet connectivity and wireless 802.11 networks are used as one of the main connection methods.

Broadcom cards are particularly interesting since they are included in a variety of devices including Apple iPhone 4, HTC Nexus One, Samsung Galaxy Tab, Motorola Xoom, Sony Xperia Play, Nokia Lumia 800, Asus Transformer Prime and LG Optimus 2X. As shown in Table 1 only these vendors together sum more than 50% of 2011 device sales, although this statistics cover all brand models and might include cell phones with no wireless capabilities (see Appendix A for a more complete vendor list.)

To demonstrate successful modification of the firmware program, we will provide *monitor mode*, not currently implemented on these type of network cards. Monitor mode provides security researchers with the capability of capturing management, control and data frames that are not specifically destined to a given station. This mode presents an effective method of passive information gathering, that can be used for example, to attack WEP and WPA-PSK cryptosystems [TB09] and [Kor04b, Kor04a]. While on monitor mode network interface cards also provide additional information related to the physical conditions at the moment of frame reception such as *RSSI*, *Rx Power* and *Preamble information* among others.

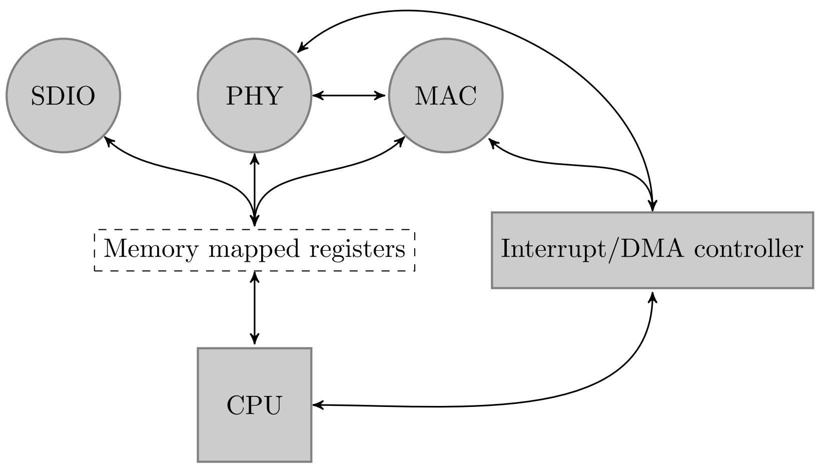
The program source code of Broadcom firmwares is not publicly available and reversing this chipsets' firmware is of great value since it enables modifications on the code to allow full control of the cards' capabilities.

The paper is organized as follows: Section 2 provides an overview of the architecture of Broadcom wireless cards, Section 3 describes how the firmware is designed (most of this information was gathered during the reversing process). Section 4 describes the approach taken in the reversing process to enable firmware modification as described in Section 5 and exemplified in Section 6 which describes how we enabled monitor mode. Finally, Sections 7 and 8 outline the conclusions and future work.

2 Architecture

The BCM4325 & BCM4329 integrate a complete IEEE 802.11 system with Bluetooth and FM radio receiver into a single card, designed for mobile devices that require low power consumption. The hardware consists of an ARM general purpose CPU, both persistent and volatile memory, a set of *cores*, a SDIO bus connection and, of course, antennas. Cores are organized into several functional modules which provide a layered service to various aspects related to the reception, decoding, encoding and transmission of network frames. They also handle the transitions of the data-link protocol state machine and the underlying physical layer details associated with the device radio operation. Some interesting core components include the *PHY core* by which the physical low level details of the antennas power, reception thresholds, temperatures, frequencies and bands are configured. The *MAC core* manages part of the data-link layer protocol implementation and, as we will see, main modes of operation of the adapter. *SDIO core* instruments the communication with the device and on the other end of the SDIO bus the adapter driver manages the communication with the operating system.

Figure 1: Communication between internal card components.



3 Firmware

As mentioned in Section 2, network interface cards (NICs) implement the physical layer protocol of communication. BCM4325 & BCM4329 also implement the data-link layer portion of the 802.11 protocol. To provide greater flexibility, a general purpose CPU is often included. The single program that is loaded and executed by this CPU is what is generally known as *'the firmware'*.

The firmware program is split into several segments or regions, each region containing code and data. We suppose this is due to space constraints. We will call *Region 1* to the segment of the firmware program uploaded by the operating system driver at boot time. Another region (starting at `SI_ARMCM3_ROM` as shown in Listing 1 on BCM4329) is mapped into a higher memory address on the card's internal memory and hence not directly accessible for inspection, we will call this *Region 2*.

Region 1 of the firmware program is usually available as a binary file on the mobile devices' (smartphone or tablet) file-system. Alternatively on older versions in iOS, a specific program named 'wifiFirmwareLoader' contained the firmware program as data. On Apple devices this file can be found at */usr/share/firmware/wifi/43xx/* and on Android devices the file is located at */system/etc/wifi*. Depending on the version of the card, *Region 1* resides in a raw *.bin* file or is contained by a *.trx* file. The trx file format has already been documented by the *openwrt* community[1] and the raw data is quite simple to extract from those files.

Region 1 of the firmware program is loaded into the card's RAM memory, at address zero, where the interruption array will be located. It provides handler functions for hardware and software interruptions and has access to a large set of memory-mapped hardware registers that control various aspects of the physical and data-link layers. These layers are partially implemented in hardware. A DMA engine is also available to the firmware for direct transmission of information between the card subcomponents.

Listing 1: hndsoc.h - bcm driver from Android project

```
#define SI_FLASH2          0x1c000000   /* Flash Region 2 (region 1
    shadowed here) */
#define SI_FLASH2_SZ       0x02000000   /* Size of Flash Region 2 */
#define SI_ARMCM3_ROM      0x1e000000   /* ARM Cortex-M3 ROM */
#define SI_FLASH1          0x1fc00000   /* MIPS Flash Region 1 */
#define SI_FLASH1_SZ       0x00400000   /* MIPS Size of Flash Region
    1 */
#define SI_ARM7S_ROM       0x20000000   /* ARM7TDMI-S ROM */
#define SI_ARMCM3_SRAM2    0x60000000   /* ARM Cortex-M3 SRAM Region
    2 */
#define SI_ARM7S_SRAM2     0x80000000   /* ARM7TDMI-S SRAM Region 2
    */
#define SI_ARM_FLASH1      0xffff0000   /* ARM Flash Region 1 */
#define SI_ARM_FLASH1_SZ   0x00010000   /* ARM Size of Flash Region 1
     */
```

[1]https://forum.openwrt.org/viewtopic.php?id=6938

4 Reversing process

Since the source code for the firmware program is not publicly available, in order to successfully modify its behavior, a reverse engineering process is required. This process will be carried out without debugging capabilities, mainly by reading and analyzing the disassembled assembly code. The necessary steps include: i) Identify the ARM instruction sets compatible with the CPU present in the card hardware. ii) Disassemble the binary program to obtain assembler instructions iii) Determine which assembly code functions produced by the previous step are the low level implementation of common primitive functions iv) Use references to primitive functions or data constants to pinpoint portions of assembly code that implement functionalities of interest. Once we gathered sufficient information with this manual process, dynamic analysis will become possible.

4.1 Instruction set identification

Recognition of the instruction sets compatible with the CPU present in the network card hardware can be accomplished by several methods: simple trial and error tests setting different versions on the disassembler or making use of public information. For instance, advertisment of the BCM4330 states the use of ARM Cortex-M3[2]. Also, the open source code driver of the Android project for these cards, is a great source of information. For example, after reversing some pieces of code, we found cross references and function calls to addresses that belonged to *Region 2*. References to the base addresses of *Region 2* were found on available source code in which the ARM architecture version used is also mentioned (see Listing 1). As a final option, we also considered the use of the *undefined instruction* interruption to determine the exact instruction set version, but we discarded this path.

4.2 Disassembly

Our approach in order to disassemble the firmware program code begins with the interruption array, located at the beginning of the firmware file, which will be loaded at memory address zero. This table contains the addresses of the interruption handler functions. Since the firmware program is loaded at address zero and this is also where the interruption array is located, we can easily know the positions of the handler functions within the file. The BCM4329 firmware usually includes a zero-padded section up to offset `0x100` where a function table relates regions (*Region 1* and *Region 2*) of the firmware. Addresses in this table are mostly functions, so we continue by disassembling instructions there. After the completion of this process, a great deal of the program has been disassembled. For the remaining, we relied on the fact that the ARM architecture requires functions to be 4-byte aligned so compilers usually use nop-padding therefore, binary opcode for ARM two-byte nop is an indication of a possible function prologue ahead.

[2]http://www.broadcom.com/products/Wireless-LAN/802.11-Wireless-LAN-Solutions/BCM4330

4.3 Primitive functions

Once the program has been disassembled, higher level function analysis can begin. The first candidates for identification are implementations of common libc-like functions such as `memcpy`, `memcmp`, `memmove`, `memset`, `strlen`, `strcpy`, `strncpy`, etc. To accomplish this, a useful heuristic is to sort functions by the number of times they are called, then filter by functions containing at least one loop. This results in the immediate discovery of `memcpy` when available in *Region 1* (code distribution between the regions varies greatly from version to version). If `memcpy` is available, and once we have identified it, a quick look on the functions located in the vicinity memory addresses results in the identification of many other of the above-mentioned functions. In fact, identifying `memset` uncovers an allocator function, because of the very common pattern `p=allocate(n); if(p)memset(p, 0, n).`

4.4 Functionalities of interest

Now that basic primitive functions have been identified, we can obtain more information by observing the use of common constants. Since the data-link layer of 802.11 is implemented by the firmware, many constants related to this protocol are referenced by its functions. Also, 6-byte `memcpy` and `memcmp` calls are usual indicators of operations with MAC addresses. In particular, the pattern generated by processing `address1`, `address2`, `address3` and (if present) `address4` fields of the 802.11 frame header[3] stands out remarkably. Identification of this portion of code alone provides enough information to modify the firmware program for basic dynamic analysis, as we will describe in Section 5. Analysis of 802.11 frames crafted by the firmware and obtained with a sniffer provide insights on the constants that are worth searching for. For example, probe request frames contain a vendor specific information element that uses the OUI 00-90-4C assigned to Epigram Inc. (this company was acquired by Broadcom). Searching for this constant reveals the location of the firmware binary program code used on information elements, for example probe requests frames.

5 Firmware modification

Modification of *Region 1* of the firmware program is quite straightforward. The file is not signed or otherwise protected. In BCM4325 and BCM4329 a CRC32 checksum is appended to the binary data for consistency checks. The checksum in BCM4330 is part of the file format. Validation of this checksum is performed by the code on *Region 1*, in other words, the program validates itself. CRC32 calculation for a block in which the last 4 bytes are the checksum result of all the previous block bytes is always the same constant. This constant `crc(data + crc(data))` is always `0xDEBB20E3`. Searching for references to `0xDEBB20E3` reveals the portion of the assembler code program where the checksum is verified. To be able to modify the firmware one possibility is to disable the verification or adjust the checksum at the corresponding offset, we chose the latter.

[3]http://standards.ieee.org/getieee802/download/802.11-2007.pdf Section 7.1.2 "General frame format"

In order to accomplish basic dynamic analysis with behavioral differences (modify the assembly, run and see if worked) we mimic the procedure used by interruption handlers. At a desired point of the code, we modify the assembler by overwriting them with a `branch-with-link` [4] instruction that points to our handler code address. The handler itself is placed over debugging and error message strings. This new handler, backs up all necessary registers into the stack, makes the desired operations, restores the saved register values and finally executes the original (overwritten) code, returning code flow to the original point. With this mechanism and having identified portions of the code that process 802.11 header MAC address fields, a simple "6-byte print" statement becomes available, by filling MAC fields with data to be printed. Data packets' source MAC addresses can now contain any information desired. This debugging mechanism is particularly useful in situations where a register mode branch[5] instruction is encountered and the value of the destination register for the branch instruction is not directly available. For example, a call to a function pointer residing within a struct that was created elsewhere.

Region 2 could also contain code fragments of interest. However, using 6-byte source MAC address field on frames to output large chunks of data is not very convenient. Since the point in the firmware where probe requests frames are assembled has already been identified, modifying this code provides a method to include arbitrary information inside frames of this type.

Probe requests frames are sent periodically by the card while scanning for wireless networks[6]. These frames can contain information elements that can hold up to 255 bytes of data. By modifying the firmware, information elements with fragments of the firmware's *Region 2* are sent over the air. On a separate host, frames of this type can be captured with a sniffer and processed in order to rebuild the hidden segment code, finally unveiling the hidden *Region 2*.

6 Monitor Mode

Monitor mode itself can be seen as 2 different capabilities: access to raw traffic including the data-link layer headers on one side and access to frames that are not directed to the station's or broadcast address on the other. With complete firmware code a thorough analysis is possible.

The first capability is achieved by the described method of modification. In the precise moment when a frame is dequeued from the DMA engine interface, a modification is applied, so that a call to a handler function is placed. The handler makes use of available firmware code functions to copy the frame, prepend an ethernet header and place the newly created frame at the bus queue on its way to the operating system driver (Appendix B). Frame filtering however is performed by hardware and hence modification of the MAC core mode of operation is required. For this reason, identification of functions which provide access to memory-mapped registers of the MAC core are of interest. We are particularly interested in the function `wlc_bmac_mctrl` since providing the right set of flags to this function should directly provide monitor mode. Inspection of available source in Listing 2 shows a possible implementation for it.

[4]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0214b/CHDCGJHB.html
[5]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABEFHAE.html
[6]See 802.11-2007 standard, chapter 5.8.2 - Infrastructure functional model overview

Listing 2: wlc_bmac.c - bcm driver from Android project

```c
1610 void wlc_bmac_mctrl(struct wlc_hw_info *wlc_hw, u32 mask, u32 val)
1611 {
1612   u32 maccontrol;
1613   u32 new_maccontrol;
1614 
1615   ASSERT((val & ~mask) == 0);
1616 
1617   maccontrol = wlc_hw->maccontrol;
1618   new_maccontrol = (maccontrol & ~mask) | val;
1619 
1620   /* if the new maccontrol value is the same as the old, nothing to do */
1621   if (new_maccontrol == maccontrol)
1622     return;
1623 
1624   /* something changed, cache the new value */
1625   wlc_hw->maccontrol = new_maccontrol;
1626 
1627   /* write the new values with overrides applied */
1628   wlc_mctrl_write(wlc_hw);
1629 }
```

In order to compile the statement that contains (`maccontrol &~ mask`) in line 1618 of Listing 2, compilers can make use of ARM instruction BIC[7] (Bit Clear) that provides logical *and not*. Searching for this particular instruction yields 292 results, however, on most results immediate mode[8] is used for one of the operands. We can infer from the source that the assembly code for this operation would not involve constants. Filtering out this type of occurrences for the BIC instruction yields about 50 candidates. We can further narrow the results by function size, leaving a number of candidates that can be manually inspected, leading to the discovery of the function.

Having identified the function that sets the flags which govern the MAC core circuitry mode of operation is of great use. Available source shown in Listing 3 provides a set of flags that are to be used in this core and inspection of the firmware binary code verifies that these flags match with the documented source.

Listing 3: d11.h - bcm driver from Android project

```c
/* maccontrol register */
#define MCTL_GMODE              (1U << 31)
#define MCTL_DISCARD_PMQ        (1 << 30)
#define MCTL_WAKE               (1 << 26)
#define MCTL_HPS                (1 << 25)
#define MCTL_PROMISC            (1 << 24)
#define MCTL_KEEPBADFCS         (1 << 23)
#define MCTL_KEEPCONTROL        (1 << 22)
#define MCTL_PHYLOCK            (1 << 21)
#define MCTL_BCNS_PROMISC       (1 << 20)
#define MCTL_LOCK_RADIO         (1 << 19)
#define MCTL_AP                 (1 << 18)
#define MCTL_INFRA              (1 << 17)
#define MCTL_BIGEND             (1 << 16)
#define MCTL_GPOUT_SEL_MASK     (3 << 14)
#define MCTL_GPOUT_SEL_SHIFT    14
```

[7]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABBFHCJ.html
[8]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/CIHFDDHB.html

```
#define MCTL_EN_PSMDBG          (1 << 13)
#define MCTL_IHR_EN             (1 << 10)
#define MCTL_SHM_UPPER          (1 <<  9)
#define MCTL_SHM_EN             (1 <<  8)
#define MCTL_PSM_JMP_0          (1 <<  2)
#define MCTL_PSM_RUN            (1 <<  1)
#define MCTL_EN_MAC             (1 <<  0)
```

Monitor mode can be accomplished by turning on `MCTL_KEEPCONTROL | MCTL_PROMISC | MCTL_BCNS_PROMISC`. Having applied this modifications we now get raw 802.11 frames mixed with processed packets on the network interface on the operating system side. In order to have a simple method to filter frames and to avoid misconceptions by the operating system state machine, our frame handler also prepends the frame with an ethernet header. We are using ether type `fa:fa` and a hard-coded MAC address to distinguish raw frames from traffic processed by the firmware. The change is now transparent to the interface driver and hence the operating system. Frames that are directed to addresses different from the one of the device where the firmware is running are ignored by the OS however available through libpcap drivers.

7 Conclusion

Firmware program binary code modification was possible and monitor mode was achieved, to make this happen open source code was very important, since it hinted us with the constants and shared memory locations that are necessary to enable this mode. It is also important to mention that a group of reverse engineers [bcm] had already devised some information (such as constants defined in Listing 3), information we reused and checked with other open source projects. Even in the absence of debugging capabilities, code modifications and manual assembly code review allowed us to analyze the behavior of the device. Proliferation of network cards across multiple vendors is an advantage, in this case we enabled monitor mode in a large set of devices across different operating systems by modification of the firmware code alone. In mobile devices, network cards take care of a great deal of complexity, power save functions, physical, data-link and even network layer protocols are implemented by firmware.

8 Future Work

- Provide the capability to transmit custom 802.11 frames usually referred as Injection. This would allow implementation of attacks on the protocol such as: deauthentication flood [9], WPS bruteforcing [10], etc.
- Implement Man-in-the-middle attacks, the firmware can be changed so that network traffic can be modified in a way that provides other attacks such as SSLStrip [Mar09]
- Implement 802.11 Covert Channel capabilities that provide stealth traffic as proposed by [BG11].

[9] *file2air*.http://www.willhackforsushi.com/?page_id=19
[10] *reaver-wps* project. http://code.google.com/p/reaver-wps/

- Provide support for other versions of the Broadcom chipset family.
- Research direct hardware access to other peripherals through the SDIO bus. NIC to flash memory or NIC to telephone baseband, seem particularly interesting.
- Research network card's non-volatile memory access, in particular analyze wheter a persistency mechanism is available.
- Analyze operating system memory modification through DMA.
- Implement peer to peer over the air communication.

References

[AD10] Damien Aumaitre and Christophe Devine. Subverting windows 7 x64 kernel with dma attacks. Hack in the Box, 2010.

[bcm] Broadcom bcm43xx specification. URL: http://bcm-v4.sipsolutions.net/Specification.

[BG11] Andrés Blanco and Ezequiel Gutesman. Abusing the windows wifi native api to create a covert channel. Hack.lu, 2011.

[Del10] Guillaume Delugré. Closer to metal: reverse-engineering the broadcom netextreme's firmware. Hack.lu, 2010.

[DPVL10] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card? CanSecWest Applied Security Conference, 2010.

[Kor04a] KoreK. chopchop (experimental wep attacks), 2004.

[Kor04b] KoreK. Next generation of wep attacks?, 2004.

[Mar09] Moxie Marlinspike. Sslstrip. BlackHat DC, 2009.

[SLND10] F. L. Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an i/ommu vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 7–14, 2010.

[TB09] Erik Tews and Martin Beck. Practical attacks against wep and wpa. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, pages 79–86, New York, NY, USA, 2009. ACM.

[Tri08] Arrigo Triulzi. Project maux mk. ii, i own the nic, now i want a shell. The 8th annual PacSec conference, 2008.

[Tri10] Arrigo Triulzi. The jedi packet trick takes over the deathstar. taking nic backdoors to the next level. CanSecWest Applied Security Conference, 2010.

A Device list

BCM4325

— Apple iPhone 3GS
— Apple iPod 2G
— HTC Touch Pro 2
— HTC Droid Incredible
— Samsung Spica
— Acer Liquid
— Motorola Devour
— Ford Edge (yes, it's a car)

BCM4329

— Apple iPhone 4
— Apple iPhone 4 Verizon
— Apple iPod 3G
— Apple iPad Wi-Fi
— Apple iPad 3G
— Apple iPad 2
— Apple Tv 2G
— Motorola Xoom
— Motorola Droid X2
— Motorola Atrix
— Samsung Galaxy Tab
— Samsung Galaxy S 4G
— Samsung Nexus S
— Samsung Stratosphere
— Samsung Fascinate
— HTC Nexus One
— HTC Evo 4G
— HTC ThunderBolt
— HTC Droid Incredible 2
— LG Revolution
— Sony Ericsson Xperia Play
— Pantech Breakout
— Nokia Lumina 800
— Kyocera Echo
— Asus Transformer Prime
— Malata ZPad

B Monitor mode handler code

```
  AREA MONITOR, CODE

CreatePacketBuff    EQU 0x026010
Memcpy              EQU 0x001764
Sender              EQU 0x004718
PacketBuffShrink    EQU 0x001A34

  push {r0-r3}
  push {r4-r11, lr}
  adr r11, FrameSkipCount    ; R11 <- Adr FrameSkipCount
  ldr r10, [r11]             ; R10 <- FrameSkipCount
  cmp r10, #0                ; Drop first frames so that
  bne DontInject             ; driver can boot normally.

  mov r8, r2                 ; r8 <- Raw frame PacketBuff.
  ldr r7, [r6,#0x73C]        ; r7 <- WLC_INFO struct pointer

  ldrh r0, [r8, #0x14]
  add r0, #0x20
  bl CreatePacketBuff
  cbz r0, done
  mov r6, r0                 ; r6 <- New packet buff.
  mov r1, #0x12
  bl PacketBuffShrink

  ldr r0, [r6, #0x10]        ; dst = New packet buff
  adr r1, PacketHeader       ; src = Ethernet frame header
  mov r2, #0xE               ; size
  bl Memcpy

  ldr r0, [r6, #0x10]        ; dst = New packet buff
  add r0, #0xE
  ldr r1, [r8, #0x10]        ; src = Raw 802.11 frame.
  ldrh r2, [r8, #0x14]       ; size
  bl Memcpy

  mov r0, r7
  mov r1, #0
  mov r2, r6
  mov r3, #1
  bl Sender            ; Enqueue the frame on the SDIO BUS Q
  b done

DontInject
  sub r10, #1
  str r10, [r11]
done
  pop {r4-r11}
  pop {lr}
  pop {r0-r3}
  ldr.w r11, [r2,#0x10]
  bx  lr

FrameSkipCount  DCD 0x200
PacketHeader    DCD 0xFFFFFFFF
                DCD 0x8888FFFF
                DCD 0x88888888
                DCW 0xFAFA
  end
```