Intro to Embedded Reverse Engineering for PC reversers

Igor Skochinsky
Hex-Rays

Recon 2010
Montreal

Outline

- Embedded systems
- Getting inside
- Filesystems
- Operating systems
- Processors
- Disassembling code

Embedded systems

- Highly integrated
- Designed as a single unit
- Not much configurability
- Emphasis on size and power consumption
- Reversing challenges
    - Many variations
    - New architectures/instructions set
    - Difficult to find information
    - No conveniently named APIs

Getting inside

- First step: getting the code
  - Firmware updates
  - Serial port
  - Communication with PC
  - JTAG
  - Flash chip dumping
  - Exotic ways

Getting inside: firmware updates

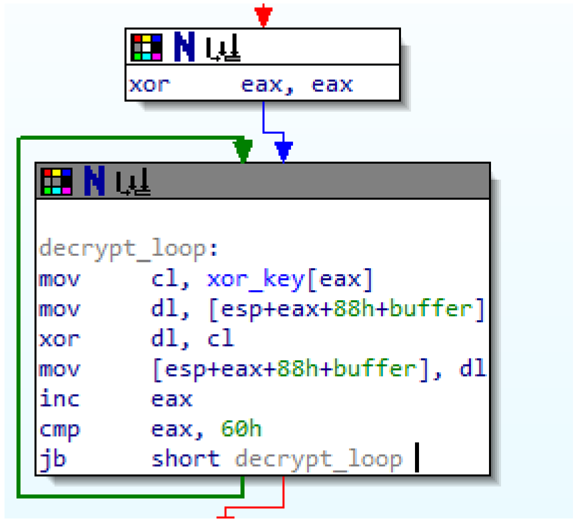
- Most manufacturers provide updates for devices
- Simplest way to get code if not encrypted/obfuscated
- Usually comes in one of two flavors:
    - An updater program to run on PC (includes the actual update file, separately or embedded)
    - Just an update file to be copied to the device manually

Getting inside: firmware updates

- Update files can contain:
  - Filesystem images
  - Kernel images
  - Bootloader images
  - Updater programs or scripts
  - Some (or all) of them can be compressed
- Look for big chunks of 00s or FFs delimiting the parts
- Check for common compression stream patterns
  - zlib: 78 01, 78 9C, 78 DA
  - gzip: 1F 8B
  - LZMA: 5D 00 00 80

Getting inside: firmware updates example 1

- Sony Librie (the first E-Ink ebook reader)
- Includes UPLIBRIE.exe, EBRCTR.dll and data.bin
- data.bin contains several images encrypted with a fixed XOR key
- Decryption is done on PC side, so images are easily recovered

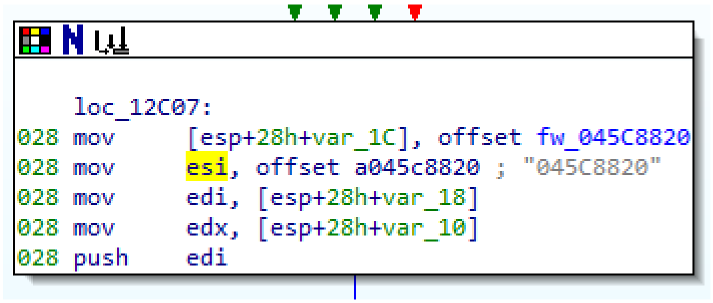


```
xor     eax, eax

decrypt_loop:
mov     cl, xor_key[eax]
mov     dl, [esp+eax+88h+buffer]
xor     dl, cl
mov     [esp+eax+88h+buffer], dl
inc     eax
cmp     eax, 60h
jb      short decrypt_loop
```

Getting inside: firmware updates example 2

- Intel SSD drive
- Update is a bootable FreeDOS CD ISO
- Actual updater is a 32-bit DOS program (LE), compiled with Watcom (iSSDFUT.exe)
- Firmware images are compiled in as byte arrays
- Need to be identified by following program logic
- ATA command DOWNLOAD MICROCODE is used
- Each image is 256000 bytes



```
   loc_12C07:
028 mov     [esp+28h+var_1C], offset fw_045C8820
028 mov     esi, offset a045c8820 ; "045C8820"
028 mov     edi, [esp+28h+var_18]
028 mov     edx, [esp+28h+var_10]
028 push    edi
```

Getting inside: firmware updates example 3

- Amazon Kindle
- Firmware update is an obfuscated .bin file
- The file needs to be copied to the Kindle's USB drive
- Update process triggered by user command
- De-obfuscation and unpacking is done on the device

```
extract_bundle()
{
    dd if=$1 bs=${BLOCK_SIZE} skip=1 | dm | tar -C $2 -xzvf -
}
```

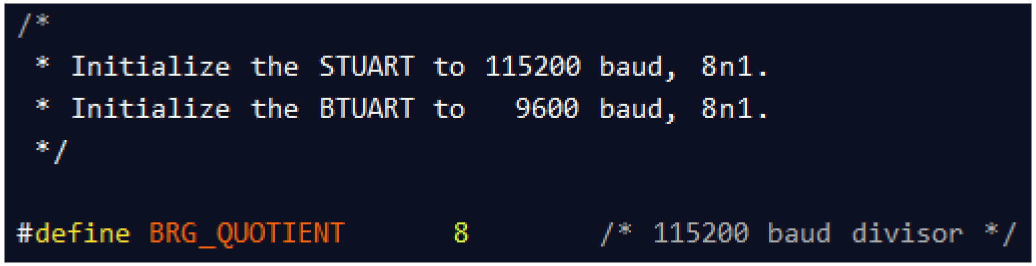
- Thus the code has to be extracted with other means

Getting inside: serial port

- Also known as UART (universal asynchronous receiver/transmitter), or just "COM port"
- Used during development and diagnostics
- Allows access to the bootloader and/or main OS
- Getting the code usually involves these steps:
  - Find UART parameters
  - Get/make the cable
  - Identify on board
  - Download the code

Getting inside: UART parameters

- Necessary to get the correct data from the device, since protocol is sensitive to timing
- Best source is Linux/bootloader sources if manufacturer provides them
- Grep for "uart”, "serial”, "console”.
- Otherwise, the most common settings are:
    - For Linux devices, 115200 8n1
    - For Windows CE devices, 38400, 8n1



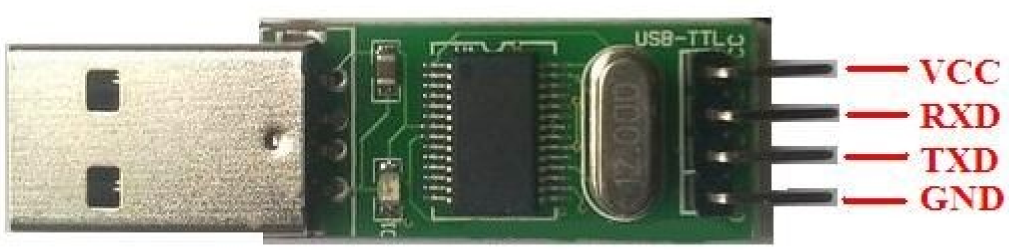
```
/*
 * Initialize the STUART to 115200 baud, 8n1.
 * Initialize the BTUART to   9600 baud, 8n1.
 */

#define BRG_QUOTIENT      8       /* 115200 baud divisor */
```

Getting inside: serial cable

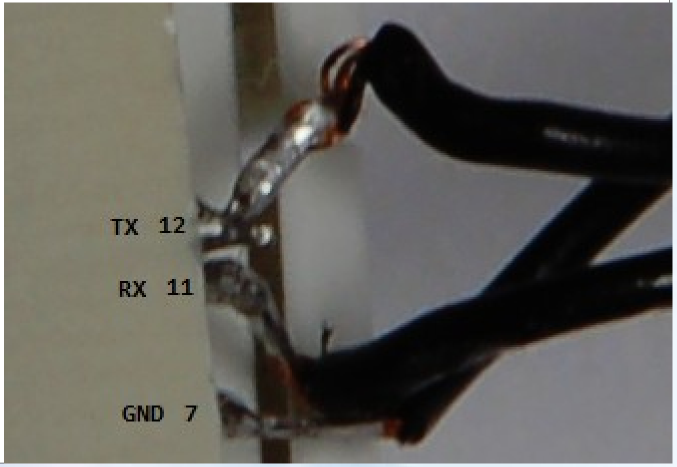
- Devices operate on TTL levels (0/3.3V or 0/5.5V)
- PC COM port operates on RS-232 levels (up to +/-15V)
- Need a level converter
  - Self made: using a MAX232/3232 chip or analog
  - USB-TTL converter: using FT232/2232 chip
  - Phone cable: many mobile phones actually used serial port to communicate with PC

Many such converters are available pretty cheap on Ebay



Getting inside: UART identification on board

- Three or four pins are used: TX, RX, GND (and Vcc)
- Check for groups of four pins on board
- Otherwise, can be found by trial and error
    - GND: use metallic shields or connector housing
    - Start up terminal program, attach RX to potential TX pin, reset device
    - Repeat until you get something on the screen
    - RX is usually nearby and often you can see the echo from typing



Getting inside: getting code via UART 1/3

- If you get to login prompt, try some obvious logins/passwords
- E.g. root/root, root/admin, root/<device name>
- Once logged in, you can usually copy the files somehow
    - USB partition: if there's a partition that's visible on the PC, you can copy files there
    - Connect/disconnect USB or insert a memory card to see if the filesystem changes
    - Check /etc/fstab, init scripts for mention of external filesystems

Getting inside: getting code via UART 2/3

- Another way is to dump the complete flash
- Linux usually uses MTD (Memory Technology Devices) translation layer to handle flash partitioning
- To see the flash map:

```
cat /proc/mtd
```

- To copy a partition:

```
dd if=/dev/mtdN of=/tmp/dumpN
```

```
dev:    size     erasesize name
mtd0:   00200000 00010000  "sdm device NOR 0"
<...>
mtd11:  00160000 00020000  "Linux0"
mtd12:  007e0000 00020000  "Rootfs2"
<...>
```

Getting inside: getting code via UART 3/3

- If you can't login into Linux, you might be able to get into the bootloader
- There are many bootloaders used in embedded systems; Das U-Boot seems to be most common
- To stop in the bootloader you usually need to press a key shortly after reset
- Actual commands to use depend on the device (try 'help')
- For Kindle, I had to use the hex dump command, as there was no way to copy data over USB or to a memory card

```
check_recovery: shift-<r>ecover, shift-<u>pdate, shift-</> reset...
normal boot...
U-Boot 1.1.2 (Oct 29 2007 - 16:35:25)
*** Welcome to Kindle ***
```

Getting inside: misusing the PC communication

- Many manufacturers provide programs to communicate with the device from PC
- It's worth reversing the program to see what it does
- There might be hidden commands not available via the UI
- There might be legitimate commands that can be used in novel ways
- USB traffic spying might be useful

Getting inside: misusing the PC communication example 1

- Sony Reader PRS-500
- Was bundled with "Sony CONNECT Library”
- The program copied ebooks to the device
- Ebooks went into the directory /Data/
- Protocol had "write”, "read”, "list” and "delete” commands
- Wrote a custom script that talked to interface DLL and could specify any path
- /proc/mtd and /dev/mtdN were accessible this way
- The complete fimware was downloaded

Getting inside: misusing the PC communication example 2

- Casio EX-Word electronic dictionary
- Casio offered free downloads of games to install on it
- Reversed the program used to install them
- Games were decrypted just before sending over USB
- Dumped decrypted games, spent some time reversing
- Eventually found references to savefile
- Replaced savebuffer address with system memory's
- Uploaded modified game; instead of savedata system memory was saved to file
- By "unloading” the game, could get the savefile back
- Dumped system memory (the OS) in chunks

Getting inside: JTAG

- Joint Test Action Group
- Originally developed for testing circuit boards
- Many microcontrollers add commands for debugging
- Allows complete control over the processor
- Can be used to download and upload flash
- Needs an adapter
- Pins are not always obvious; programs exist to brute-force the layout
- Another option is to trace pins from the processor (might need desoldering)

Getting inside: Flash chip dumping

- The actual flash content is (so far?) rarely encrypted
- So dumping the chip itself (if it is separate) is an option
- Sometimes can be done in-place, but usually requires desoldering the chip
- Best done by an experienced person

Getting inside: exotic ways

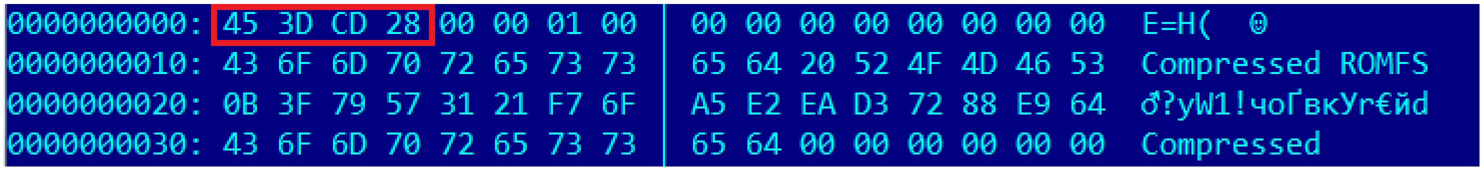
- CHDK project (custom fimware for Canon cameras)
- Used the camera LED to dump the firmware
- Firmware bits were transmitted as LED light pulses with some redundancy and error checking
- Pulses were recorded as sound via a phototransistor
- Resulting waveform was decoded to binary data

Embedded filesystems

- Embedded systems often use special filesystems
- Space is limited, so compression is used
- Read-only filesystems for permanent files (rootfs)
- Flash wear should be taken into account
- Most common filesystems nowadays:
    - cramfs
    - SquashFS
    - JFFS2
    - YAFFS
    - others (VFAT, ext2, minix, UBIFS)

Embedded filesystems: cramfs

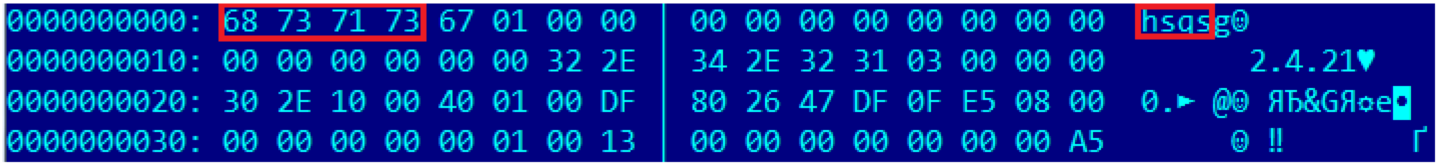
- Stands for "Compressed ROM file-system"
- Read-only filesystem
- Files are compressed with zlib (aka deflate)
- Has some limitations, so getting less popular
- Signature magic: 0x28CD3D45
- To unpack: cramfsck -x <dir> image.cramfs



```
0000000000: 45 3D CD 28 00 00 01 00 │ 00 00 00 00 00 00 00 00  E=H(   ⚇
0000000010: 43 6F 6D 70 72 65 73 73 │ 65 64 20 52 4F 4D 46 53  Compressed ROMFS
0000000020: 0B 3F 79 57 31 21 F7 6F │ A5 E2 EA D3 72 88 E9 64  ♂?yW1!чoҐвкУr€йd
0000000030: 43 6F 6D 70 72 65 73 73 │ 65 64 00 00 00 00 00 00  Compressed
```

Embedded filesystems: SquashFS

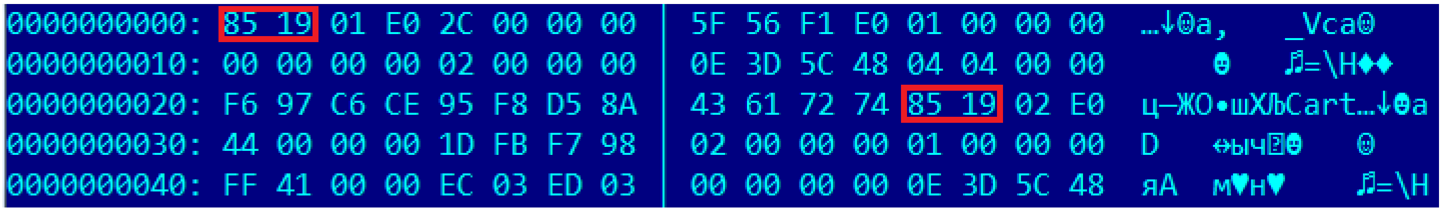
- Another compressed read-only filesystem
- Can use zlib or LZMA
- Present in mainline Linux
- Was used in Kindle
- Signature magic: 0x73717368 ("sqsh")
- To unpack: unsquashfs -d <dir> image.sqs



```
0000000000: 68 73 71 73 67 01 00 00 | 00 00 00 00 00 00 00 00  hsqsg☺
0000000010: 00 00 00 00 00 00 32 2E | 34 2E 32 31 03 00 00 00        2.4.21♥
0000000020: 30 2E 10 00 40 01 00 DF | 80 26 47 DF 0F E5 08 00  0.► @☺ ЯЂ&GЯ☼е◘
0000000030: 00 00 00 00 00 01 00 13 | 00 00 00 00 00 00 00 A5       ☺ ‼       Ґ
```

Embedded filesystems: JFFS2

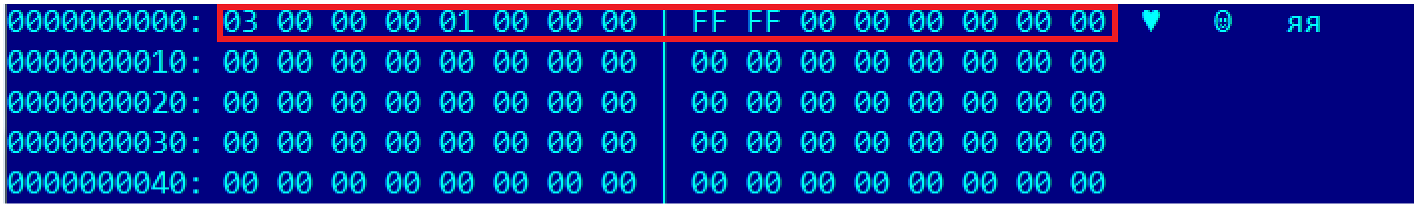
- Journalling Flash File System 2
- Used when writable FS is needed (e.g. user files)
- Designed to reduce flash wear
- Has several compression algorithms
- Signature magic: nodes start with 0x1985
- Official way to unpack involves Linux and MTD devices
- Wrote a Python script to do it in one go on any platform



```
0000000000: 85 19 01 E0 2C 00 00 00 | 5F 56 F1 E0 01 00 00 00  …↓☺a,   _Vca☺
0000000010: 00 00 00 00 02 00 00 00 | 0E 3D 5C 48 04 04 00 00      ☻   ♫=\H♦♦
0000000020: F6 97 C6 CE 95 F8 D5 8A | 43 61 72 74 85 19 02 E0  ц—ЖО•шХЉCart…↓☻a
0000000030: 44 00 00 00 1D FB F7 98 | 02 00 00 00 01 00 00 00  D   ↔ыч☻   ☺
0000000040: FF 41 00 00 EC 03 ED 03 | 00 00 00 00 0E 3D 5C 48  яA  м♥н♥    ♫=\H
```

Embedded filesystems: YAFFS

- Yet Another Flash File System
- Recently popularized by use in Google Android
- Also designed for flash chips
- No dedicated magic
- Tends to begin with 03 00 00 00 01 00 00 00 FF F
- To unpack: unyaffs image.yaffs



```
0000000000: 03 00 00 00 01 00 00 00 | FF FF 00 00 00 00 00 00  ♥   ☺   яя
0000000010: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0000000020: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0000000030: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0000000040: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

Embedded filesystems: others

- ext2/ext3: sometimes used even though not optimized for flash
- VFAT: used as backing FS for mass-storage devices
- UBIFS: "successor" to JFFS2, used in N900
- LogFS: another successor
- minix
- romfs – used for initial ramdisk inside the kernel
- Propietary/non-Linux
  - Microsoft: TFAT, TExFAT
  - Samsung: TFS4 (Transactional File System 4), RFS (Robust FAT File System)
  - Wind River: TrueFFS

Embedded Operating Systems

- Embedded devices are constrained by processing power and memory
- Simplest devices do not use any OS at all
- RTOS (Real-time OS)
    - Emphasis on fast reaction to events and predictability
    - Handles narrow set of tasks
    - Provides basic functionality for running tasks (or threads), sync primitives, messaging and other APIs
    - Can be very small, from a few KB
    - Tasks usually fixed at compile time and linked into final image
- Linux is used for "bigger", usually CE devices

Embedded operating systems: Linux

- Gets more and more widespread
- Generally needs a processor with MMU (Memory Mapping Unit); MMU-less variant exists (uCLinux)
- Due to GPL sources have to be provided by the maker
- Thus often easiest to reverse
- Needs a bootloader; Das U-Boot used often
- WebOS (Palm) and Android (Google) are also based on Linux
- Identification: "Linux" is usually present somewhere in the image or in the boot output; also check maker site for sources

Embedded operating systems: Nucleus RTOS

- A small RTOS from Mentor Graphics
- Distributed in source form (NOT open source)
- Mostly written in C with few processor-specific parts
- Used in many mobile phones (Siemens, Samsung etc)
- Example ID string:

```
Copyright (c) 1993-2002 ATI - Nucleus PLUS -
Integrator ADS v. 1.13.4
```

Embedded operating systems: VxWorks

- RTOS, made by Wind River Systems (bought by Intel)
- Used in:
  - Network appliances (home routers/modems)
  - Set-top boxes, DVD players
  - Even in spacecraft (Mars rovers Spirit and Opportunity)
- Example ID string:

  `Copyright 1999-2001 Wind River Systems, Inc.`

- http://chargen.matasano.com/chargen/2008/4/29/retsaot-is-toaster-reversed-quick-n-dirty-firmware-reversing.html

Embedded operating systems: Windows CE

- Written from scratch RTOS with Win32 APIs
- Pretty popular due to low licensing charges and familiar API
- First versions ran on ARM, MIPS, PowerPC, SuperH, x86
- Now only ARM and x86 officially supported
- Pocket PC, Windows Mobile run on Windows CE kernel
- Not just phones, also common on GPS devices and other portable devices (e.g. Panasonic Words Gear)

- ID string: "CECE" at offset 0x40

Embedded operating systems: others

- QNX, ThreadX, µC/OS-II
- Symbian
- eCos
- TRON (ITRON, BTRON etc): a common RTOS specification/interface used mostly by Japanese makers
- Sometimes chip manufacturers offer a standard OS
    - SuperH: HI7700/4 for from Renesas

Embedded processors

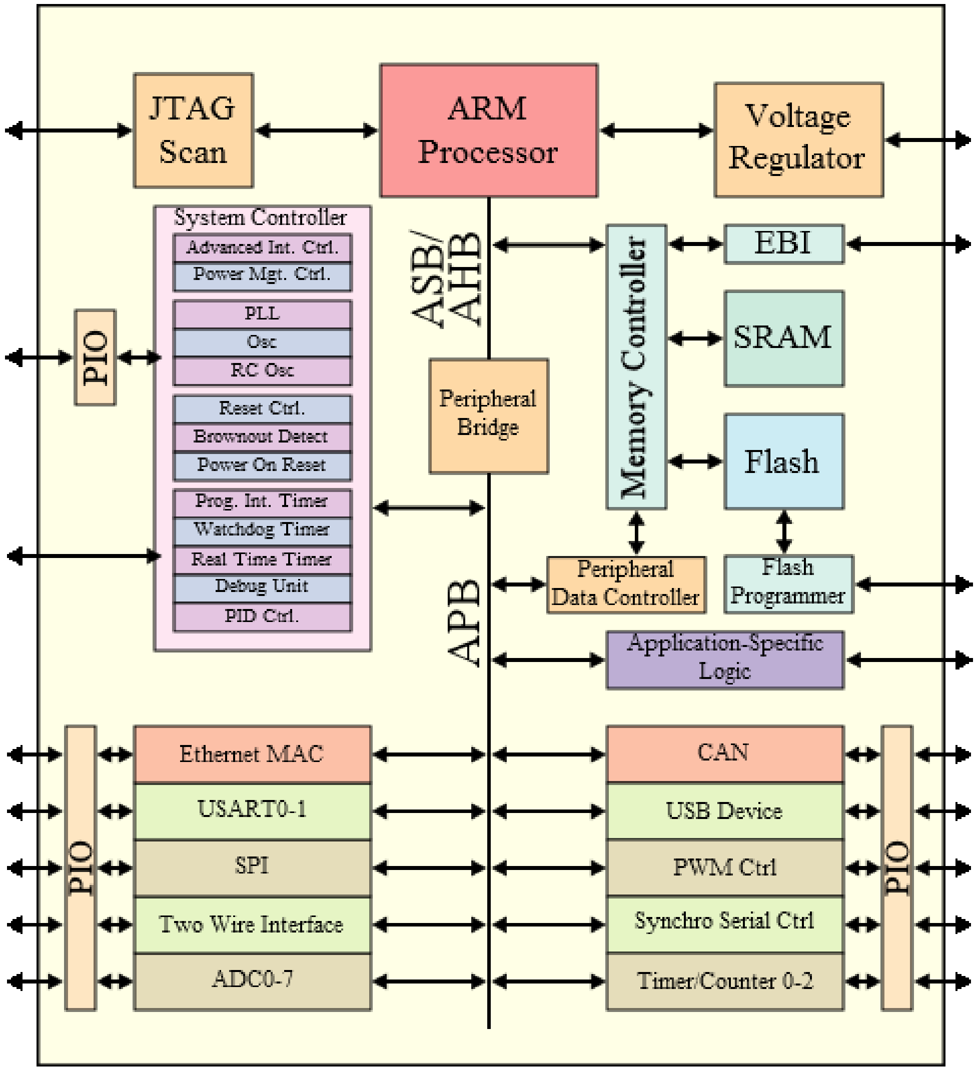
- Microcontrollers vs. microprocessors
- RISC vs. CISC
- Common processors

Embedded processors: microcontrollers vs. microprocessors

- Microprocessor:
  - Includes processor core (decoder, ALU, registers etc)
  - Interfaces with extra controllers for RAM, ROM and other peripherals
  - General-purpose
- Microcontroller:
  - Besides processor core, integrates RAM, program ROM (often Flash ROM), and peripherals
  - Thus commonly called "System-on-Chip" (SoC)
  - Specialized
  - In the same family many variations exist with different sets of peripherals tailored for different tasks

Embedded processors: microcontrollers vs. microprocessors

System-on-Chip



Modern PC

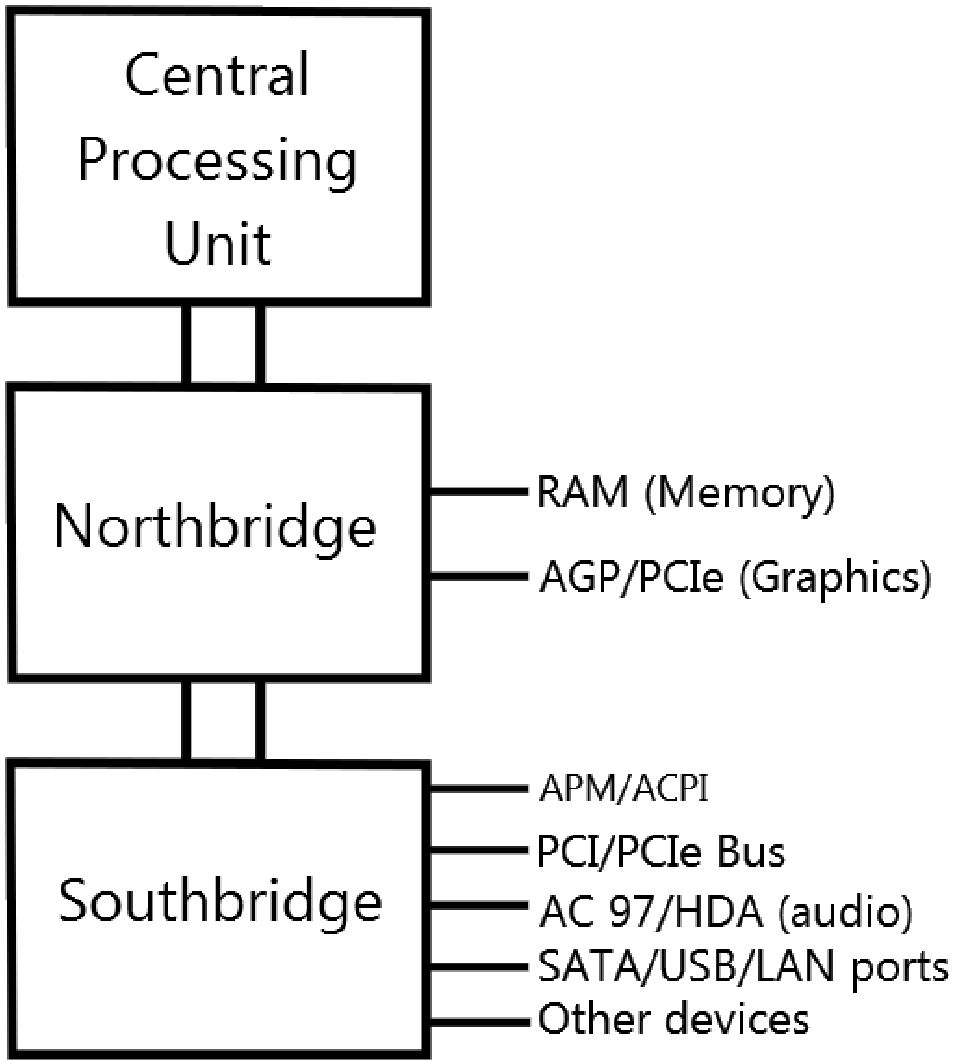


Image: wiki commons/Cburnett, CC-BY-SA-3.0

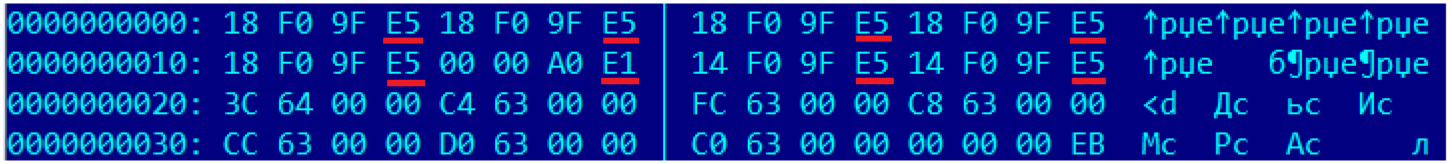
Image: wiki commons/David Futcher, CC-BY-SA-3.0

Embedded processors: RISC vs. CISC

- RISC: Reduced instruction set computing
  - Basic strategy: simple instructions that execute quickly
  - Most instructions usually need one cycle
  - Short, fixed-length encodings (commonly 2/4 bytes)
  - Load/store architecture
  - Orthogonal instruction set and registers
- CISC: Complex instruction set computing
  - Complex instructions that do complex things
  - Often use complex, multi-byte instruction encodings
  - Specialized registers for some operations
- In recent years the boundary between RISC and CISC is getting blurry

Embedded processors: ARM

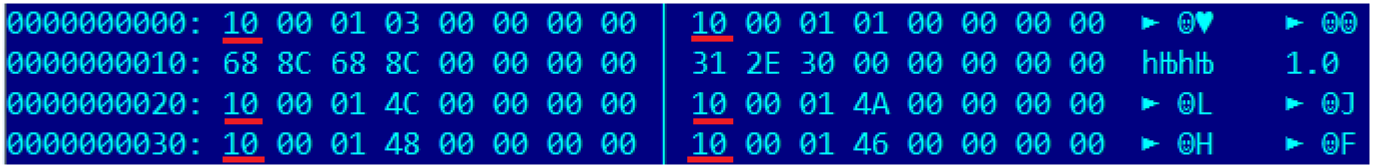
- Flat 4GB memory space
- Actual layout depends on the chip
- Reset entry: 0 for "classic" ARMs
- Usually has B xxx (0xEAxxxxxx) or LDR PC (0xE59FFxxx) there
- For Cortex-M: value at 0 is initial SP, at 4 – initial PC
- Common patterns:
  - ARM mode (little-endian): xx xx xx Ex (always-executing instruction)
  - Thumb mode (little-endian): 47 70 (BX LR), xx B5/xx B4 (PUSH)



```
0000000000: 18 F0 9F E5 18 F0 9F E5 | 18 F0 9F E5 18 F0 9F E5  ↑pџe↑pџe↑pџe↑pџe
0000000010: 18 F0 9F E5 00 00 A0 E1 | 14 F0 9F E5 14 F0 9F E5  ↑pџe    6¶pџe¶pџe
0000000020: 3C 64 00 00 C4 63 00 00 | FC 63 00 00 C8 63 00 00  <d  Дc  ьc  Иc
0000000030: CC 63 00 00 D0 63 00 00 | C0 63 00 00 00 00 00 EB  Mc  Pc  Ac     л
```

Embedded processors: MIPS

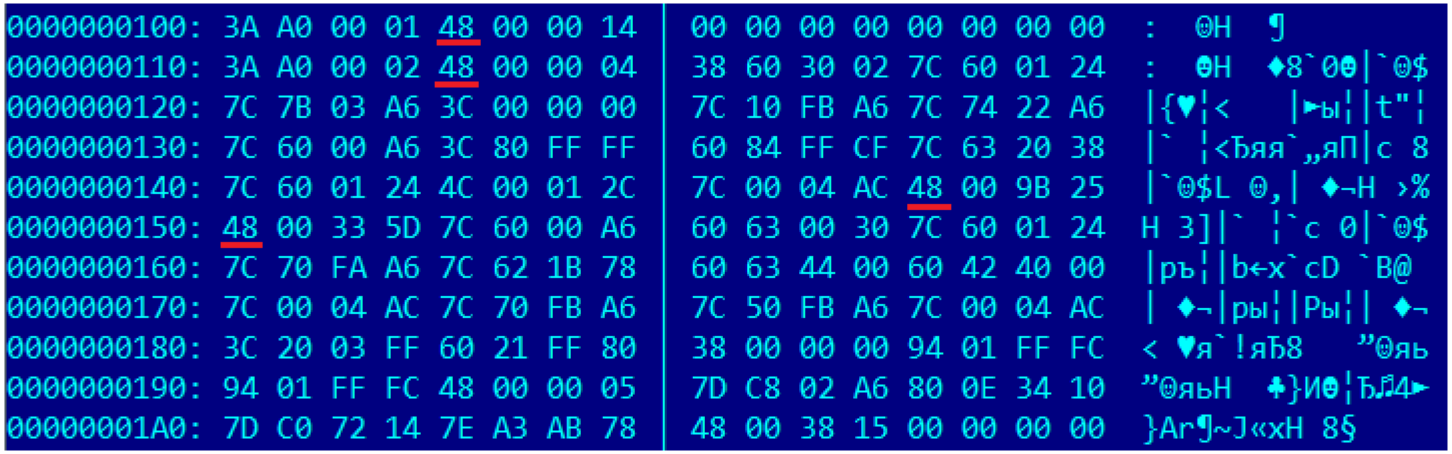
- Memory usually divided in several segments with different virtual addresses mapping to the same physical address
- Reset entry: virtual 0xBFC00000, physical 0x1FC00000
- Usually has a branch there: 0x1000xxxx
- Common patterns:
  - 03 E0 00 08 (jr $ra – return)
  - 3C xx xx xx (lui)
  - 24 xx xx xx (addiu)
  - 1x xx xx xx (branches)
  - 0C xx xx xx (jal)



```
0000000000: 10 00 01 03 00 00 00 00  10 00 01 01 00 00 00 00  ► ☺♥     ► ☺☺
0000000010: 68 8C 68 8C 00 00 00 00  31 2E 30 00 00 00 00 00  hЊhЊ     1.0
0000000020: 10 00 01 4C 00 00 00 00  10 00 01 4A 00 00 00 00  ► ☺L     ► ☺J
0000000030: 10 00 01 48 00 00 00 00  10 00 01 46 00 00 00 00  ► ☺H     ► ☺F
```

Embedded processors: PowerPC

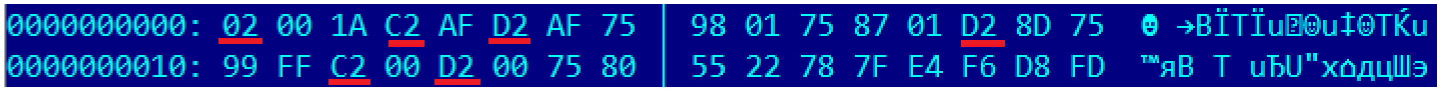
- Flat 4GB memory space (for 32-bit version)
- Reset entry: 0xFFFFFFFC for "Book E" (embedded) variant
- 0x100 or 0xFFF00100 for "classic" PPC
- Common patterns
  - 4E 80 00 20 (blr – return)
  - 48 xx xx xx (branches)
  - 7C 08 xx A6 (mtlr %rx – save link register)



```
0000000100: 3A A0 00 01 48 00 00 14  00 00 00 00 00 00 00 00  :  ☻H  ¶
0000000110: 3A A0 00 02 48 00 00 04  38 60 30 02 7C 60 01 24  :  ☻H  ♦8`0☻|`☺$
0000000120: 7C 7B 03 A6 3C 00 00 00  7C 10 FB A6 7C 74 22 A6  |{♥¦<   |►ы¦|t"¦
0000000130: 7C 60 00 A6 3C 80 FF FF  60 84 FF CF 7C 63 20 38  |` ¦<Ђяя`„яП|c 8
0000000140: 7C 60 01 24 4C 00 01 2C  7C 00 04 AC 48 00 9B 25  |`☺$L ☺,| ♦¬H ›%
0000000150: 48 00 33 5D 7C 60 00 A6  60 63 00 30 7C 60 01 24  H 3]|` ¦`c 0|`☺$
0000000160: 7C 70 FA A6 7C 62 1B 78  60 63 44 00 60 42 40 00  |pъ¦|b←x`cD `B@
0000000170: 7C 00 04 AC 7C 70 FB A6  7C 50 FB A6 7C 00 04 AC  | ♦¬|pы¦|Pы¦| ♦¬
0000000180: 3C 20 03 FF 60 21 FF 80  38 00 00 00 94 01 FF FC  < ♥я`!яЂ8   ”☺яь
0000000190: 94 01 FF FC 48 00 00 05  7D C8 02 A6 80 0E 34 10  ”☺яьH  ♣}И☻¦Ђ♫4►
00000001A0: 7D C0 72 14 7E A3 AB 78  48 00 38 15 00 00 00 00  }Ar¶~J«xH 8§
```

Embedded processors: 8051

- Probably the most ubiquitous microcontroller architecture
- Developed in 1980, new models still appear today
- Harvard architecture (instructions separately from data)
- Bit-addressing instructions
- Reset vector: 0000H
- Common patterns
    - 02 xx xx (ljmp – absolute jump)
    - 22 (ret)
    - C2 xx, D2 xx (clr – clear bit, setb – set bit)



```
0000000000: 02 00 1A C2 AF D2 AF 75 │ 98 01 75 87 01 D2 8D 75   ☻ →BÏTÏu?☺u‡☺TЌu
0000000010: 99 FF C2 00 D2 00 75 80 │ 55 22 78 7F E4 F6 D8 FD  ™яB T uЂU"x⌂дцШэ
```

Embedded processors: Others

- 32-bit
  - M68K: CPU32, ColdFire (CISC)
  - SuperH (RISC with 16-bit instructions)
  - M32R
  - NEC V850, 78K0
- 8-bit
  - Microchip PIC
  - Atmel AVR
  - 68HC08, 68HC11
- Identification
  - Search for the chip markings
  - Check the manufacturer logo:

  http://www.elnec.com/support/ic-logos/?method=logo

Disassembling code

- After getting the code and identifying processor, next step is the actual disassembly
- Common image formats
    - ELF
    - OS-specific
    - Kernel images
- Raw binary
- Recovering symbol information

Disassembling code: ELF

- Executable and Linkable Format
- Was developed for use on UNIX systems
- Nowadays used in other embedded systems and even OS-less environments
- Many tools available
    - binutils: objdump, objcopy
    - IDA Pro
    - pyelf
- Format specifies loading addresses
- For a common processor usually no extra work required
- Less common processors might need some manual work (relocations, imports etc.)

Disassembling code: OS-specific formats

- Some OSes use custom formats
- Windows CE: PE files
- iOS (formerly iPhone OS): Mach-O
- Symbian: EPOC
- Unix 'file' command can be useful in identifying formats
- Also objdump from binutils (compile with --enable-targets=all)

Disassembling code: Kernel images

- Often kernel is a structured image itself, e.g. an ELF or Mach-O file
- However, often this is not the case
- Usually one of two options is used
- 1) Image is prepared to be run from a specific address, where it is loaded by bootloader
- 2) A small bootstrap is prepended to the kernel ('piggy' in Linux terms), which unpacks or just copies the rest of the image to the final location, then jumps to it
- For ARM Linux kernel, you can extract the final image by looking for gzip signature (1F 8B) and extracting the data from there

Disassembling code: raw binary

- Sometimes there are no structured files, just binary code
- First you need to determine the load base
- A good first try would be just to load it at 0 and see if things match up
- Otherwise try to use hints from the code
    - Self-relocating code
    - Initialization code
    - Jump tables
    - String tables

Disassembling code: raw binary, self-relocating code

- Self-relocating code copies itself to proper address if running from a different one

```
00000040         MOV     R0, PC      ; R0 = current address + 8
00000044         LDR     R1, =0x48   ; subtract 0x48 to get the load base
00000048         SUB     R0, R0, R1  ; R0=load base
0000004C         LDR     R1, =0      ; 0 is the preferred load base
00000050         CMP     R0, R1      ; are we at the preferred base?
00000054         BEQ     loc_78      ; if yes, continue
00000058         LDR     R1, =0      ; R1 = target
0000005C         LDR     R2, =0xB000 ; R2 = count
00000060 copyloop
00000060         LDR     R3, [R0],#4 ; load word
00000064         SUBS    R2, R2, #4  ; decrease counter
00000068         STR     R3, [R1],#4 ; store word
0000006C         BNE     copyloop    ; repeat until done
00000070         LDR     R1, =0      ; load the new address
00000074         BX      R1          ; branch to it
```

- This way you can also determine exact boundaries of the code fragment

Disassembling code: raw binary, initialization code

- Initial code usually runs from flash
- But the main program needs writable data, both initialized (.data) and uninitialized (.bss)
- The usual algorithm of the startup code:
    - Copy code for faster execution to RAM (optional)
    - Copy initialized data from read-only memory to RAM corresponding to .data segment
    - Zero out the uninitialized data area (.bss)
- Identifying these steps can help to determine the actual load address of the code

Disassembling code: raw binary, jump tables

- Compilers often implement switch statements with jump tables
- Offsets in the jump table should point to valid code near the indirect jump instruction

```
1001458C    CMP     R2, #3          ; switch 4 cases
10014590    LDRLS   PC, [PC,R2,LSL#2] ; switch jump
10014594    B       loc_100145E0    ; default
10014594 ; ---------------------------------------------------------------
10014598    DCD loc_100145A8        ; jump table for switch statement
10014598    DCD loc_100145AC
10014598    DCD loc_100145AC
10014598    DCD loc_100145D8
100145A8 ; ---------------------------------------------------------------
100145A8 loc_100145A8
100145A8   B       loc_100145E0    ; jumptable 10014590 case 0
```

Disassembling code: raw binary, string tables

- Sometimes the program uses a table of strings
- It is usually represented by an array of offsets to strings
- By subtracting offsets you can get lengths of strings
- Those can then be matched against strings in the binary
- This can even be automated

Disassembling code: recovering symbol information 1

- When disassembling raw binaries, you don't have nice API names as with user-mode apps
- However, kernels often include symbol tables, to simplify debugging and provide better stack traces for crash dumps
- For a running Linux kernel, symbol table can be extracted by reading /proc/ksyms or /proc/kallsyms
- In 2.4 kernels a simple <address, name> table was used
- In new kernels a compressed format is used, see kallsyms.c
- Wrote a Python script to extract them

Disassembling code: recovering symbol information 2

- Other binaries might use simple symbol tables too
- E.g. VxWorks:

```
struct _SYMBOL {char *name; char *value; SYM_TYPE type; UINT16 group;};
```

- Search for "panic", "trace", "exception" or "assert".
- Often such functions get passed a function or source file name

```
panic("start_stop_timer", "start_stop_timer: timer %d not supported\n", id);
_assert("Assertion failed: pGraph!=((void *)0), CA_Connection.c, line 89\n");
```

- Once you identify some function names, search Internet for them, you might find where they came from
- If reversing a known OS, check official sites for demo/eval versions; even if for different processor, you can often match the code flow and identify other matches

Disassembling code

Demo

Conclusion

- Embedded reversing can be quite different
- However, some skills can be reused
- A lot of information is available if you know what to look for
- Can be a very rewarding process
- A goldmine if you're into vulnerability research
- Some links
    - igorsk.blogspot.com
    - www.lostscrews.com
    - mbed.org

Thank you!

Questions?