

Minimum, Single Threaded HTTP/TCP/IP implementation for boot loader

NIIBE Yutaka

<gniibe@fsij.org>



The Free Software Initiative of Japan

National Institute of Advanced Industrial Science and Technology, Japan

Agenda

- Who am I?
- The boot loader: g00ff (Go off!)
- GNU/Linux for Embedded
- Background
- Discussion (HTTP? Redboot? Requirement...)
- The Protocol Stack
- Usage of the stack
- How it works
- Summary

Who am I?

- Hacker around GNU, Linux, ...
- Founder of GNU/Linux on SuperH Project (1999)
- Chairman of Free Software Initiative of Japan (2003)
- Researcher at: National Institute of AIST, Japan
- Who hacks for embedded kernel technology:
 - Cache optimization for SH-4
 - Mutual exclusion technique (gUSA)
 - MMU usage improvement (exec-bit handling for M32R)
- Debian developer (2005)



The boot loader: g00ff (Go off!)

- Minimum firmware for M32R processor in Flash ROM. It supports following hardware:
 - Mappi2, Mappi3, M32700 (uT-engine), OAKS32R
 - NE2000-compatible, SMSC 91C111
- Written in C
- Loading Kernel from Compact Flash
- Loading Kernel through network
 - by HTTP
 - Using Minimum, single threaded HTTP/TCP/IP protocol stack

GNU/Linux for Embedded (... with my SuperH kernel) (1)

- Corega, Cbox: Home Web Server



- I/O Data, LANDISK: Disk Storage via LAN



- Kinkei System: Earth Quake Recorder



GNU/Linux for Embedded (2)

- Pixera: Wireless Router



- Elecom: Wireless Router with ADSL modem



- NTT-ME: Versatile Home Server



Background

The development of kernel for embedded system

- Cross-compilation environment
- Host machine — Target machine
 1. Edit kernel source code on Host
 2. (Cross-)Compile on Host
 3. Copy kernel to Target
 4. Reboot Target

Loading kernel by HTTP?

- While Etherboot support TFTP...
- For embedded engineers, setup of TFTP would be problematic (only for kernel load)
- HTTP servers are common, putting the kernel file is easy
- HTTP could be through WAN

Why not Redboot?

- Redboot requires C++
- Redboot is an eCos application
- Do we need to port another OS for boot loader?

Development Constraints for Boot Loader

- 'free standing' environment (no C library)
- Boot loader is the first 'application'
- Requirements should be minimum
 - Threads? — No
 - Interrupt driven? — Not necessarily
- We want to develop kernel!

Specific Purpose, Limited Features

- People love: General Purpose, Full Features
- Embedded guys have to handle: Specific Purpose, Limited Features
- It's like a HAIKU (Japanese style of poem)
 - Not adding features, but REMOVING as possible!

The Protocol Stack of g00f (1)

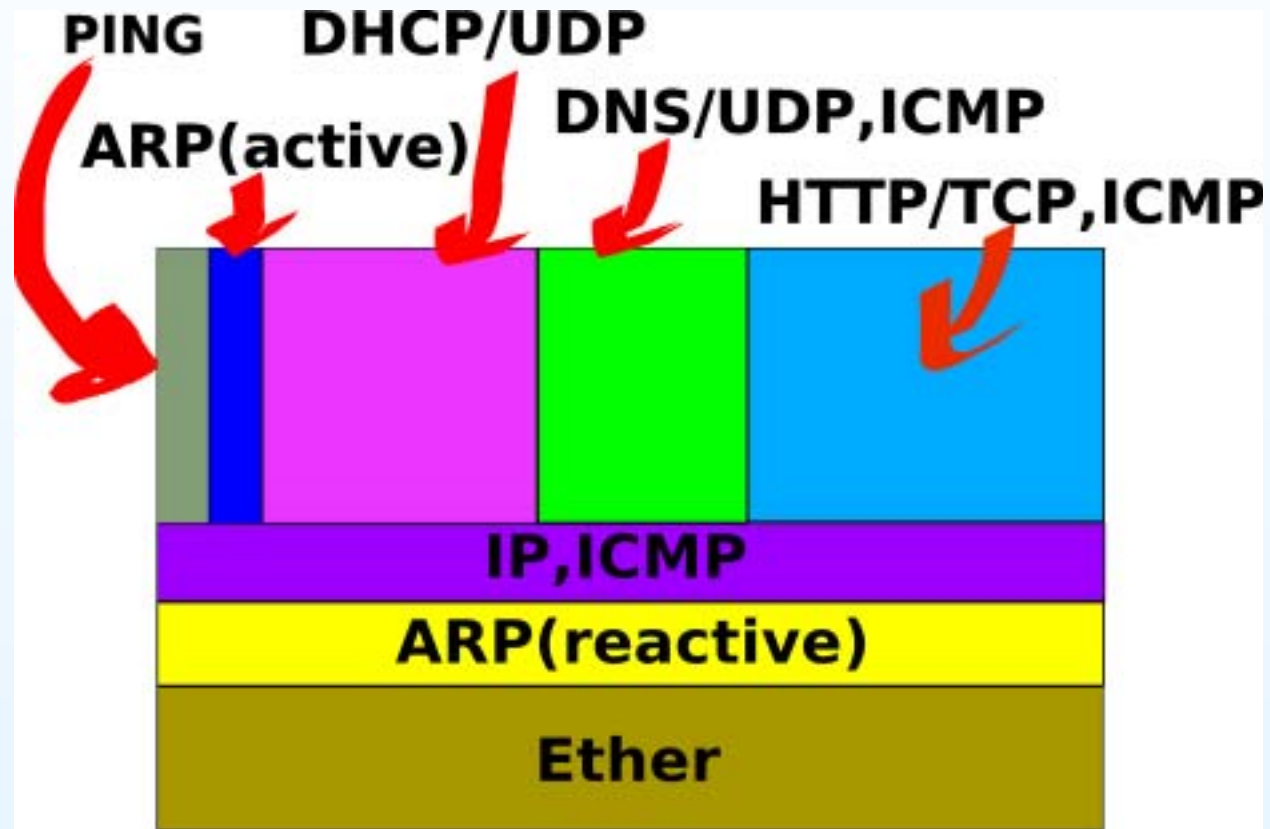
- Single threaded, polling
- No semaphore, no context switch
- No interrupt handling
- Specific purpose: Loading kernel only

The Protocol Stack of g00f (2)

- Single connection of HTTP only
- No support of urgent data of TCP
- Client side TCP/IP only
 - No retransmission needed (except first SYN)
 - No timer handling needed
 - No fancy features (slow start, congestion control) needed

The Protocol Stack of g00f (3)

- *Minimum* Network Protocol Stack



Usage (IPv4)

- Initialize Ethernet: `eth_init`
- Assign hard-coded IP or get it by DHCP:
`inet_aton` or by `dhcp`
- Configure network interface with IP:
`net_interface_config`
- Do ARP and get router's Ether address: `arp_resolv`
- Resolve domain name by DNS: `dns_resolv`
- Ping to server: `ping`
- Get kernel image by HTTP: `tcp_maton`

Usage (IPv6)

- Initialize Ethernet: `eth_init`
- Multicast setup: `eth_mc_setup_ipv6`
- Auto-configure Network address:
`net6_address_autoconfig`
- NDISC and get route: `ipv6_get_route`
- Ping to server: `ping`
- Get DNS information by DHCPv6: `dhcp6`
- Resolve domain name by DNS: `dns_resolvev6`
- Get kernel image by HTTP: `tcp_maton`

Endian specific binary handling

- `u8 peek_u8 (unsigned char *p);`
- `u16 peek_u16_raw (unsigned char *p);`
- `u16 peek_u16_ntoh (unsigned char *p);`
- `u32 peek_u32_raw (unsigned char *p);`
- `u32 peek_u32_ntoh (unsigned char *p);`
- `void poke_u8 (unsigned char *p, u8 v);`
- `void poke_u16_raw (unsigned char *p, u16 v);`
- `void poke_u16_hton(unsigned char *p, u16 v);`
- `void poke_u32_raw (unsigned char *p, u32 v);`
- `void poke_u32_hton(unsigned char *p, u32 v);`

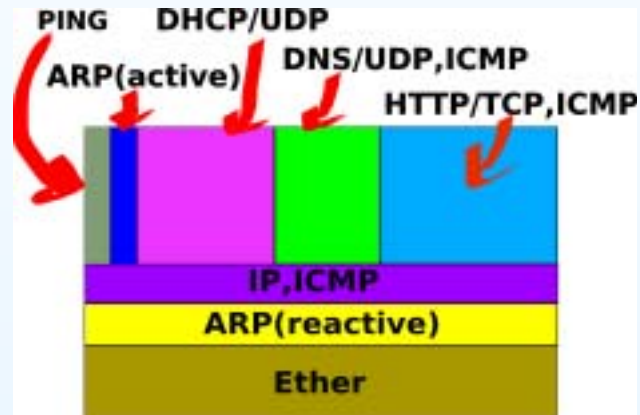
Implementation

- {HTTP/TCP, {BOOTP,DNS}/UDP, PING/ICMP}/IP, ARP: \approx 1500 lines
- {HTTP/TCP, {DHCPv6,DNS}/UDP, PING/ICMPv6}/IP, NDISC: \approx 1700 lines
- Ethernet driver (NE2000): \approx 600 lines
- Ethernet driver (SMC91c111): \approx 600 lines
- Binary handling: \approx 60 lines

How it works? (Driver)

Ethernet driver:

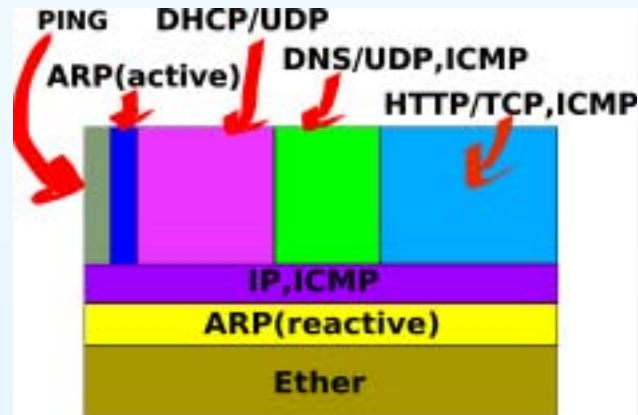
- Sending? \Rightarrow Send a packet
- or else \Rightarrow Receive a packet



How it works? (IP)

IP packet handling:

- Call Ethernet driver, then
- ARP? \Rightarrow ARP packet handling
- ICMP echo? \Rightarrow ICMP echo request handling
- Yup, we get a IP packet



How it works? (APP 1/5)

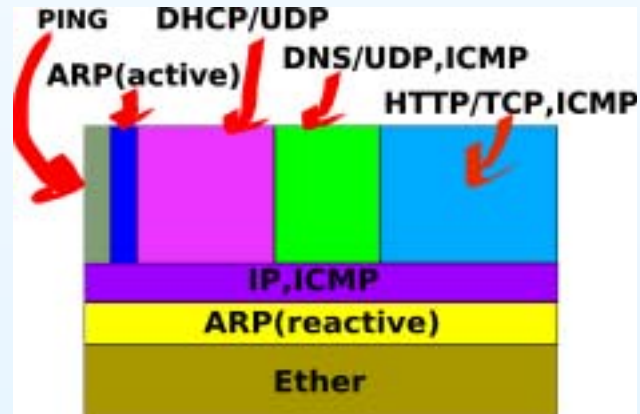
DHCP:

- Make DHCP discover to send
- Call IP handler
- do it again until we get reply, or timeout
- parse the packet
- Make DHCP request to send
- Call IP handler
- do it again until we get reply, or timeout
- parse the packet

How it works? (APP 2/5)

ARP Resolver:

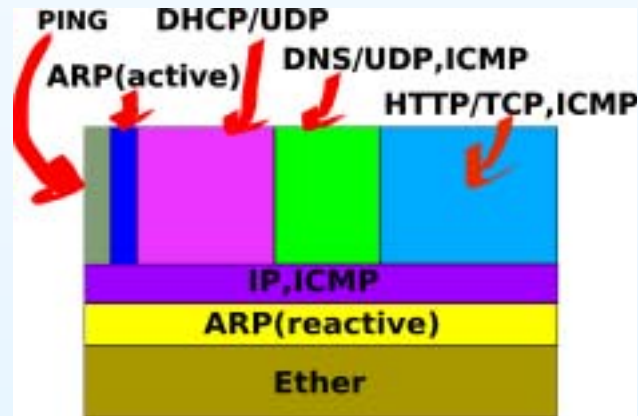
- Make ARP request to send
- Call IP handler
- do it again until we get ARP reply, or timeout
- parse the packet



How it works? (APP 3/5)

DNS Resolver:

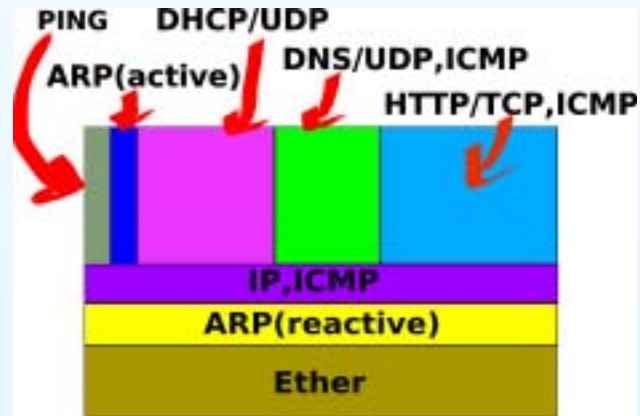
- Make DNS resolve request to send
- Call IP handler
- do it again until we get reply, or timeout
- parse the packet



How it works? (APP 4/5)

PING:

- Make ICMP echo request to send
- Call IP handler
- do it again until we get reply or timeout
- parse the packet



How it works? (APP 5/5)

HTTP/TCP:

- Make SYN to send
- Call IP handler
- ICMP unreachable? \Rightarrow failure
- do it again until we get reply or timeout
- parse the SYN+ACK packet
- Make ACK + data (of HTTP get) to send
- FIN-WAIT-1 loop:
 - Call IP handler
 - ICMP unreachable? \Rightarrow failure
 - Receive data, Make ACK to send
 - FIN? \Rightarrow transmission finish
- FIN-WAIT-2 or CLOSED handling

More information on g00ff and its stack

- M32R g00ff
<http://www.gniibe.org/software/m32r-g00ff-20060107.tar.gz>
- IPv4: Copyright (C) 2004 Free Software Initiative of Japan
- IPv6: Copyright (C) 2005 Free Software Initiative of Japan



- License: GNU GPL version 2 (or later)

Summary

The boot loader g00f££ and its protocol stack

- Specific Purpose, Limited Features
- Minimum requirement
 - No threads
 - No interrupt handling
- Minimum implementation
 - Layer violation is your friend

References

1. GNU/Linux on M32R Project
<http://www.linux-m32r.org/>
2. Etherboot
<http://etherboot.sourceforge.net/>
3. Redboot
<http://ecos.sourceware.org/redboot/>
4. GRUB 2
<http://www.gnu.org/software/grub/grub-2.en.html>