Network booting for Embedded Linux

Motivation:
The idea comes from a little x86 based device, which can boot itself using some sort of on board flash/EEPROM and has build-in tftp/UDP/IP stack to support download of embedded linux kernel from server. Since I can't get the hardware, I will use a floppy instead just for simulation purpose. The function is the same. I will use floppy to boot to dos mode, run the program I’m going to develop and then boot the Linux in local PC without using any local hard disk.

1. Introduction
Today, the hardware and chipset innovation is ongoing at a rapidly accelerating pace, accompanied by extremely rapid obsolescence of the older devices. At the same time, intelligent dedicated systems and appliances used in interface, monitoring, communications and control applications increasingly demand the services of a sophisticated operating system. The advanced capabilities are required, such as high resolution and user-friendly graphical user interfaces, TCP/IP connectivity, substitution of reliable and low-power flash memory solid-state disk for conventional disk drives, and so on. Combine these two, it has become an enormous challenge for commercial RTOS vendors to support the newest devices in a timely manner. It takes a large and constant resource commitment.

1.1 Using Linux in Embedded Systems
Embedded system developers are facing a dilemma: On one hand, today's highly sophisticated and empowered intelligent embedded systems--based on the newest chips and hardware capabilities--demand the power, sophistication, and timeliness of support provided by a popular high-end operating system like Windows. On the other hand, embedded systems demand extremely high reliability and the ability to customize the OS to match an application's unique requirements.

General-purpose desktop operating systems (like Windows) aren't well suited to the unique needs of appliance-like embedded systems. However, commercial RTOS, while designed to satisfy the reliability and configuration flexibility requirements of embedded applications, are increasingly less desirable due to their lack of standardization and their inability to keep pace with the rapid evolution of technology.
Linux offers powerful and sophisticated system management facilities, a rich cadre of device support, a superb reputation for reliability and robustness and extensive documentation. It provides both a basic kernel for performing the embedded functions and all the user interface bells and whistles you could ever want. Unlike Windows, Linux is inherently modular and can be scaled easily into compact configurations. Best of all, Linux source code is freely available, which makes it possible to customize the OS according to unique embedded system requirements.

All in all, in embedded and real-time applications where the OS is an underlying and hidden technology supporting appliance-like operation of a non-computer device, several key features of Linux are making it a growing preference among system developers:

- Source is available and free.
- There are no runtime royalties.
- Linux supports a vast array of devices.
- Linux is truly a global standard.
- Linux is sophisticated, efficient, robust, reliable, modular and highly configurable.

Given the cost effectiveness, configurability, reliability, and Internet capability of Linux, the Linux OS has become an increasingly popular choice as the embedded OS for embedded systems of many types. The use of Linux in embedded devices and systems has been exploding dramatically.

### 1.2 What is Embedded Linux

Embedded Linux is the use of the Linux kernel and associated software in embedded systems. "Embedded systems" are devices which contain the functionality of a computer, but which are not themselves perceived as computers. In most of these applications, the computational power is very minimal and often the computer consists of a single chip "microcontroller". Most electronic devices have embedded computers inside -- this includes everything from handheld calculators to automobiles to stereo equipment. Recently, the term "embedded computer" has been used to encompass new types of computing devices like handheld computers including Personal Digital Assistants ("PDAs"). Although this use of the term embedded computer applies to something that is considered a computer, the devices are not perceived as general purpose computers in the sense of the desktop personal computer (PC).

A minimal embedded Linux system needs just these essential elements:

- a boot utility
- the Linux micro-kernel, composed of memory management, process management and timing services
- an initialization process
To get it to do something useful and still remain minimal, you need to add:

- drivers for hardware
- one or more application processes to provide the needed functionality

As you add more capabilities, you might also need these:

- a file system (perhaps in ROM or RAM)
- TCP/IP network stack
- a disk for storing semi-transient data and swap capability

### 1.3 Small-footprint "embedded" Linux versions

For many embedded systems, the main challenge in embedding Linux is to minimize system resource requirements in order to fit within constraints such as RAM, solid-state disk (SSD), processor speed and power consumption. Embedded operation may require booting from (and fitting within) a DiskOnChip or CompactFlash SSD; or booting and running without a display and keyboard ("headless" operation); or loading the application from a remote device via an Ethernet LAN connection. There are many sources of ready-made small-footprint Linux. Among these are a growing number of application-oriented Linux configurations and distributions that are tuned to specific applications.

- **ETLinux** — a complete Linux distribution designed to run on small industrial computers, especially PC/104 modules.
- **LEM** — a small (< 8 MBytes) multi-user, networked Linux version that runs on 386's.
- **LOAF** — "Linux On A Floppy" distribution that runs on 386's.
- **uCLinux** — Linux for systems without MMUs. Currently supports Motorola 68K, MCF5206, and MCF5207 ColdFire microprocessors.
- **uLinux** — tiny Linux distribution that runs on 386's.
- **ThinLinux** — a minimized Linux distribution for dedicated camera servers, X-10 controllers, MP3 players, and other such embedded applications.

### 2. Linux-based Embedded System Development Model

In embedded systems, hardware and software are usually specifically designed and developed for particular applications. On the other hand, embedded systems are widely used in such a way that it extends over all the aspects of social life. It’s impossible and unnecessary to completely redesign and redevelop the whole systems. Most embedded systems have remarkable similarity that implies great reusability of hardware and software. To improve the development efficiency and shorten the development cycle
time, it’s important to make the most of the reusability.

We hope to have a development platform that can provide a friendly interface for embedded system application development. Basically, it is a Linux-based development solution that has the ability to:

- Reuse the hardware and software resources efficiently
- Facilitate the software development of application modules
- Satisfy the scalability of functionality
- Support the development and debugging utility efficiently

### 2.1 Target-Host Environment

To describe it shortly, an embedded system is the one that built for some dedicated application, like the cell phone or motion controller of some machine. It is built with limited computing resource and I/O device. Obviously it is usually hard to have some powerful/fancy/general application to run on it. As a result, another system (called the host) is needed to write the program of that simpler system (now called the target). The introduction of host system brings the solution of coding and compilation of embedded system applications.

A common target-host configuration is through communication lines such as ethernet, serial, parallel port or boundary-scan methods like IEEE 1149.1. The target-host configuration has two distinct advantages: 1. The entire project can be stored and managed in a designated desktop computer just like developing a desktop application. 2. No extra hardware is required on target to accommodate the development system. This makes the development of deeply-embedded system (e.g., very little memory space or no hard disk) much more feasible.

### 2.2 Target/Host Development Model

Common computer has good human-machine interface. Given some application development tools and environment, it can develop applications on itself. The embedded systems don’t have the bootstrap development capability. So even after the design completed, users usually can’t modify the program’s functions. A set of development tools and environment, generally based on the software and hardware of common computer, and all kinds of logic analysis apparatus, mixed signal oscillograph etc., are needed.

Choosing the embedded system design solution based on Linux, we can exploit some existent or self-developed software on Linux to build the embedded system development environment effectively. This kind of environment differs from the above-mentioned software/hardware emulation environment. It’s a debug/development environment directly built on hardware boards. Its development model is host/target model shown in the following figure.
In this model, target machine is an embedded mainboard (single board), on which running the Linux kernel and some debugging service processes. The Linux kernel includes the ethernet or serial port communication support module. Host machine is a common computer, storing the Linux kernel image, related configuration files, data files (file system), and compiled module object files for the target machine. Through network or serial ports, host machine can remotely monitor or control the system running on target machine, load or unload specifically developed real-time modules. This kind of host/target development way solves the problem of the unfriendly interface of embedded system software development.

The implementation of this host/target model needs the support of two-phase C/S model computing. Firstly, as a data/file server, host machine provides the required configuration data, kernel image and file system for target machine’s bootstrap. Here the target machine is running a remotely bootstrap procedure. After completing its bootstrap and starting the Linux kernel, target machine turns into a server, running some service processes. Then host machine can run remote control terminal software to manipulate the processes on target machine.

Based on this development model, we can develop applications on common host machine. Then load the application modules on target machine, run and debug. After the debugging of application modules is completed, we can solidify the real-time OS kernel, application modules and data into FLASH/ROM, the storage device of embedded system.

Adopting this Linux-based development model need to satisfy the following requirements:

- **Hardware board supporting**
  
  Hardware board should provide external serial port or network communication support. This kind of hardware board usually consists of embedded single board and communication extension board.

- **Basic software supporting**
  
  The supporting of configuration code (BIOS) of single board and system boot code is required.

---

**Figure 2.1 Host/Target Development Model**
3 Network Booting

There’re lots of work to do to design and implement the above-mentioned development model. In my project, I focus my work on implementing the key functionality of this model – remote bootstrap.

3.1 Network Booting Sequence

When a microprocessor first powers up, it begins executing instructions at a predetermined address. Usually there is some sort of read-only memory at that location, which contains the initial start-up or boot code. In a PC, this is the BIOS. It performs some low-level CPU initialization and configures other hardware. The BIOS goes on to figure out which disk contains the operating system (if the OS doesn’t exist on any disk, then it probably could be downloaded from networks), copies the OS to RAM and jumps to it. Actually, it is significantly more complex than that, but this is sufficient for our purposes. Linux systems running on a PC depend on the PC’s BIOS to provide these configuration and OS-loading functions.

In an embedded system, there often is no such BIOS. Thus, you need to provide the equivalent startup code. Fortunately, an embedded system does not need the flexibility of a PC BIOS boot program, since it usually needs to deal with only one hardware configuration. The code is simpler. It is just a list of instructions that jam fixed numbers into hardware registers. However, this is critical code, because these values need to be correct for your hardware and often must be done in a specific order. There is also, in most cases, a minimal power-on self-test module that sanity-checks the memory, blinks some LEDs, and may exercise some other hardware necessary to get the main Linux OS up and running. This startup code is highly hardware-specific and not portable.

In my project, my system takes over the control from the end of hardware initialization. After the system is started up and initialized, it checks the boot sector to load the OS. Since we suppose the limitation of hardware, there’s no hard disk in my prototype system and the OS is not located in local. So the network card begins to work. There is usually a boot ROM in the network card for this purpose.

In the general cases, the ROM code has several conditions placed on it.

- The first two bytes of the ROM must be 55 AA hex.
- The third byte of the ROM should contain the number of bytes in the ROM code divided by 512. So if the ROM code is 16kB long, then this byte would hold 20 hex (32 decimal).
- All the bytes in the ROM (specified by the length byte just mentioned) must checksum to 8 bits of binary zero.

If such a ROM is detected and validated by a scan, then the main BIOS does a far call to ROMSEG: 3, where ROMSEG is the segment of the ROM and 3 is the offset to transfer control to the discovered extension BIOS. Typically a network boot ROM does not take
full control at this point. Instead the normal procedure to do some initialization or probing of the hardware and then plant a vector that will be called when the BIOS is ready to boot the OS. The vector used for this purpose is normally interrupt 0x19 although interrupt 0x18 is sometimes used.

The ROM init code, which is assumed to be at offset 3, normally accomplishes the following tasks:

1. It initialises the network hardware so that it is ready to send and receive packets.

2. It sends a Boot Protocol (BOOTP) or Dynamic Host Configuration Protocol (DHCP) broadcast query packet. An alternative is Reverse Address Resolution Protocol (RARP).

3. Assuming a reply is received, the instructions in ROM decode the fields of the reply, sets its IP address and other parameters, and sends a Trivial File Transfer Protocol (TFTP) request to download the file. An alternative loading protocol is Network File System (NFS) protocol. In this instance a mount of the remote file system is done (with the bare minimum of features) and the boot file is read off the file system.

4. The file to be loaded is in a special format, it contains a "directory" in the first block that specifies where in memory the various pieces of the file are to be loaded.

5. It transfers control to the loaded image, which is Linux kernel in this case.

From this point, the loaded Linux kernel takes over the control and start up the OS, which is not covered in this report.

4. System Design

In my project, I use floppy to boot the computer and simulate the functionality of network booting by software. The system consists mainly of two parts: the main booting part, and the network downloading part. The main booting part is written by assembly language. The network downloading part is written by C language, using Borland C as the compiler. The main booting file is MyBoot.asm, which is compiled to MyBoot.com in order to be easily loaded and executed in the RAM. It is used to initialize some hardware registers and prepare the network card for downloading. The network downloading file is tftp.c.

Explanation of the main part code in tftp.c:

Function startup
(uchar *filename, EthInfoStruct * pInfoStruct)

Input:

uchar *filename;      /* Image file name */
EthInfoStruct /* Pointer to the global data input & output control structure; Type is declared in tftp.h; the corresponding declaration in assemble language is in boot.inc */

*pInfoStruct;

Output: void

Function description:
Invoke function tftpfile to get the OS kernel image named filename from the tftp server indicated by pInfoStruct through the network.

Function tftpfile
Input:
char filename; /* Object file name */
EthInfoStruct /* Pointer to the global data input & output control structure */
*pInfoStruct;
int offset; /* Offset, mainly considering for the segmentation transferring, but the image is transferred in a lump in current implementation */

Output: int
/* the next segment address offset image module should be put after exporting a segment for transfer. The value -1 means error in transmission (this will bring some different meanings, so we should change -1 to 0) */

Function description:
1. Invoke InitDP8390 (pInfoStruct) to initialize the NIC (in dp8390.c);
2. Invoke test-16 using loopback way to check whether remote DMA of the NIC is working normally;
3. Invoke function connect to send arp packet, get the hardware address of tftp server, preparing for latter using tftp protocol to download image data;
4. Then send tftp protocol data request packet, and intercept the data packet on the network; If the data packet is tftp protocol packet, then check whether it’s the packet wanted; If yes, then invoke function memload to copy the packet to the corresponding image assembling memory place, and request next 512B data packet; If not, request for re-sending the packet, until file is transferred and assembled completely.
5. Return the next transferring and assembling segment address offset (meaningful to segmentation transfer, requiring that the precedence transferred segment size is able to be divided by 16). Here has an error control code segment. If too much error, then return -1(should change to 0) directly.
Function memload

Input:

- ushort segNo; /* segment address for assembling, concept of real mode */
- ushort offVal; /* assembling object address offset*/
- uchar *dataBuf; /* Pointer to source data, relative segment offset in fact */
- uint size; /* Data size */

Output: void

Function description:

This is a strong assembling code segment, and it’s written by embedded ASM, because doing this assembling job in real mode. The assembling object address is 1000h+segNo<<4+offVal. We use two data selectors. 18h(es) denotes one selector, which base address is 0 and size is 4G-limited. 20h is the other selector, which base address is 90000h, for dataBuf to reference. The job of this function is to copy size Bytes data from 90000h+dataBuf to 1000h+segNo<<4+offVal.

Here also have a lot of protocol-related function code for sending request packets and receiving data packets, mainly for arp packets and tftp packets. And some protocol handling code for sending these packets (verify etc.).

Function connect

Input: EthInfoStruct *pInfoStruct;

/* Pointer to the global data input & output control structure */

Output: return void

Get the physical address of server from the structure which is pointed by pInfoStruct.

Function description:

Send arp request packet for the tftp server, wait for the response, get the physical address of tftp server, prepare for the latter packets transferring with tftp server.

Basic procedure:

1. Invoke arp_init to initialize tftp packet to send;
2. Invoke xmit_packet to send the data packet;
3. Invoke recv_packet to pool and receive data packet from the network;
4. If receive failed, repeat step 3; Until necessary (a counter decrease to 0), start again from step 1, 2, re-send the request packet;
5. Analyze the received data packet, check whether it is an ether packet; If yes, get the source physical address; If not, repeat step 3 to continue to receive; If necessary, start again from step 1,2, re-send the request packet (a counter controls);
6. Analyze the received data packet, check whether it is arp response packet; If yes, get the object physical address successfully and return; If not, repeat step 3 to continue...
receiving; If necessary, start again from step 1,2, re-send the request packet (a counter controls);

**Function arp_init**
**Input:**

```c
struct arp *ap; /* Pointer to Ether Frame buffer for sending arp packet */
EthInfoStruct *pInfoStruct; /* Pointer to the global data input & output control structure */
```

**Output:**
return void, *ap* points to the ready Ether Frame containing arp packet

**Function description:**
Initialize arp request packet Ether Frame. If *ap* is NULL, then return directly.

**Function tftp_rrq_init**
**Input:**

```c
struct rrq *tp; /* Pointer to Ether Frame buffer for tftp request packet */
char *filename; /* Pointer to request file name */
EthInfoStruct *pInfoStruct; /* Pointer to the global data input & output control structure */
```

**Output:**
return the size of tftp request Ether Frame, which is pointed by *tp.

**Function description:**
Compose the tftp request Ether Frame to request the file from tftp server.

**Function ack_init**
**Input:**

```c
struct ack *tp; /* Pointer to Ether Frame buffer for tftp response packet */
Short   block; /* Response packet no. */
EthInfoStruct *pInfoStruct; /* Pointer to the global data input & output control structure */
```

**Output:**
return void, *tp* points to the ready Ether Frame containing arp response packet.

**Function description:**
Compose the tftp response Ether Frame.

**Function tftp_ack**
**Input:**

```c
uchar *buf; /* Pointer to the received tftp packet */
```
short block; /* Response packet no. */
EthInfoStruct *pInfoStruct; /* Pointer to the global data input & output control structure */

Output: return void

Function description:
Response to the tftp packet of the corresponding block no.

5. Further Work
In my project, I used fixed IP for simplification. For more flexibility, we can use BOOTP(Boot Protocol) /DHCP(Dynamic Host Configuration Protocol) to obtain IP address dynamically.

An example of a BOOTP exchange goes like this:

DC: Hello, my hardware address is 00:60:08:C7:A3:D8, please give me my IP address.

BOOTP server: (Looks up address in database.) Your name is aldebaran, your IP address is 192.168.1.100, your server is 192.168.1.1, the file you are supposed to boot from is /tftpboot/vmlinux.nb (and a few other pieces of information).

In order to accomplish this task, there're more things to do with BOOTP/DHCP.