EuroDesign

EuroDesign
embedded technologies GmbH

Hardware platform:

SolidCard-Family

Contents:

HyperBoot Loader

HBL-03072001-A

Ausgabe Ie
Stand April 2003

Copyrights and Trademarks

EuroDesign embedded technologies GmbH
Waldstraße 4A
D-85414 Kirchdorf an der Amper

Contents:

Concept of the HyperBoot loader:

A *standard PC* uses the so-called BIOS (BASIC into output system) after switching on. This has the task, to recognize and initialize the connected peripheral equipments the available resources like CPU time speed, memory type and size.
This process is definitely meaningful for standard PCs, such a computer shall start and remain usable at modification of the configuration anyway furthermore, too. Because the possible configurations can be almost arbitrary, however, the BIOS must recognize and check all conceivable variants to be able to execute the initialization successfully. This process needs therefore a long time (10 to 20 seconds) independently of the speed of the found CPU. The time period until which the complete system is usable is correspondingly long.
In some cases, however, the automatic initialization fails and the user must configure system manually. The BIOS has a setup function in which many parameters will be initialized or in some cases changes will be make possible to this first.

The configuration of the hardware is always the same in an embedded PC yet: Neither the CPU changes, nor the memory type or size. The connected other periphery also changes for example only in the repair case. The recognition of the configuration is well superfluous and needs only unnecessary time.

Our HyperBoot loader starts here: The system will be initialized for the scheduled configuration. This needs only a minimum of the time which the automatic recognition would require. Expressed in numbers: The pure initialization only takes between 0.030 and 0.1 seconds. The operating system then can already be loaded and started.

Because the configuration is always the same the HyperBoot loader doesn't have menu, like a setup in BIOS.

Nevertheless for development, maintenance or repair in can be practical, to proceed on the boot process. The HyperBoot loader provides a debug mode for this by being able to be delayed or changed.

Since no display screen is needed in an embedded PC often, the HyperBoot loader is being usable also about the serial interface. If the HyperBoot loader doesn't find any graphics board, automatically it uses the serial interface for all inputs and outputs. If a graphics board but no keyboard is found, the outputs will be on the display screen, the input takes place via the serial interface.

The HyperBoot loader can primarily boots the free operating system Linux. However, it is also possibly to boot other software, for example programs without the need of an operating system. On the other hand it is not possible to boot an operating system which needs a standard BIOS. The functionality of a standard BIOS isn't supported by the HyperBoot loader.

Unlike a standard BIOS the HyperBoot loader already works in the x86 operating mode "protected mode".

Use of the HyperBoot loader

As a rule, you will hardly notice the HyperBoot loader. It runs only very short-time and then submits control to the loaded operating system.
To create this high-speed boot process, the boot device must, however, first be defined, prepared and initialized.

If a display screen is connected, you can observe the process directly. If the HyperBoot loader doesn't find any graphic controller, it uses the serial interface COM1 with the following parameters automatically:

- 19200 bauds
- 8 data bits
- 1 stop bit
- no parity bit
- no protocol

If a graphic controller will be found but no keyboard, the outputs will be send to the graphic controller, the inputs are expected from the serial interface, however.

The HyperBoot loader is able to load a standard Linux-binary image or also a HyperBoot binary file. The latter makes possible to load a command line and a RAM Disk besides the kernel. The HyperBoot binary file isn't limited on Linux but can load arbitrary programs.

When a graphics board and a display screen is available, the HyperBoot loader shows the following picture after switching the system on:

```
                    EuroDesign
              embedded technologies GmbH
            www.eurodsn.de info@eurodsn.de
             phone: +49 (0) 8166/99495-80
 ____________________________________________

Waiting on IDE harddisk ....
Harddisk type: DC108
capacity: 540 MByte (LBA)
497 KB Image size
.............................................
```

In this case an IDE harddisk is provided to boot from. The dots display the progress of the loading process. It has to be taken into account that a boot process from harddisk can be delayed by starting process of the harddisk by itself.
The operating system immediately begins to run after loading. The function of the Hyper-Boot loader is ending at this point.

Initialize the boot devices

So that the HyperBoot loader can load an operating system, the scheduled device must be prepared for this.

The following devices are suitable for a boot process as sources:

- 1.44 MB floppy disks
- IDE harddisks or compatible FlashDisk drives
- DiskOnChip flash
- Flash ROM

It is expected to all boot devices that the binary image is written linearly. i.e. no file system may be available there. Except for the floppy disk and the flash ROM it therefore will be required to partition the respective device to be able to use a normal file system.
At this at least two partitions are created:

- one for the binary image of the operating system (as a rule around 1 MB large)
- a further one for the file system

The binary image is written linearly into the first partition, the second partition is formatted as normal with the desired file system.

Here one example with an IDE harddisk:

This IDE harddisk is the device "hda" from view of Linux.
With

```
fdisk /dev/hda
```

you can partition the harddisk now.

Create a first primary partition, choose as a size about 1 MB. Then create a second primary partition which for example covers the rest of the disk. The first partition is "hda1" the second partition "hda2" now.
The operating system binary image gets the device "hda2" typed in as a root device. It then can

```
dd bs=8192 if=vmlinux of=/dev/hda1
```

being written to the 1st partition linearly.
With

```
mkfs.ext2 /dev/hda2
```

you create a file system (in this case the "extended 2" format) on the 2nd partition. You can copy all other files needed by the operating system on this file system now.

The in principle same approach is necessary at the use of a DiskOnChip. In this case it is the device "nftla" or "nftla1" and "nftla2" from view of Linux which have to become partitioned, however.

Divergent use at 1.44 MB floppy disks and flash ROM:

1.44 MB of floppy disks don't become partitioned. It contains the standard Linux or the HyperBoot binary image linearly directly.
With

```
dd bs=8192 if=vmlinux of=/dev/fd0
```

any time a new system can be written on floppy disk and then be loaded by the HyperBoot loader.

The flash ROM is organized differently so that its contents cannot be changed by Linux. To be able to make modifications here, you must go back to the functionality of the integrated debugger in the HyperBoot loader. Read the details to this in the chapter "HyperBoot from the flash ROM".

Selection of the Boot Device

After you have prepared one or more devices for the boot process, you must select it now so that the HyperBoot loader has access on it. To take this you have to bring the HyperBoot loader into the maintenance mode.
If you use our evaluation kit, you must take the microswitch to a certain position for the maintenance function respectively. Looking up this information in the accompanying manual from your evaluation kit.

The contents of the start display screen will change in this operating mode. Now the system will not boot immediately. It reports a number of found binary images and waits until you select one of them to boot from.

If no bootable binary image was found, the following screen will be displayed:

```
                    EuroDesign
              embedded technologies GmbH
            www.eurodsn.de info@eurodsn.de
             phone: +49 (0) 8166/99495-80
_____________________________________________

No loadable image found!
Set default image: d <Input>
Starting debugger: q
Enable sources: e
Save settings: s
>
```

In this case either no binary images were copied on the device, or the respective device hasn't been switched free.

If binary images were found, the contents of the start display screen will change:

```
                    EuroDesign
              embedded technologies GmbH
            www.eurodsn.de info@eurodsn.de
             phone: +49 (0) 8166/99495-80
_____________________________________________

Image: 0 standard image -> Floppy
Image: 1 partition 0 -> IDE

Load image: r <image>
Set default image: d <Input>
Starting debugger: q
Enable sources: e
Save settings: s
>
```

In this case a loadable image was found on the inserted floppy disk and on the harddisk. With the input

```
r <image number>
```

this can be loaded now. Well,

```
r 0
```

loads the Kernel from floppy disk,

```
r 1
```

from harddisk at partition 1.

If you want to load and run image immediately from the standard connected IDE harddisk in future, now choose the menu item

```
d <image number>
```

To save this setting, you must then still enter

```
s
```

Make a test by pushing the reset key now. You get a display screen with the same contents as before, but the IDE entry has, however, a preceding „*" symbol now. In this way the HyperBoot loader shows that this binary image would be immediately loaded, if the HyperBoot loader is no more in the maintenance mode.

If your binary image isn't found, it also can be that this device isn't free switched. i.e. the HyperBoot loader doesn't search this device for binary images and either doesn't list the there situated binary images consequently.

In this case enter the letter

```
e
```

You receive a list of the possible devices now:

```
                    EuroDesign
              embedded technologies GmbH
            www.eurodsn.de info@eurodsn.de
              Tel.: +49 (0)8166/99495-80
_____________________________________________

Image: 0 standard image > Floppy
Image: 1 partition 0-> IDE

Load image: r <image>
Set default image: d <input>
Start debugger: q
Enable sources: e
Save settings: s
>e
1 = disable Floppy
2 = disable IDE
3 = enable DOC
4 = enable flash ROM
```

Either you can disable the source, if it was free switched or enable it in other case by input a number. A modification that you make here will becomes true at the next system start or RESET. However, changes must be saved by input

```
s
```

otherwise the modification will be lost.
If you have enabled your desired device and saved the binary image correctly there, it will be found in future and can be loaded from there.

End the maintenance mode so that the HyperBoot loader will execute the high-speed boot process.

Shortcuts

r <no.>:
load one of the images listed above

d <no.>:
select one of the images listed above as default loading.
In future, this will loaded immediately.
e:
create a list with all possible devices by which an image can be loaded.
The device is selected by number (lock or unlock).

s:
save modifications permanently changed with "e".

q:
start integrated debugger.

Other functionality

If the system is in the maintenance mode, you can change to the integrated debugger by input of

q

besides selection of a boot binary images. This provides some basic functions to get information from hardware of the system or make changes to it.

You will return from debugger into the HyperBoot loader only by restarting the system.

You will get a list of all possible instructions by pressing a

?

Depending on SolidCard you will get different details. The list represented here dates from a SolidCard II.

```
Display                 D [start address] [ending address]
Enter                   E start address value [value]
Read E/A                I address
Write E/A               O address value
Filling                 F address value number
Test memory             T <start address> <length> <no. trials>
Reading                 R area address
                        5 <bwd> SC520 MMCR
Write                   W area address value
                        5 <bwd> SC520 MMCR
PAR contents            PA [l]
Display PCI Config      PI
Load IDE block          LI <sector number>
Load FDC block          LF <sector number>
Linux-Image:
From Disk into flash    LS
Displaying list         LL
Delete in flash         LE <number>
Compress flash          LC
All addresses have to be input linearly and hexadecimally!
```

parameters into tips clips ('< >') are required, in square clips ('[ ]') are optionally.

Display a memory area
Command: D [start address [ending address]]

Display the memory area in tabular form beginning from *start address* to *ending address*. If the *ending address* is left out, 256 bytes are displayed beginning at *start address*. If both parameters are left out, the next 256 bytes are shown, beginning at *ending address* of the previous call.

Input data values to memory area
Command: E <start address> <value> [value2] ...

Put *value* in the memory beginning at *start address*. More than a *value* can be given. *Start address* is incremented by one with every *value*.

Read one byte from I/O addressable storage
Command: I <address>

Read a byte from *address* and display this *value*. *Address* is valid into area from 0...FFFF.

Write one byte into the I/O addressable storage
Command: O <address> <value>

The byte *value* will be written in *address*. *Address* is valid into area from 0...FFFF, *value* from 0...FF.

Filling a memory area
Command: F <address> <value> <number>

Fill the memory with *value* starting at *address*. Becomes filled up to *address+number-1*

Testing a memory area
Command: T <start address> <length> <number of repetitions>

Tests a memory area, whether arbitrary bit patterns can be saved into this. The test will begin at *start address*, is executed on *length* and repeated by *number of repetitions*. If the *number of repetitions* is set to 0, the test will repeated infinitely.

Functional principle of the test: The memory area is divided into long words (32 bits). Initially an afferent value is written in every long word beginning with 0. After this the memory area is checked for these results. If repetitions are scheduled, the memory area is written with afferent long words, starting with 1 this time, however. At every repetition the start value is incremented by one. Well, if infinite repetitions are demanded, every memory location is written and checked with every possible pattern. Address bit errors (soldering defects etc.) are recognized in this way certainly.

The test will stopped at a bug and the faulty memory address will be output.

Please noticing you, that today's processors have a cache memory. So that the external memory is really written and read, the length tested must exceed the size of the cache. Otherwise only the internal cache would be tested, but not the external memory, however.

Summarized reading (SolidCard 1)
Command: R <area> <offset>

To be able to read some configuration registers, certain operations must be executed without interruption. This provides the instruction "R". It reads an 8 bit value from the *offset* of the indicated area.
area informs the debugger which area shall be selected:

4	SolidCard 1 chip set register
C	SolidCard 1 graphic register of the CGA emulation
M	SolidCard 1 graphic register of the MDA emulation

Example: R 4 0 reading bank 0 DRAM configuration register
R C E reading the high address of the CGA cursor

Summarized reading (SolidCard 2)
Command: R 5 <b | w | d> <offset>

Read a byte (8 bits), word (16 bits) or double word (32 bits) from "Memory Mapped Configuration Register"-area of the SC520 chip set. Because the re-

gisters length isn't standard in this area, the word length must be set in front of the *offset* parameter. The *offset* is the MMCR offset of the data sheet. The true access address (beginning at $FFFE0000_{16}$) is completed automatically.

Example: R5 b 2 reads a byte from MMCR offset 2 (cache setting)
R5 w 0 reads a word from MMCR offset 0 (chip revision)
R5 d 8C reads a double word from MMCR offset 8C (PAR1)

Summarized writing (SolidCard 1)
Command: W <area> <offset> <value>

To be able to write some configuration registers, certain operations must be executed without interruption. This provides the instruction W. It writes an 8 bit value in *offset* of the indicated area.
Area informs the debugger which area shall be written:
4 SolidCard 1 chip set register
C SolidCard 1 graphic register of the CGA emulation
M SolidCard 1 graphic register of the MDA emulation

Example: W 4 80 writes bank 0 DRAM configuration register
W C E writes the high address of the CGA cursor

Summarized writing (SolidCard 2)
Command: W5 <b | w | d> <offset> <value>

Write a byte (8 bits), word (16 bits) or double word (32 bits) to the "Memory Mapped Configuration register"-area of the SC520 chip set. The parameters have nearly the same structure than "summarized reading", only that a value is still added here. The bit breadth indicated in the first parameter is used for writing independently of the breadth of value. For example the *value 'FF'* is written as double word (000000 FF_{16}) when first parameter is *'d'*.

Display current status of PAR registers (SolidCard 2)
Command: PA [I]

Display the current results of the SC520-PAR registers. These will not be shown in binary but in readable form. Without the parameter *I* a short table is displayed, with parameter *I* you get a detailed list with all settings.

Display devices at the PCI bus (SolidCard 2)
Command: PI

List the devices connected at the PCI bus and show the resources like memory and E/A areas and Interrupt channels.

Read and display a sector from IDE harddisk
Command: LI <sector number>

Reads selected sector from IDE harddisk (master) and show the contents at the display screen.

Read and display a sector from 1.44 MB floppy disk
Command: LF <sector number>

Reads selected sector from a floppy disk and show the contents at the display screen.

Most features should be self-explanatory. Please consult our support (info@eurodsn.de) if you have questions to these instructions.

HyperBoot from the flash ROM

Besides the boot process from an IDE-Flashdisk or a DiscOnChip one of the most high-speed methods to start an operating system is to start system from the flash ROM. A 2 MB NOR flash is integrated, which is linearly displayed into the addressable storage of the CPU makes possible a very high-speed access. Different size is optionally. Please, if you like to use it, contact us before the purchase of a SolidCard 2.

To be able to use the flash ROM for the boot process, at first you must activate the maintenance mode. Then you change to the integrated debugger with

q

The instructions

LS
LL
LE and
LC

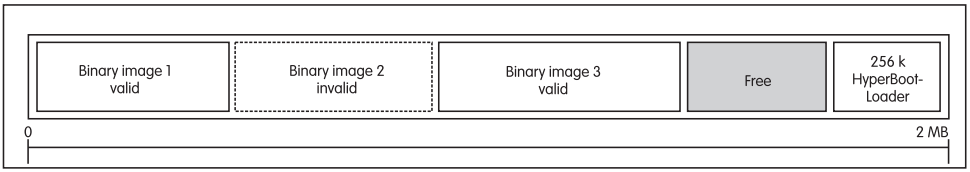
are scheduled to manipulate the contents of the flash ROM.

LL: Lists all available binary images and shows the remaining space in the flash ROM.

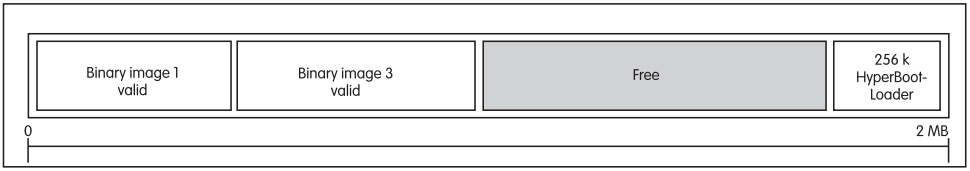
LS: Reads a binary image from 1.44 MB floppy disk and copy it to a free space in the flash ROM. An error message appears if there is not enough free space in the flash ROM.

LE: Deletes a binary image in the flash ROM. With this the binary image will be set invalid, not deleted yet. It cannot, however, be valid explained again.

LC: Summarize valid binary images and free areas in the flash ROM. This is required, if you delete the second for example from three available binary images in the list. Because new images only are appended to the end of the list, it can be despite deletion that enough space isn't available for the new binary image. The following graphic shall clarify this: binary image 2 is invalid, together with the free area enough memory would be available for another binary image.



The following picture demonstrate function of instruction LC



Now sufficient coherent memory space is available for another binary image.

Make bootable binary images

The HyperBoot loader can handle two kinds of binary images:

- Standard Linux-binary image
- HyperBoot binary format

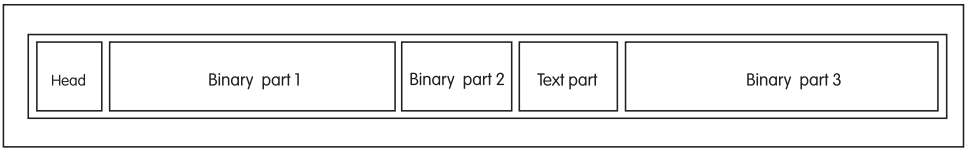
Both kinds of binary images can be read from floppy disks, DiscOnChip or harddisks. From flash ROM only the HyperBoot binary format is readable.

You create the standard Linux-binary image as normal.

`make zImage` or `make bzImage`

creates the necessary and utilizable file. It is disadvantage of this method that only the real kernel is loaded and cannot want to submit any kernel commands either. In this case the always necessary root file system must already be available.

The HyperBoot binary format also makes possible loading some command line information and a RAM Disk next to loading the kernel by itself.
For this such a binary file has the following structure:



The individual elements (of a Linux system):

- Head: contains some detection and administration technical information
- Binary part 1: Contains the pure kernel (without setup etc.)
- Binary part 2: Contains the setup part of bootable Linux kernel
- Text part: The text which is submitted as a command line at the starting kernel
- Binary part 3: Contains a RAM Disk-Image

The following lines will explain the structure of the head, written in programming language 'C':

```
struct HYPERBOOT
{
        unsigned char marks [20];
        unsigned Long ver;
        unsigned char name [160];
        unsigned Long totalBlocks;
        unsigned Long headerSize;

        unsigned Long kernelBlocks;
        unsigned Long kernelSize;
        unsigned Long kernelTarget;
        unsigned Long kernelStart;
        unsigned Long kernelFlags;

        unsigned Long paramBlocks;
        unsigned Long paramSize;
        unsigned Long paramTarget;

        unsigned Long commandBlocks;
        unsigned Long commandSize;
        unsigned Long commandTarget;

        unsigned Long rootBlocks;
        unsigned Long rootSize;
        unsigned Long rootTarget;
};
```

The contents of the single entries:

- **mark:** represents an identification, which will be evaluated by the HyperBoot loader to classify the found file. A HyperBoot-binary image contains the character string "EURODESIGN_HYPERBOOT" here, 0 completing without one, however!
- **ver:** contains a release number to be able to redefine the structure in future. The value 0x00010000 represents version 1.0, 0x00010001 version 1.1
- **name:** About this a title can be given to the binary image. This text is reported in the maintenance mode.
- **totalBlocks:** Number of the sectors at 512 bytes covered by this file.
- **headerSize:** Number of sectors which are covered by this head. Currently 1.
- **kernelBlocks:** Number of sectors which the kernel covers.
- **KernelSize:** Size of the kernel in bytes
- **kernelTarget:** Destination address of the kernel in the memory. Linear 32 bits number.
- **kernelStart:** Start address at which the kernel shall be started. Linear 32 bits number.
- **kernelFlags:** Information about the kernel. Currently unused.
- **paramBlocks:** Number of sectors from Setup part.
- **ParamSize:** The size Setup part in byte
- **paramTarget:** Destination address of the Setup part in the memory. Linear 32 bits number.
- **commandBlocks:** Number of sectors which the text of the command line covers.
- **commandSize:** Size of the text of the command line in byte.
- **commandTarget:** Destination address of the command line. Linear 32 bits number.
- **rootBlocks:** Number of sectors which the RAM Disk covers.
- **rootSize:** Size of the RAM Disk in byte
- **rootTarget:** Destination address of the RAM Disk in the memory. Linear 32 bits number.

The structure is adapted to the requirements of the Linux operating system. However, an arbitrary other program also can be loaded with that.

For example: to change a rescue system into a HyperBoot-binary image from a command line, a minimal kernel and a RAM Disk with necessary tools, you need the following files:

- mImage
- create the binary image from the following files
- arch/i386/boot/setup
- arch/i386/boot/bootsect
- arch/i386/boot/compressed/vmlinux.out
- a text file with the contents of the command line
- the Image of a RAM Disk

If the text file name is "sc2command.txt" and the RAM Disk name is "image.gz", mImage can be invoked as follows:

```
mImage  -k linuxRAM.out 0x1000 0x1000
                -p bootsect setup /dev/ram0 0x90000
                -c sc2command.txt 0x90800
                -r image.gz 16777216
                -n "rescue system with RAM Disk"
                -o sc2rescue.bin
```

This call creates a Linux with a RAM Disk as a root-device, names this Image with "rescue system with RAM Disk" and writes the result into the file "sc2rescue.bin".

In which:

`-k <kernel name> <destination address> <start address>-`

kernel name: For example /usr/src/arch/i386/boot/compressed/vmlinux.out

Don't use "vmlinux", the boot sector and the Setup part are already integrated there!
Destination address: At a zImage 0x1000, at a bzImage 0x100000
Start address: At a zImage 0x1000, at a bzImage 0x100000

`-p <boot-sector> <setup> <root device> <destination address>-`
Boot sector: For example /usr/src/arch/i386/boot/bootsect
Setup: For example /usr/src/arch/i386/boot/setup
root-device: Name of the device file which the kernel shall consult as a root-device
Destination address: At present always 0x90000

`-c <batch file> <destination address>-`
Command file name: Name of a file with ASCII contents
Destination address: At present always 0x90800

`-r <RAM-Disk-Name> <End of memory area>-`
RAM Disk Name: Name of the file with the RAM Disk-Image
End of the memory area: Because the RAM Disk-Image shall be stored at the end of the available memory here the available main memory of the target system is to be indicated. Make sure that it isn't meaningful to operate such a RAM Disk at a memory consolidation of only 2 MB.

`-n <Description of binary image>-`
You can indicate a description (max. 160 characters). This is reported at loading or selection.

`-o <filename of binary image>-`
You can define a name of the HyperBoot-binary images to be created over this button.

Be care, that a blank must be input between all buttons and parameters.

System state at the transition from the HyperBoot loader to the operating system

The HyperBoot loader isn't running absolutely alone on Linux in which it was conceived, however. You also can load and take another operating system or also a simple program with loading HyperBoot-binary image. This chapter describes the status of processor and other hardware at the end of the HyperBoot loader after preparing to start operating system or program.

Periphery: LPT: Standard.
COM1 ... 4: Standard
DMA controller: Standard
Keyboard controller: Standard
Floppy controller: Standard
Timer: Standard

Real time clock: This is active, the CMOS RAM, however, doesn't include any data. The HyperBoot loader doesn't use the CMOS RAM.

Interrupt controller: These are active, initialized, however, just like Linux uses them.

Processor: The running mode of the HyperBoot loader is the protected-mode. The IDT starts at the physical address 0x00032 with the length of 1024 bytes, the GDT address 0x0000C with the length 32 bytes.

Memory: The complete memory is at disposal by an application. The classic gap in the area of 0xA0000...0xFFFFF doesn't exists any more, if the chip set will supported it. In this area normal working memory will be found. All memory addressable peripheral equipments lie above the working memory.

Glossary:

Boot device:	For example floppy disks, harddisks, ROM memories by which a program or operating system can be loaded.
Boot process:	Load and start a work surroundings, for example an operating system.
Command line:	Word-by-word commands which a Linux kernel can process. Serve the configuration of the kernel.
DiscOnChip:	A NAND flash technology of the company MSystems. The use of a file system facilitates without mechanical parts. Harddisk replacement.
File system:	By the principle of sectors procedures must be available to manage logically files with more than one sector. i.e. structures will be needed which guarantee restoring a file from a number of sectors. A whole number of file management systems exists, each with special qualities.
HyperBoot:	Special kind of initialization and start a PC system. Don't need the use of a standard BIOS.
HyperBoot binary file:	Is a binary file combined of several parts which contains all necessary parts of the operating system. However, don't need contain any active boot loader for the boot process (see standard Linux-Binary image). Copying from boot device into the working memory is made by the HyperBoot loader completely.
Kernel:	This is the description of a program and its data which the real operating system represents.
Linear:	Floppy disks and harddisks save its data in numbered sectors at its memory device. Through this files must be stored on several sectors if it is larger than fit to one sector. Linearly means that increasing data file will be stored also in increasing numbered sectors. To reconstruct the file it suffices, to read the sectors increasing beginning from sector 0. At use of a file system this linear cohesion doesn't exist.
NAND flash:	Block oriented memory. Possible device for saving data. A processor can not execute a program in this device directly. First data must be copied block way to memory to execute from there.
NOR flash:	A processor has linearly access on it, so using as a program memory is possible.
Partitions:	If a harddisk is divided up to several independent areas, the single areas are called partitions. Every partition can be formatted differently or also contain another operating system. Every device can contain primary partitions up to 4. Besides this an extended partition which then can contain 4 partitions in turn can be created. As a rule, you can only boot from primary partitions, extended partitions serve the data taking mostly.
Protected mode:	One of the operating modes in which a x86 processor can process programs. In this operating mode a program has available use of complete power of processor and memory space.
RAM Disk:	Contain a file system and data like other devices, however it will be completely in the working memory. This is meaningful for rescue systems or the first starting of a board, if no data are available on the connected peripheral equipments yet.
Real mode:	One of the operating modes in which a x86 processor can process programs. In this operating mode the processor works like a 16 bit CPU and have access to 1 MB of the addressable storage.
Sector:	The smallest administration device of a floppy disk or harddisk in which data can be filed. Most of 512 bytes of capacity.
Standard Linux binary image:	Contains real mode boot loader and the operating system image. The Bootlader can load and start the operating system under a normal PC BIOS.
x86:	Expression to a processor family, under this one you find 80386, 80486, pentium, pentium II...IV. Originally, developed by the Intel company.