# Documentation for SCSI controller project
# Target module
# Hardware V2.0 / Firmware V1.0

2006−06−17 / Michael Bäuerle <micha@hilfe−fuer−linux.de>

## Preamble
The goal of this project is a general purpose parallel SCSI controller[1] for hobby usage. It was designed to be modular and SCSI3 compliant.

## Description
The Target module handles the high level functions like the SIP (**S**CSI **I**nterlock **P**rotocol), the logical units and their device model.

The Target module implements the following components:
- PIA (**P**arallel **I**nterface **A**gent) driver
- SIP message handler
- Taskrouter
- Taskmanager for LUN (**L**ogical **UN**it) 0
- Deviceserver for LUN 0
- Logical block access routines
- PCMCIA (**P**ersonal **C**omputer **M**emory **C**ard **I**nternational **A**ssociation) driver
- ATA (**A**dvanced **T**echnology **A**ttachment) driver

## Hardware Features
Hardware V2.x is based on an Atmel ATmega64−16 AVR microcontroller. The SCSI ID must be configured via the jumper block JP1.
Closing jumper JP30 forces a hard reset on all modules (but not on the SCSI bus). An external SCSI bus reset does NOT force a hard reset on all modules immediately (as with hardware V1.x), instead the Target module executes a soft reset.
PCMCIA memory cards are used as media. Firmware V1.x supports "Linear SRAM" and "ATA Flash−EPROM" cards. Other "Linear" memory cards should at least be readable.

## Firmware Features
The Target firmware correctly handles commands to nonexisting LUNs and implements all commands that are mandatory for Type 0 ("direct access") devices on LUN 0. Some optional commands are also supported. Disconnect/reconnect is currently not implemented. Linked commands and TCQ are not supported (requires more ressources than the ATmega64 can provide).

The firmware supports all mandatory features that are required by the SCSI3 standard. Nice to have for future version would be support for:
- Disconnect/reconnect (Impossible with PIA firmware V1.x)

[1]) The circuit that controls a SCSI device not the hostadapter

Content



## 0. RAM usage

Hardware V2.0 is equipped with 4KiByte RAM memory. RAM memory is used as shown in the following memory map:

| ***Address*** | ***Usage*** |
|---|---|
| 0x10FF<br>variable | Stack (should be approx. 2KiByte) |
| | Gap (Unused space to which heap and stack can dynamically grow) |
| variable<br>see below | Heap (Used for ’malloc()’ function call) |
| see below<br>0x0100 | Global and static variables |
| 0x00FF<br>0x0020 | I/O address space |
| 0x001F<br>0x0000 | Register file |

Grey: Register file & memory mapped I/O
Green: Always allocated memory
Yellow: Memory that is dynamically allocated

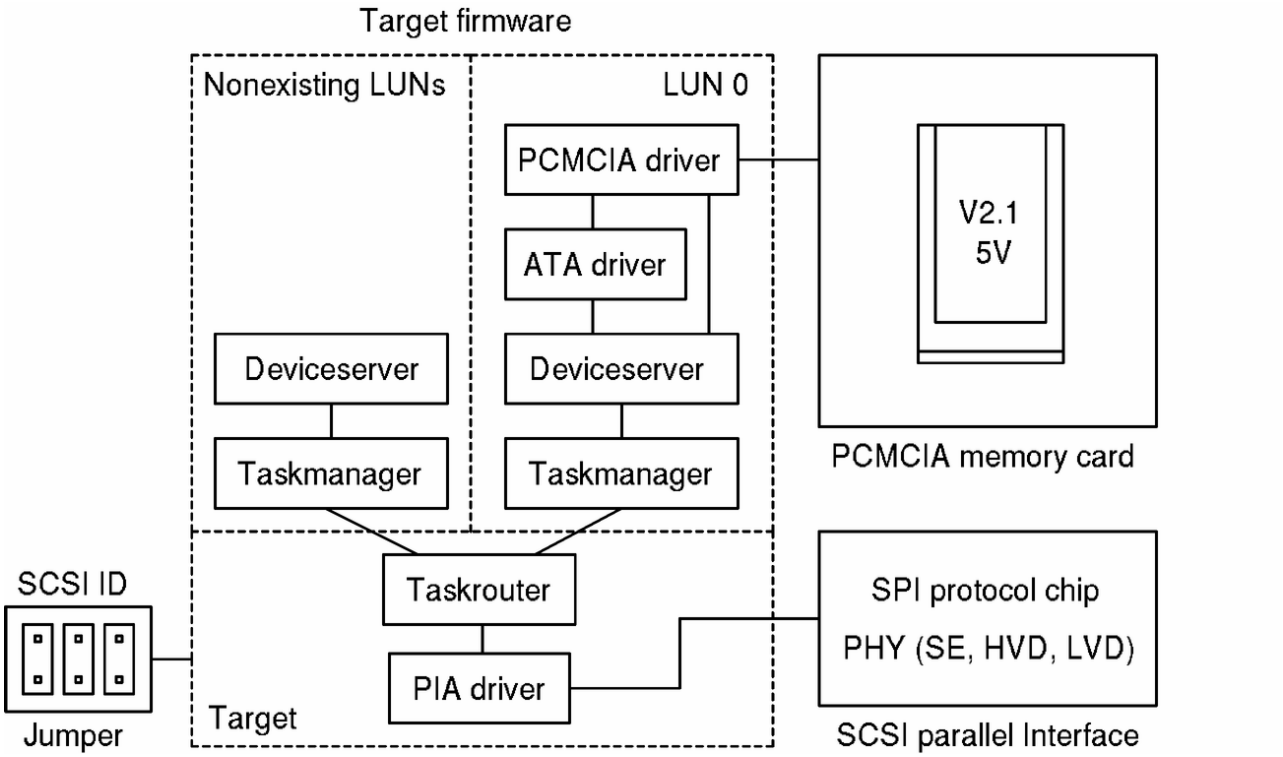
The heap may be used internally by the libc for stdio−functions like ’printf()’ and ’scanf()’. The memory returned by ’malloc()’ is also allocated on the heap.

The stack is used for local variables and return values. Note that the local block buffer is allocated on the stack with ’BLOCKSIZE’ bytes (Default: 512)!

If the debug system is enabled, a global message buffer ’sbuf’ is allocated with ’BUFFERSIZE’ bytes (Default: 768).

1. Abstract

The target module as block diagram:




**Nonexisting LUNs:**

The deviceserver for nonexisting LUNs must exist to execute INQUIRY commands that reflects the correct status of LUN1 − LUN31 ("LUN not supported" in this case). Otherwise commands to all LUNs are executed by the same deviceserver that report the status "LUN available" ... this would lead to multiple detection of the same LUN and probably multiple OS drives for the same physical device.

**SCSI ID:**

The SCSI ID jumpers are sampled only once after POST. The ID is used for PIA configuration and resides in PIAs memory from now on.

**PCMCIA memory cards for LUN 0:**

Any type of PCMCIA V2.1 memory card can be used (only SRAM and ATA Flash−EPROM cards have been tested) as long as the card is compatible with:

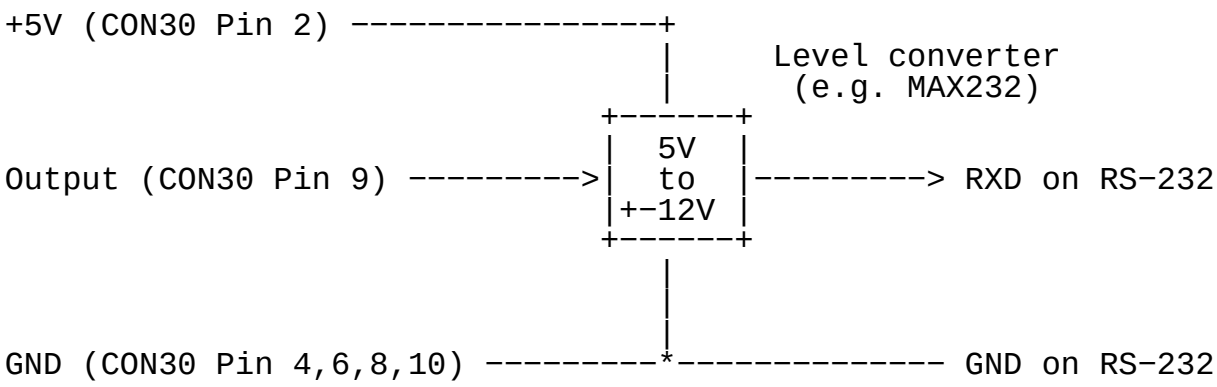
- 5V supply voltage ($V_{CC}$)
- 5V programming voltage ($V_{PP}$)
  PCMCIA board V1.x also supports 0V and 12V programming voltage but Firmware V1.0 always set it to 5V!
- If the card contains an ATA controller, LBA mode must be supported

"3V only" cards do not fit into the socket. Cardbus cards are not supported!
Cards with linear memory are mapped to logical blocks of 'BLOCKSIZE' Byte size without reserved space. Logical blocks on cards with ATA controller are mapped 1:1 (SCSI LBA is equal to ATA LBA).

2. Debug system

**How to connect the terminal**

The target firmware contains a debug system that can print messages to a VT100 compatible terminal connected to CON30. To use a RS−232 terminal (or emulator on a PC) you have to connect a level converter between CON30 and the RS−232 of the terminal or PC. The level converter can take its supply from the target module:




Do not connect any of the other pins of CON30!
The bitrate/frameformat is 38400/8N1 by default (configured in 'init.c').

**Configuration**

The debug system is configured with the following to constants defined in the file 'include/debug.h':

ADD_CR — Define this constant if your terminal want to see LF/CR line breaks instead of ony LF (defined by default).

DEBUGMASK — This constant must be defined to enable the debug system. The code is removed if undefined (defined to 0x1D by default). The value selects the messages you want to see:

- *0* No debug info (But code present)
- *1* POST (**P**ower **O**n **S**elf **T**est), configuration and status change.
  Note: Recommended
- *2* PIA driver (Parallel interface agent driver)
- *4* Taskrouter (SIP protocol handler)
  Note: Nexus, status and message information, recommended
- *8* LUN 0 (Taskmanager and deviceserver)
  Note: This includes information for tasks of nonexisting LUNs
- *16* LUN 0 (Logical block transfer)
- *32* PCMCIA driver (Socket services)
- *64* PCMCIA driver (Card services)
- *128* ATA controller driver

You can select multiple items by OR−ing them togeher.

Note: The debug system can significantly reduce firmware execution speed!

3. PIA driver

The PIA driver can be found in 'pia.c'. The return values for all routines are the same:

- 0 Success
- PIA_ERR_TIMEOUT Register access timeout or missing IRQ
- PIA_ERR_STATUS PIA not in expected state (=> try ABORT&RECOVER)
- PIA_ERR_FAILED PIA command failed
- PIA_ERR_PARITY PIA detected parity error
- PIA_ERR_BUFFER ATA controller buffer access error

The error constants and function prototypes are declared in 'pia.h'. The following routines are provided:

*uint8_t pia_get_status(uint8_t* status)*

Parameters: status Pointer to result

Returns the content of the PIA status register in 'status'.

*uint8_t pia_init(uint8_t id, uint8_t config)*

Parameters: id Our SCSI ID
config PIA config word

Configures the PIA and enables the SCSI interface.

*uint8_t pia_abort(void)*

Aborts currently executed PIA command.

*uint8_t pia_recover(void)*

Recover PIA after error.

*uint8_t pia_busfree(void)*

Release the SCSI bus.

*uint8_t pia_accept_selection(uint8_t* initiator)*

Parameters: initiator Pointer to SCSI ID of link partner

Accept selection and take over SCSI bus control from initiator. A successful selection establish an I_T ("**I**nitiator−**T**arget") nexus.
Must be called within 200µs after the PIA have set the 'SELECT' status bit! Otherwise a Selection Abort Timeout will occur and this command fails.
The SCSI ID of the selecting initiator is returned in 'initiator'.

Note:
Without this command the PIA stay passive and the initiator will think we are not present.

*uint8_t pia_get_message(uint8_t* buffer, uint8_t buffersize)*

Parameters: buffer — Pointer to data buffer
buffersize — Data buffer size

Receive message from initiator.
This command is only valid if an ATTENTION condition is present on the initiator (indicated by the ’ATN’ status bit of the PIA).
The message is copied to ’buffer’. If the message is larger than ’buffersize’, the message is silently truncated.

*uint8_t pia_put_message(uint8_t* buffer, uint8_t messagesize)*

Parameters: buffer — Pointer to data buffer
messagesize — Message length

Send message in ’buffer’ to initiator. ’messagesize’ bytes are written to the PIA.

*uint8_t pia_put_message_again(void)*

Sends the last message to the initiator again.

*uint8_t pia_get_command(uint8_t* buffer, uint8_t buffersize)*

Parameters: buffer — Pointer to data buffer
buffersize — Data buffer size

Receive command from initiator. The CDB is copied to ’buffer’. If the CDB is larger than ’buffersize’, it is silently truncated.

*uint8_t pia_put_status(uint8_t status)*

Parameters: status — SIP status byte

Send status of executed task to initiator.

*uint8_t pia_get_data(uint8_t* buffer, uint16_t length)*

Parameters: buffer — Pointer to data buffer
length — Number of bytes to read

Receive ’length’ bytes of data from initiator and store them in ’buffer’.

*uint8_t pia_put_data(uint8_t* buffer, uint16_t length)*

Parameters: buffer — Pointer to data buffer
length — Number of bytes to read

Read ’length’ bytes of data from ’buffer’ and send them to initiator.

*uint8_t pia_get_data_fast(uint8_t* buffer, uint16_t length)*

| Parameters: | buffer | Pointer to data buffer |
|---|---|---|
| | length | Number of bytes to read |

Receive ’length’ bytes of data from initiator and copy them to the buffer of the ATA controller. ’length’ must be a multiple of 512 and ’buffer’ must point to a memory block of 512 Byte in size.

*uint8_t pia_put_data_fast(uint8_t* buffer, uint16_t length)*

| Parameters: | buffer | Pointer to data buffer |
|---|---|---|
| | length | Number of bytes to read |

Read ’length’ bytes of data from ATA controller buffer and send them to initiator. ’length’ must be a multiple of 512 and ’buffer’ must point to a memory block of 512 Byte in size.

4. PCMCIA card services

The PCMCIA socket and card services can be found in ’pcmcia.c’.
The socket services are for internal use only and should never be called directly.

The return values for all cardservice routines are the same:

- 0 Success
- CS_ERR_FAIL Operation failed
- CS_ERR_NOCARD No card present
- CS_ERR_CIS CIS parse error
- CS_ERR_TIMING Card timing not supported (>12.5us or zero)

The error and status constants, function prototypes and the following CIS structure are declared in ’pcmcia.h’.

CIS structure used for ’cs_parse_cis()’ function:

```
struct cistpl_device {
   uint8_t  type;                  /* Card type */
                                   /* 0x0  No device
                                    * 0x1  ROM
                                    * 0x2  OTP-ROM
                                    * 0x3  EPROM
                                    * 0x4  EEPROM
                                    * 0x5  Flash-EPROM
                                    * 0x6  SRAM
                                    * 0x7  DRAM
                                    * 0xD  I/O controller
                                    * 0xE  Reserved
                                    */
   uint8_t  wps;                   /* Write protect switch exist
*/
   uint8_t  speed;                 /* Interface speed [ns] */
                                   /* (0xFF = Extended speed) */
   uint32_t  ext_speed;            /* Extended speed [ns] */
   uint32_t  size;                 /* Common memory size [Byte] */
};

struct  cistpl_vers_1 {
   uint8_t  major;                 /* Spec major number */
                                   /* (4: JEIDA 4.x, PCMCIA 2.x)
*/
   uint8_t  minor;                 /* Standard minor number
                                      (JEIDA, subtract 1 for
                                      PCMCIA standard) */
   char  vendor[17];               /* Manufacturer name */
   char  product[17];              /* Product name */
};

struct cistpl_device_oc {
   uint8_t  wait;                  /* WAIT signal is supported */
   uint8_t  power_3v;              /* 3V operation is allowed */
};
```

```
struct  cistpl_funcid {
   uint8_t  function;              /* Card function */
                                   /* 0x0  Multifunction
                                    * 0x1  Linear memory
                                    * 0x2  Serial interface
                                    * 0x3  Parallel interface
                                    * 0x4  ATA controller
                                    * 0x5  Video interface
                                    * 0x6  Network interface
                                    * 0x7  Auto increment mass
                                           storage
                                    * 0x8  SCSI controller
                                    * 0x9  USB controller
                                    */
   uint8_t  install;               /* Install card at POST */
};

typedef struct {
   struct cistpl_device  tpl_device;  /* CISTPL_DEVICE */
   struct cistpl_funcid  tpl_funcid;  /* CISTPL_FUNCID */
   struct cistpl_device_oc  tpl_device_oc;  /* CISTPL_DEVICE_OC */
   struct cistpl_vers_1  tpl_vers_1;  /* CISTPL_VERS_1 */
} CIS;
```

The following routines are provided:

*uint8_t cs_init(void)*

Initialize card services. Must be called before any of the other routines can be used.

*uint8_t cs_status(uint8_t* status)*

Parameters: status Pointer to requested value

Returns current status of cardservice. The following three status bits are valid:

| | |
|---|---|
| CS_CP | PCMCIA card present if set to one (all other bits are invalid if zero) |
| CS_RDY | PCMCIA card ready for access if set to one |
| CS_WP | PCMCIA card write protected if set to one (This bit indicates the state of the WP signal of the PCMCIA card, you have to check the CIS to find out if this signal is valid for the current card). This bit is invalid if ’CS_RDY’ is zero. |

Use ’cs_read_wp()’ function for convenience if you want to check for write protection (this will check the CIS automatically).

*uint8_t cs_enable_card(void)*

Switch on supply of the socket and reset PCMCIA card. If this call was successful, the ’CS_RDY’ status bit should be set to one (the ’CS_WP’ bit is valid in this case).

*uint8_t cs_disable_card(void)*

Switch off supply of the socket.

*uint8_t cs_read_attribute(uint32_t addr, uint8_t* buffer, uint8_t count)*

| Parameters: | addr | Start address |
|---|---|---|
| | buffer | Pointer to buffer |
| | count | Number of bytes |

Read ’count’ bytes starting at address ’addr’ from attribute memory of the card and store them in ’buffer’.

*uint8_t cs_parse_cis(CIS* cis)*

| Parameters: | cis | Pointer to CIS structure |
|---|---|---|

This routine returns a valid CIS stucture for the current card. Note that this function return default values if the corresponding tuples are not found in the CIS. In other words, an error is returned only if the CIS is invalid − no error is returned for an empty CIS (default values are set for an 1MiByte linear memory card in this case).

*uint8_t cs_read_wps(uint8_t* wp)*

| Parameters: | wp | Pointer to requested value |
|---|---|---|

This routine returns wp = 0 if the card is writeable or have no WP switch and wp = 1 if the card is write protected.

*uint8_t cs_read_common(uint32_t addr, uint8_t* buffer, uint16_t count)*

| Parameters: | addr | Start address |
|---|---|---|
| | buffer | Pointer to buffer |
| | count | Number of bytes |

Read ’count’ bytes starting at address ’addr’ from common memory of the card and store them in ’buffer’.

*uint8_t cs_write_common(uint32_t addr, uint8_t* buffer, uint16_t count)*

| Parameters: | addr | Start address |
|---|---|---|
| | buffer | Pointer to buffer |
| | count | Number of bytes |

Write ’count’ bytes from buffer to common memory of the card starting at address ’addr’.

*uint8_t cs_read_common_fast(uint8_t addr, uint8_t* buffer, uint16_t count)*

| Parameters: | | |
|---|---|---|
| | addr | Start address (must be even!) |
| | buffer | Pointer to buffer |
| | count | Number of bytes (must be even!) |

Read ’count’ bytes from address ’addr’ in common memory of the card and store them in ’buffer’. All data is read from the same address! This is intended to get data from an ATA controller FIFO.

*uint8_t cs_write_common_fast(uint8_t addr, uint8_t* buffer, uint16_t count)*

| Parameters: | | |
|---|---|---|
| | addr | Start address (must be even!) |
| | buffer | Pointer to buffer |
| | count | Number of bytes (must be even!) |

Write ’count’ bytes from ’buffer’ to address ’addr’ in common memory of the card. All data is written to the same address! This is intended to store data in an ATA controller FIFO.

5. ATA driver

This driver can be found in 'ata.c'. It was designed to work with ATA5 compliant controllers. The return values for all routines are the same:

- 0 Success
- ATA_ERR_FAIL Operation failed
- ATA_ERR_TIMEOUT Operation timed out

This constants are declared in 'ata.h'. The timeout can also be adjusted in 'ata.h' with the following constants (Values are in miliseconds):

| | |
|---|---|
| ATA_TIMEOUT | Timeout for ATA controller to become ready while idle |
| ATA_CMD_TIMEOUT | Timeout for command to complete |

The following routines are provided:

*uint8_t ata_init(void)*

Reset ATA controller. Must be called before any of the other routines can be used.

*uint8_t ata_get_status(uint8_t* status)*

Parameters: status Pointer to requested value

Returns current status of ATA controller. The following three status bits are valid:

| | |
|---|---|
| ATA_ERR | Last command completed with error |
| ATA_CORR | Data was corrected in last command |
| ATA_DRQ | Data request |
| ATA_DWF | Write fault occured |
| ATA_RDY | ATA controller ready |
| ATA_BUSY | ATA controller busy (all other bits are invalid if set to one) |

Look at the ATA5 standard for details about the bits in the status register.

*uint8_t ata_execute_diagnostic(void)*

Execute a EXECUTE DIAGNOSTIC command.

*uint8_t ata_flush_cache(void)*

Execute a FLUSH CACHE command.

*uint8_t ata_identify_drive(uint8_t* buffer)*

Parameters: buffer Pointer to data buffer

Execute an IDENTIFY DRIVE command and return the data in 'buffer'. 'buffer' must have enough space for one block (512 bytes).

*uint8_t ata_read_blocks(uint32_t lba, uint8_t count)*

| Parameters: | lba | Address of first block |
|---|---|---|
| | count | Number of blocks |

Send a READ SECTORS command to read 'count' blocks starting at address 'lba'. Note: The data is not transferred by this function!

*uint8_t ata_write_blocks(uint32_t lba, uint8_t count)*

| Parameters: | lba | Address of first block |
|---|---|---|
| | count | Number of blocks |

Send a WRITE SECTORS command to write 'count' blocks starting at address 'lba'. Note: The data is not transferred by this function!

*inline uint8_t ata_read_buffer(uint8_t* buffer)*

| Parameters: | buffer | Pointer to data buffer |
|---|---|---|

Copy one block from ATA controller to local memory buffer 'buffer'.

*inline uint8_t ata_write_buffer(uint8_t* buffer)*

| Parameters: | buffer | Pointer to data buffer |
|---|---|---|

Copy one block from local memory buffer 'buffer' to ATA controller.

*uint8_t ata_complete_cmd(uint8_t* corr)*

| Parameters: | corr | Corrected data flag |
|---|---|---|

Complete ATA commmand, check for success and check for recovered errors. If the ATA controller has corrected data for the current command, 'corr' is set to an unzero value.

Example: To read some some data the following sequence of functions must be called:

| | |
|---|---|
| 1) ata_read_blocks() | Send command |
| 2) ata_read_buffer() | Transfer data |
| 3) ata_complete_cmd() | Complete command and check for status |

6. SIP protocol handling

The SIP protocol is implemented in 'taskrouter.c'. The taskrouter accepts a selection and handles the messages from the initator (establish the I_T_L nexus and reject potential SYNCHRONOUS DATA TRANSFER REQUEST and WIDE DATA TRANSFER REQUEST messages). Then it receives the CDB and creates a task for the corresponding LUN.

The taskrouter is called as 'sip_taskrouter()' by 'main()' after POST and runs in an infinite loop. Because we use no multitasking OS, the taskmanagers and deviceservers of the LUNs are called by the taskrouter when they have to do something.

When a deviceserver have completed a task, the taskrouter sends the status byte and a TASK COMPLETE message to the initator and releases the SCSI bus (this destroys the nexus).

**The taskrouter supports the following messages:**

Link control messages:

| | |
|---|---|
| ● TASK COMPLETE | Prepares regular bus release |
| ● RESTORE POINTERS | Reset initiators state machine for current task |
| ● INITIATOR DETECTED ERROR | Message from initiator to indicate corrupted state machine for current task |
| ● MESSAGE REJECT | Message unknown or not supported |
| ● NO OPERATION | Message from initiator to indicate that the reason for the current ATTENTION condition is no longer present |
| ● MESSAGE PARITY ERROR | Message from initator to indicate parity error |
| ● IDENTIFY | Establish I_T_L ("**I**nitiator−**T**arget−**L**ogical unit") nexus |

Task management messages:

| | |
|---|---|
| ● ABORT TASK SET | Abort all tasks for current LUN and release the SCSI bus |
| ● TARGET RESET | Executes a soft reset (media data is preserved) |

**The following status codes are used:**

| | |
|---|---|
| ● GOOD | Task completed successfully |
| ● CHECK CONDITION | Error (sense data available) |
| ● BUSY | Cannot create task (other task pending) [Not relevant without disconnect/reconnect] |
| ● RESERVATION CONFLICT | LUN reserved for other initiator (wait until other initiator have released the LUN) |

7. Taskmanager

The taskmanager is implemented in 'lun0.c' and consists of two simple routines that can be called by the taskrouter:

*uint8_t lun0_tm_check_reservation(uint8_t ini_id, uint8_t* cdb)*

| | | |
|---|---|---|
| Parameters: | ini_id | Initiators SCSI ID |
| | cdb | Pointer to CDB of new task |

Check for a pending reservation of another initiator.
Note: The commands INQUIRY and REPORT LUNS are always accepted.

Returns zero on success and one otherwise (in this case the taskrouter should return status RESERVATION CONFLICT).

*uint8_t lun0_tm_create(uint8_t ini_id, uint8_t* cdb)*

| | | |
|---|---|---|
| Parameters: | ini_id | Initiators SCSI ID |
| | cdb | Pointer to CDB of new task |

Creates a new task. This command fails if the task set is full.

Returns zero on success and one otherwise (in this case the taskrouter should return status BUSY).

8. Deviceserver

The deviceserver is implemented in ’lun0.c’. It executes the tasks created by the taskmanager. The deviceserver have no queue and there can be only one task in the taskset at any time for any number of initiators. The deviceserver can be idle (task in ENDED state) or busy (task in CURRENT state).

After a task is created, the taskrouter calls the deviceserver using the following function:

*uint8_t lun0_ds_execute(void)*

Execute current task

If the deviceserver returns zero the task was executed successfully. If one is returned, an error occured and sense data have been prepared for the initiator (the taskrouter should return CHECK CONDITION status in this case).

**UNIT ATTENTION condition:**

The UNIT ATTENTION condition is used to inform the initiators about a state change on a LUN (transfer mode agreements and nexus my be lost after a reset event, cache may be invalid after a medium change event, etc.). If the LUN is in UNIT ATTENTION condition any command (except INQUIRY) is aborted and CHECK CONDITION status is returned. The initiator can read the reason for the UNIT ATTENTION condition with the REQUEST SENSE command. The next command always clear the UNIT ATTENTION condition.

LUN 0 of this project activates UNIT ATTENTION condition for the following cases:

- Reset (hard and soft reset)
- Medium change

The sense key is set to UNIT ATTENTION and the sense code is set to "Power on or target reset" (0x29 / 0x00) or "Not ready to ready change, medium may have changed" (0x28 / 0x00) respectively.

Note:
Every UNIT ATTENTION condition is managed separately for every initiator (stay pending for all other initiators after initiator x has read the sense data). LUN 0 of this project supports a queue for every initiator so that multiple UNIT ATTENTION conditions can be pending.

**Power conditions:**

If a supported medium is present, LUN0 can be in one of three power conditions:

- Idle (LUN ready)
  This is the default state after "power on" or "medium change" event. The first access creates a small delay and the LUN automatically switch to active state.
- Active (LUN ready, yellow LED switched on)
  All commands are excecuted with no delay.
- Standby (LUN not ready)
  This state can only be entered by a START STOP UNIT command with ’START’ bit set to zero. All commands who needs medium access will be aborted and CHECK CONDITION status is returned. The sense key is set to NOT READY and the sense code is set to "Not ready, initializing command required" (0x04/0x02).

**Missing or unsupported medium:**
If the medium is missing, the deviceserver rejects all commands who needs medium access with CHECK CONDITION status. The sense key is set to NOT READY and the sense code is set to "Medium not present" (0x3A/0x00).
If the deviceserver detects an unsupported medium, the red LED is switched on and all commands behave like there is no medium present.

**Write protection:**
The deviceserver supports write protection. If LUN 0 is a linear memory card, the hardware write protect switch of the PCMCIA card is used if available. If LUN 0 is an I/O card (ATA controller), the ATA controller on the card must handle write protection.

**The deviceserver supports the following commands:**

- TEST UNIT READY
  Check for LUN to be ready
  If no media is present the command will fail, the sense key is set to NOT READY and the sense code is set to "Medium not present".
- REQUEST SENSE
  Returns the standard sense data (18Byte). Only valid after a CHECK CONDITION status. This command will never fail (even on nonexisting LUNs).
- FORMAT UNIT
  This command is implemented as "no operation".
  If no media is present the command will fail, the sense key is set to NOT READY and the sense code is set to "Medium not present".
  If a parameter list is present, the command will fail, the sense key is set to ILLEGAL REQUEST and the sense code is set to "Invalid field in CDB".
- MODE SENSE(6)
  All values for the ’PC’ (page control) field are valid.
  If current values for the not implemented mode page 0x00 are requested, a parameter list header and the block descriptor (but no data) are returned as specified by the SPC document.
  If the ’DBD’ Bit is set to zero, the command returns a block descriptor. If the ’DBD’ Bit is set to one, no block descriptor is returned.
- START STOP UNIT
  If the ’POWER CONDITION’ field is not zero, this command is executed as "no operation" (power management is not implemented).
  Otherwise the logical unit is put to standby state if the ’START’ bit is cleared or to active state if the ’START’ bit is set. The ’LOEJ’ Bit is always ignored.

- READ(6)
  Reads a logical block (**LB**) with size of constant ’BLOCKSIZE’ from media.
  If no media is present the command will fail, the sense key is set to NOT READY and the sense code is set to "Medium not present".
  If the requested LBA (**L**ogical **B**lock **A**ddress) is out of range, the command will fail, the sense key is set to ILLEGAL REQUEST and the sense code is set to "LBA out of range".
  If the LBs are read using error correction, the command will terminate with status CHECK CONDITION. The sense key is set to RECOVERED ERROR and the sense code is set to "ECC and retries applied".
  If the LBs cannot be read from the media, the command will fail, the sense key is set to MEDIUM ERROR and the sense code is set to "Unrecovered read error".
  If the LB cannot be transferred to the initiator, the command will fail, the sense key is set to HARDWARE ERROR and the sense code is set to "Data phase error".
- WRITE(6)
  Writes a LB with size of constant ’BLOCKSIZE’ to the media.
  If no media is present the command will fail, the sense key is set to NOT READY and the sense code is set to "Medium not present".
  If the LBs are written using auto−reallocation, the command will terminate with status CHECK CONDITION. The sense key is set to RECOVERED ERROR and the sense code is set to "Block reallocated".
  If the requested LBA is out of range, the command will fail, the sense key is set to ILLEGAL REQUEST and the sense code is set to "LBA out of range".
  If the LB cannot be read from the initiator, the command will fail, the sense key is set to HARDWARE ERROR and the sense code is set to "Data phase error".
  If the LB cannot be written to the media, the command will fail and the sense key is set to MEDIUM ERROR and the sense code is set to "Write error".
- INQUIRY
  Returns the standard inquiry data (36Byte). This command will never fail (even on nonexisting LUNs or if a reservation is pending).
  The inquiry data is defined in ’inquiry.h’.
- REPORT LUNS
  Returns a list of supported LUNs (contains only LUN 0 for this project).
- RESERVE(6)
  Reserve LUN for initiator.
  Extends are not supported and cannot be reserved, the command fails in this case, the sense key is set to ILLEGAL REQUEST and the sense code is set to "Invalid field in CDB".
  3rd party reservations (reservations for other initiators) are supported.
- RELEASE(6)
  Release reservation of LUN.
  A 3rd party reservation can only be released by the initiator who installed it. Otherwise the command fails, the sense key is set to ILLEAGL REQUEST and the sense code is set to "Invalid field in CDB".
- SEND DIAGNOSTIC
  Only the standad selftest is supported (’ST’ bit set to one). Otherwise the command will fail, the sense key is set to ILLEGAL REQUEST and the sense code is set to "Invalid field in CDB".
  The command is implemented as "no operation".

- READ CAPACITY(10)
  The last valid LBA and the blocksize are returned.
  The blocksize is taken from the constant ’BLOCKSIZE’.
  If no media is present the command will fail, the sense key is set to NOT READY and the sense code is set to "Medium not present".
- READ(10)
  Same behaviour as READ(6).
- WRITE(10)
  Same behaviour as WRITE(6).
- SYNCHRONIZE CACHE(10)
  If the medium is a "Linear memory device", this command is implemented as "no operation" because the SCSI controller have no local cache. If the medium is an "I/O device" (ATA controller), the deviceserver sends a FLUSH CACHE command to the ATA controller and reports the exit status of that command.
- RESERVE(10)
  Same behaviour as RESERVE(6).
- RELEASE(10)
  Same behaviour as RELEASE(6).
- REPORT LUNS
  This command returns a list indicating that only LUN 0 is present.

**Mode pages:**
LUN 0 supports the following mode pages:
- Read/write error recovery page (0x01)
  Parameter changing and saving to nonvolatile memory is not supported.

**Logical block access:**
The deviceserver accesses the media using the following abstract API (easily portable to any type of media):

*uint8_t lb_read(uint16_t lba, uint8_t* buffer, uint16_t count)*

| | | |
|---|---|---|
| Parameters: | lba | Logical block address |
| | buffer[BLOCKSIZE] | Data buffer |
| | count | Number of blocks to read |

Reads ’count’ logical blocks starting from ’LBA’ and copy them to ’buffer’.

*uint8_t lb_write(uint16_t lba, uint8_t* buffer, uint16_t count)*

| | | |
|---|---|---|
| Parameters: | lba | Logical block address |
| | buffer[BLOCKSIZE] | Data buffer |
| | count | Number of blocks to write |

Reads enough data to fill ’count’ logical blocks from ’buffer’ and copy it to logical blocks starting from ’LBA’.

Behaviour for linear memory cards:
Both routines use the constant ’BLOCKSIZE’ for the size of logical blocks.
Note: Changing ’BLOCKSIZE’ may be dangerous! Values that are not a power of two may cause various problems and values larger than 512 may cause stack overflows.

Behaviour for I/O cards with ATA controller:
’BLOCKSIZE’ must always be 512 or any I/O card is rejected before the block access routines are called.

EOF