Documentation for SCSI controller project
Target module
Hardware V1.0.1 / Firmware V0.0

Final 2005−07−17 / 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 **U**nit **N**umber) 0
- Deviceserver for LUN 0
- Logical block access routines
- DRAM (LUN 0 media) controller

Hardware Features
Hardware V1.x is based on an Atmel ATmega64−16 AVR Microcontroller. The SCSI ID can be configured via the jumper block JP30.
Closing jumper JP31 forces a hard reset on all modules (but not on the SCSI bus). An external SCSI bus reset also forces a hard reset on all modules immediately.
Standard 30Pin asynchronous DRAM memory modules with 1MiByte or 4MiByte capacity are accepted as media.

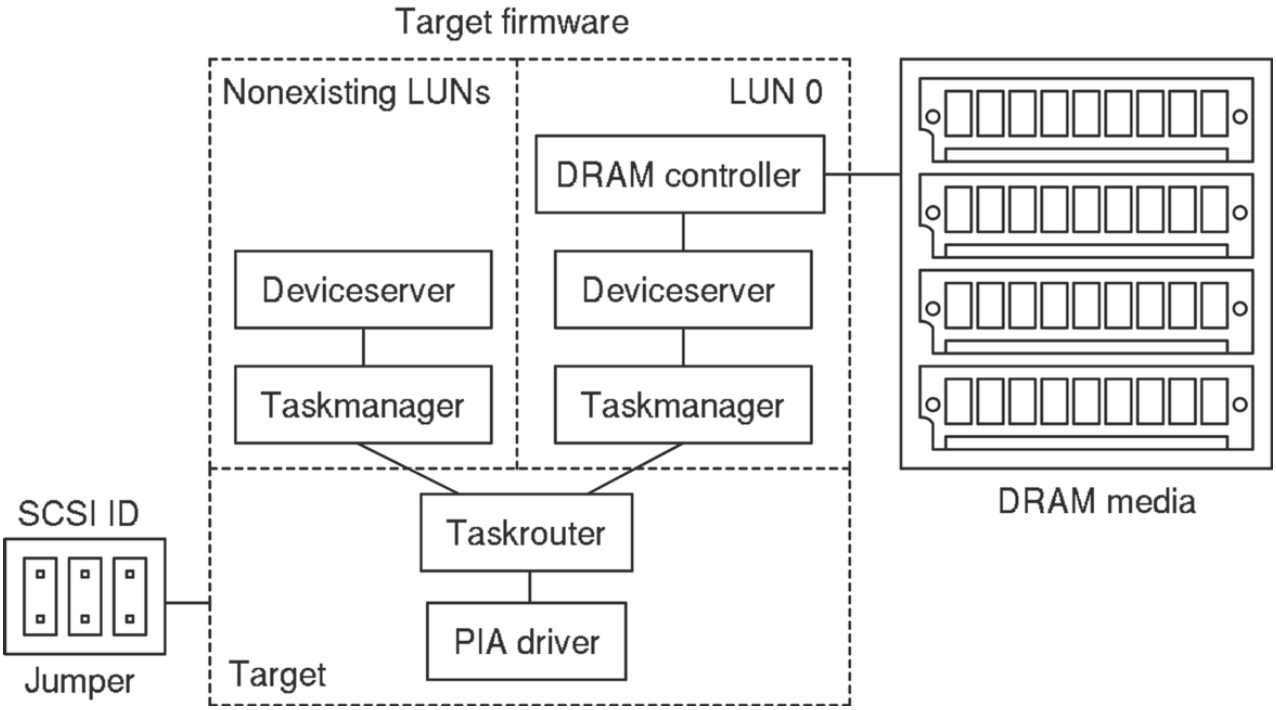
Firmware Features
The Target firmware correctly handles commands to nonexisting LUNs and implements all commands that are mandatory for Type 0 ("direct access") devices on LUN 0. Some optional commands are also supported. Disconnect/reconnect is currently not implemented (makes no sense for a RAM disk). Linked commands and TCQ are not supported (requires more ressources than the ATmega64 can provide).

To do:
- Implement optional commands REPORT LUNS, READ(12), WRITE(12), READ(16), WRITE(16) and READ CAPACITY(16) that are useful or have become mandatory for Type 0 devices in newer versions of the SCSI standard
- Implement CBR and Burst refresh for DRAM controller (reduces CPU load)
- Enhance page mode routines for DRAM controller (together with burst refresh)

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

0. Block diagram

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. Changing the ID jumpers have no effect until the next reset.

DRAM memory:

It is expected that there are always four 1MiByte or 4MiByte memory modules of the same type be present in the system! Target firmware V0.0 neither write nor check parity so 8Bit and 9Bit modules can be used.

1. 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

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 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, uint8_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, uint8_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.

2. 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 taskmangers 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)

3. 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 command INQUIRY is 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).

4. 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 aggreements 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.

Note: The UNIT ATTENTION condition is managed separately for every initiator (stay pending for all other initiators after initiator x has read the sense data).

The logical unit in this project activates UNIT ATTENTION condition only after reset (hard and soft reset). The sense key is set to UNIT ATTENTION and the sense code is set to "Power on or target reset".

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
  We use no media format, therefore 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".
- READ(6)
  Reads a logical block (**LB**) with size ’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 LB 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 ’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 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’.
- 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".
  Currently the command is implemented as "no operation" because we cannot run any useful test without losing media data.
- 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).
- RESERVE(10)
  Same behaviour as RESERVE(6).
- RELEASE(10)
  Same behaviour as RELEASE(6).

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)*

Parameters:	lba	Logical block address
	buffer	Data buffer

Reads logical block number ’LBA’ and copy it to ’buffer’.

uint8_t lb_write(uint16_t lba, uint8_t buffer)*

Parameters:	lba	Logical block address
	buffer	Data buffer

Reads data from ’buffer’ and copy it to logical lock number ’LBA’.

Both routines use the constant ’BLOCKSIZE’ for the size of logical blocks.
Note: Changing ’BLOCKSIZE’ may be dangerous − values below 256 may be unsupported due to DRAM access design, values that are not a power of two may cause various problems (untested) and values larger than 512 may cause stack overflows because the block buffer in SRAM is allocated with ’BLOCKSIZE’ bytes).

The logical blocks are linearly mapped to the DRAM address space. There is no defect management and no defect tables exist.

5. DRAM controller

DRAM access is implemented in ’dram.c’, DRAM refresh is implemented in ’interrupt.c’.
First of all: An AVR is not a good DRAM controller! The timings for the RAM chips that are required for good performance are by far too fast and complicated to generate for an AVR (even at 20MHz).
=> The result is significant CPU load for the refresh and really slow performance (I have measured from 57KiByte with GCC code up to 182KiByte using fast page mode and address splitting routines written in assembler).

A routine is provided to detect the capacity of the first memory module (the one in socket CON40):

uint8_t dram_get_size(void)
Returns the size of the first memory module in MiBytes multiplied by four

Refresh:

This is the heart of every DRAM controller. DRAM stores the data in capacitors (approx. 10fF) which are discharged by leakage currents. Reading the data also discharges the capacitors. Therefore all DRAMs contain special circuits that writes the data back to the capacitors immediately after reading. This circuit is also used for repeated data retention refreshing (a refresh cycle internally simply reads the data but do not activate the output drivers). Because the capacitors are organized as matrix, reading one byte always refreshes the complete row this byte is located in. Therefore it is sufficiant to refresh every row once whithin the maximum data retention interval.

Some of the newer DRAM chips can internally execute parts of the refresh procedure like generating the row address. Because I did not know if this is supported by all modules, I have implemented the old "RAS only" refresh that works for all asynchronous DRAMs. A "RAS only" refresh cycle is a half read cycle (aborted after the row address was transmitted).

1MiByte memory modules normally have 512rows that must be refreshed every 8ms, 4MiByte modules normally have 2048rows that must be refreshed every 32ms.
The refresh is implemented using Timer2. The Timer2 compare match interrupt handler is executed every 12.5us and refreshes the next row resulting in a refresh cycle of 6.4ms for 512rows (security margin for weak cells).

Because GCC have produced "suboptimal" code, I have written the refresh in assembler. This has halved the CPU load for the refresh to approx. 20%.

Access:

For a normal access the row and column addresses must be transmitted to the memory module for every byte. Some DRAM chips support page mode. In page mode the row address must only be tranmitted once and multiple bytes within that row can be read be transmitting new column addresses without releasing the /RAS signal. Page mode is used if the constant ’PAGE_MODE’ is defined in ’dram.c’ The constant ’PP_SIZE’ define the number of bytes to transfer in page mode.
To prevent the DRAM accesses from being clobbered by a refresh interrupt all interrupts are disabled during the access. Therefore higher values for ’PP_SIZE’ increase interrupt latency and may timeout the refresh!

The DRAM access API is implemented as follows. The return values for both routines are the same:

- 0 Success
- DRAM_PAGE_ERR Invalid parameter ’addr’ or ’len’
- DRAM_PARITY_ERR Parity error

uint8_t dram_read(uint32_t addr, uint8_t buffer, uint16_t len)*

Parameters:	addr	Address (linear address space) Must be a multiple of 256 or zero
	buffer	Pointer to data buffer
	len	Number of bytes to read Must be a multiple of 256

Copy ’len’ bytes from DRAM address ’addr’ to ’buffer’.

uint8_t dram_write(uint32_t addr, uint8_t buffer, uint16_t len)*

Parameters:	addr	Address (linear address space) Must be a multiple of 256 or zero
	buffer	Pointer to data buffer
	len	Number of bytes to read Must be a multiple of 256

Copy ’len’ bytes from ’buffer’ to DRAM address ’addr’.

The memory is linearly mapped to the memory modules in the order of the socket numbers (CON40 to CON43). The column address is mapped to the least significant bits of the linear address (necessary for page mode). Splitting the linear 32Bit address to module−, row− and column address is (really) slow and can be accelerated using assembler routines by defining the constant ’FAST_ADDR_SPLIT’ in ’dram.c’.

EOF