M U L T I C L U S T E R Series

Hardware Documetation
&
Software Documentation

M S C

Author:

Winfried Mularski

Mass Storage Controler

Busless Transputer Module
with SCSI and floppy interface

Technical Documentation

Version 1.3 , May 1989

PART I Hardware

1. Block Diagram and General Description

1.1 Introduction

The MSC board is part of the MULTICLUSTER and SUPERCLUSTER series. The MSC board serves as a mass storage controller, which interfaces via SCSI bus and floppy bus to SCSI and loppy disk drive devices. It can operate as a host or as a fileserver/mass storage subsystem in a transputer network.

Multiple MSC boards can be used to drastically increase the I/O bandwidth of a transputer network. This is achieved by connecting to every MSC it's own mass storage devices (mainly winchesters). A MSC can do SCSI bus transfers in parallel with link data transfers over all 4 links without significant performance degradation.

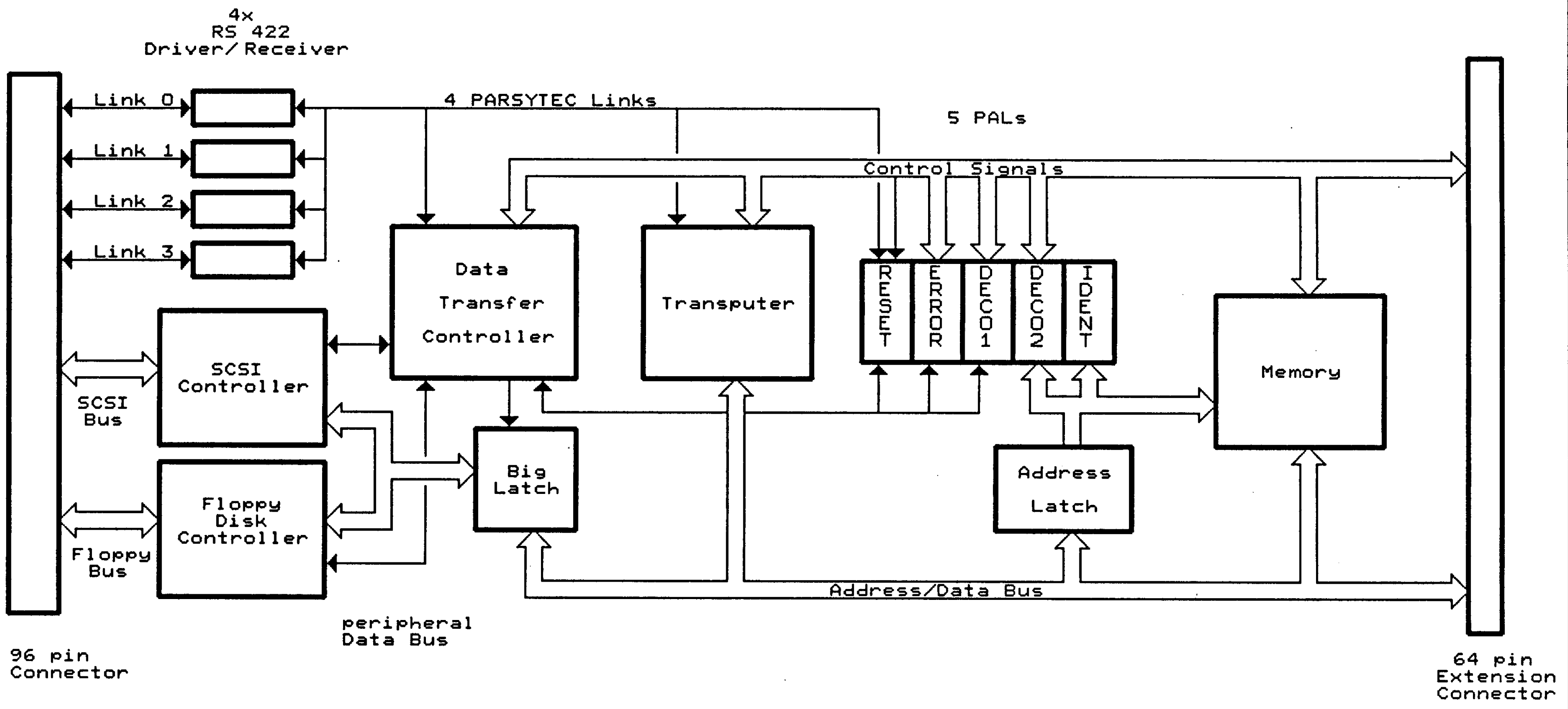
The big memory size of 4 MByte enables implementation of sophisticated buffer algorithms.

The MEGAFRAME modular concept also supports fail save systems: For example two or more MSC's, each with its own winchester of the same type, perform in parallel read and write operations with the same data (n-fold redundancy).

Furtheron in case of a program crash the MSC board can be reset by sending to it a link reset signal over any of the 4 MEGAFRAME links.

Features:

- 32 bit Transputer T800 (opt. T414)
- 4 MByte memory with parity checking (16 MByte, when 4 MBit DRAMs are avaiable)
- 4 RS-422 driven MULTICLUSTER links
- directly interfaces to SCSI bus (ANSI SCSI X3T9.2)
- directly interfaces to floppy bus
- average asynchronous SCSI transfer rate: 1.1 MByte/s
- average synchronous SCSI transfer rate: 1.8 MByte/s
- bidirectional data transfer over all 4 links in parallel with SCSI bus data transfer
- additional onboard 64 pin connector with transputer interface for extension boards
- software package available with complete set of medium level communication procedures (clear, load, unload, read, write etc.); dynamical multisector blocks
- 5 Volt only, low power
- small board size: extended euro card



PARSYTEC GmbH		
Size	Document Number	REV
A	MSC schematic	1.1
Date: February 19, 1988	Sheet of	

1.2 Processor

The MSC board runs with a T800 or T414 Transputer. The T800 is a 32 bit processor with 4 K bytes of on-chip static RAM. It runs at 20 MHz to perform an instruction throughput of 10 MIPS. Four high speed serial links (10 or 20 Mbit/sec) support the communication with other transputers in a network.

Furthermore the T800 has a floating point unit on chip to perform 1.5 MFLOPS/sec.

1.3 Clock

All transputers derive their processor clock from an internal oscillator, which is synchronized by an internal PLL to an external 5 MHz oscillator. The transputer clock speed is defined by jumpers. See Jumper Allocation.

1.4 Links

The four bidirectional serial links of the transputer operate independently of the processing element when transfering data from or to memory by using fast DMA. So the use of the links does only lightly degrade processor performance. The links have a default speed of 10 Mbit/sec and can also operate at 5 and 20 Mbit/sec. The link speeds can be selected by jumpers.

The MSC board has four MULTICLUSTER links. A MULTICLUSTER link consists of four signals: Two signals transfer the serial link data, one for both directions, and two reset signals, also one for both directions. Using these reset signals the transputer can generate a reset for any of its four neighbour transputers. Further on such a link reset does not effect a normal transputer reset but a so called Analyse/Reset. The Analyse/Reset preserves the internal status of the resetted transputer section. This internal status can be analysed by a user written procedure, which must be downloaded into the transputer (see "Error and Analyse").

Link and link reset signals, which leave the board, are driven by RS422 drivers. Link and link reset signals which enter the board are conditioned by RS-422 receivers. This provides a considerably higher noise immunity and longer distances for data transmission (up to 10 meters at 20 MBits/sec, up to 30 meters at 10 or 5 MBits/sec)).

1.5 Memory

The memory is organized as 1M x 32 bit, i.e. 4 MByte, and consists of dynamic RAM's. The 4 bytes in the 32 bit word are parity checked by additional 1M x 4 bit memory. The MSC board can be populated with 16 MByte memory when the next generation dynamic RAM's (4 Mbit DRAMs) are available.

1.6 SCSI/Floppy Controller

The WD33C93 SCSI Controller provides the interface to the SCSI Bus. To speed up SCSI data transfer a multiplexer-buffer-latch, called **BigLatch** (see "The BigLatch"), connects the 8 bit controller data bus to the 32 bit transputer data bus.

The WD37C65 Floppy Controller interfaces to the floppy bus. Both controllers and the BigLatch are under control of the Data Transfer Controller DTC.

1.7 Booting

After a transputer reset there are two ways to boot the transputer. They are selected by a jumper. The first is that the transputer executes the boot code of an EPROM, which resides on an optional extension board. In this case physical memory location #7FFFFFFE is interpreted as an instruction and executed. The second way is to load the boot code over a transputer link: The first incoming data from one of the links will be interpreted as boot code.

2. The SCSI Section

2.1 The SCSI Controller

The MSC board contains the WD33C93 SCSI-bus interface controller from Western Digital Corp.. The internal registers are accessible by the transputer (see "Overview of the OCCAM Adress Space").

The controller works in non multiplexed bus mode. That means, to access an internal register of the controler the transputer must first write the address of the desired register into the controlers address register and then accesses the register.

There are three ways for the transputer to notice an interrupt of the SCSI Controller.

- The interrupt bit of the Auxiliary Status Register of the SCSI Controller can be polled.
- The interrupt line can be polled (see "BigLatch Status Register")
- The interrupt line can activate the EVENTREQ input of the transputer (see "Events" and "Jumper J9").

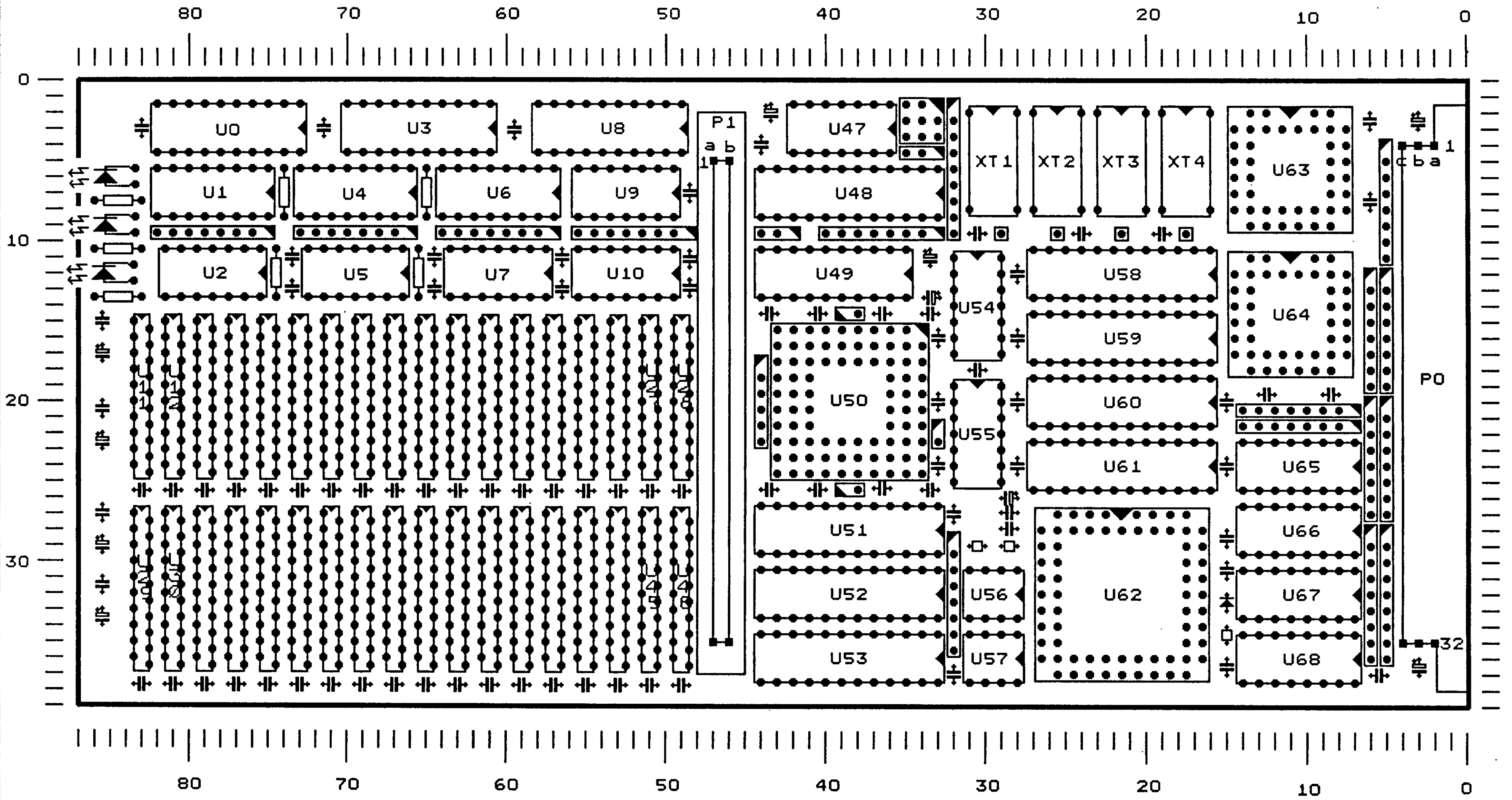
The reset input is activated at power on, external reset (see "Reset Signals") or by clearing the flip flop PRES (see "Flip flop PRES").

The DBA mode (Direct Buffer Access) of the WD33C93 is implemented, because it delivers the maximum SCSI performance. In this mode the controller actively performs read/write cycles during SCSI data phases.

2.2 The SCSI Interface

The WD33C93 SCSI controller is fully compatible with ANSI SCSI X3T9.2 specifications. The MSC board achieves an average data transfer rate of 1.1 MByte/sec (asynchronous) and 1.8 MByte/sec (synchronous). The chip includes 48 mA drivers for direct connection to the single ended SCSI bus, therefore the SCSI bus can be up to 6 meters in length.

All SCSI signals are terminated with 220/330 ohm on board. The SCSI RST signal is connected to the 96 pin bus connector and cannot be activated by the transputer.



	PARSYTEC GmbH Juelicher Str. 338 D - 5100 Aachen	
Size	Document Number	REV
A	MSC Board Layout	1.1
Date: February 10, 1988	Sheet 7 of 8	

2.3 The speed of the SCSI Interface

The MSC achieves a peek rate of 3.0 - 3.5 MBytes/sec in asynchronous mode. This high data rate can not be sustained by most winchesters, so the limiting factor in SCSI bandwidth is nearly always the winchester.

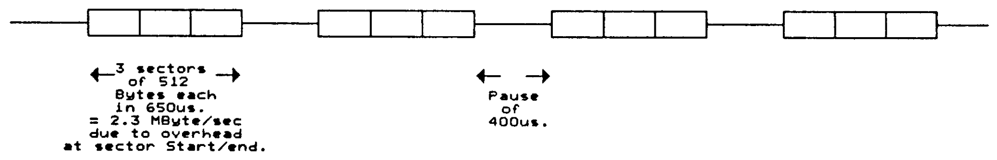
See sheet "MSC SCSI & Link transfer rate". Due to overhead of the SCSI protocol and internal operation in the winchester, the continuous data rate will be lower than 3.0 MBytes/Sec.

MSC SCSI and Link Transfer Rate

(All values measured with stretched WREN IV)

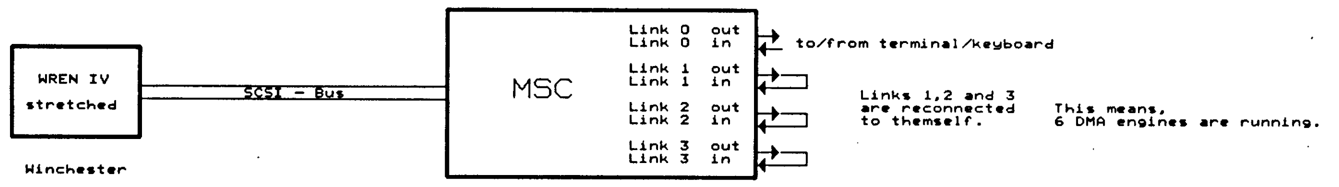
- SCSI transfer, process switching and all 4 links (8 DMA engines) can operate in parallel.
- SCSI peak transfer rate is 3.5 MByte/sec during a sector transfer. (independant of number of running links)
- Average transfer rate during long transfers is more than 1 MByte /sec.

SCSI Bus activity (measured with stretched WREN IV)



3 sectors of 512 Bytes each in 650us. = 2.3 MByte/sec due to overhead at sector Start/end.
Pause of 400us.

- Test environment:



- Link transfer rate during SCSI sector transfer:
  (During SCSI pause the links can reach their maximum speed.)

T8:	1.1 MByte/sec on any link, even if all 6 DMA engines are running.
T4:	570 KByte/sec on any link, even if all 6 DMA engines are running.

PARSYTEC GmbH		
Title: MSC SCSI & Link Transfer Rate		
Size B	Document Number MSCRATE.ORC	REV 1.1
Date: October 21, 1988	Sheet of	

3. The Floppy Section

3.1 The Floppy Controller

The MSC board contains the WD37C65 Floppy Disk Subsystem Controller from Western Digital Corporation. The features are: IBM PC/AT compatible format, dual speed spindle drive support, direct floppy disk drive interface, drives up to 4 floppy or micro floppy disk drives, data rates of 125, 250, 300, and 500 kbit/sec. Input XT1 (26) is driven by a 16 MHz clock.

The internal registers are accessible by the transputer (see "Overview of the OCCAM Address Space").

At the end of a multisector data transfer the Terminal Count input of the floppy controller must be activated to signal the transfer of the last data byte to the controller. See "Flip flop Read" in "Overview of the OCCAM address space".

Floppy data transfers are intended to be done without DMA but by polling. So this flip flop combination was used for this special purpose.

There are three ways for the transputer to notice an interrupt of the floppy controler.

- Reading the Status Register of the floppy controller.
- The interrupt line can be polled (see "BigLatch Status Register")
- The interrupt line can activate the EVENTREQ input of the transputer (see "Events" and "Jumper J9").

The reset input is activated at power on, external reset (see "Reset Signals") or by clearing the flip flop PRES (see "Flip flop PRES").

3.2 The Floppy Interface

Outgoing signals are open collector. Incoming signals have pull up resistors of 1 k ohm. See "Pin out of 34-way Floppy Connector".

The msc.driver software package uses the WD37C65 in IBM-AT mode. This means, Motor Select 1 and Drive Select 1 are both active during accesses to Floppy Drive 1. Floppy Drive 2 is accessed, when Motor Select 2 and Drive Select 2 are both active. The floppy drive 1 must be connected at the end of the 34-wire floppy cable. The wires 10 - 16 comprise a subcable, which must be twistet half a turn before connecting to drive 1. See wiring diagramm of "MSC Backplane".

4. Transputer to Controllers Interface

4.1 The BigLatch

The BigLatch is a multifunctional symmetrical bidirectional driver/latch, which connects an 8 bit bus with a 32 bit bus. Its purpose is to buffer data during SCSI data transfers. The bytewise incoming high speed SCSI data is assembled in the BigLatch to produce a stream of 32 bit words at lower speed for the transputer and vice versa.

For convenience lets call the two busses the P (peripheral) data bus and the T (transputer) data bus. At both bus interfaces there are 4 8-bit latches. The BigLatch connects the 8 bit wide peripheral data bus P to the 32 bit wide transputer data bus T (see sheet "MSC schematic").

The BigLatch can transfer the data on the T bus to the P bus and vice versa (transparent mode).

The BigLatch can latch the data of one of the busses and later send this data to the other bus (latch mode).

There are the following functions:

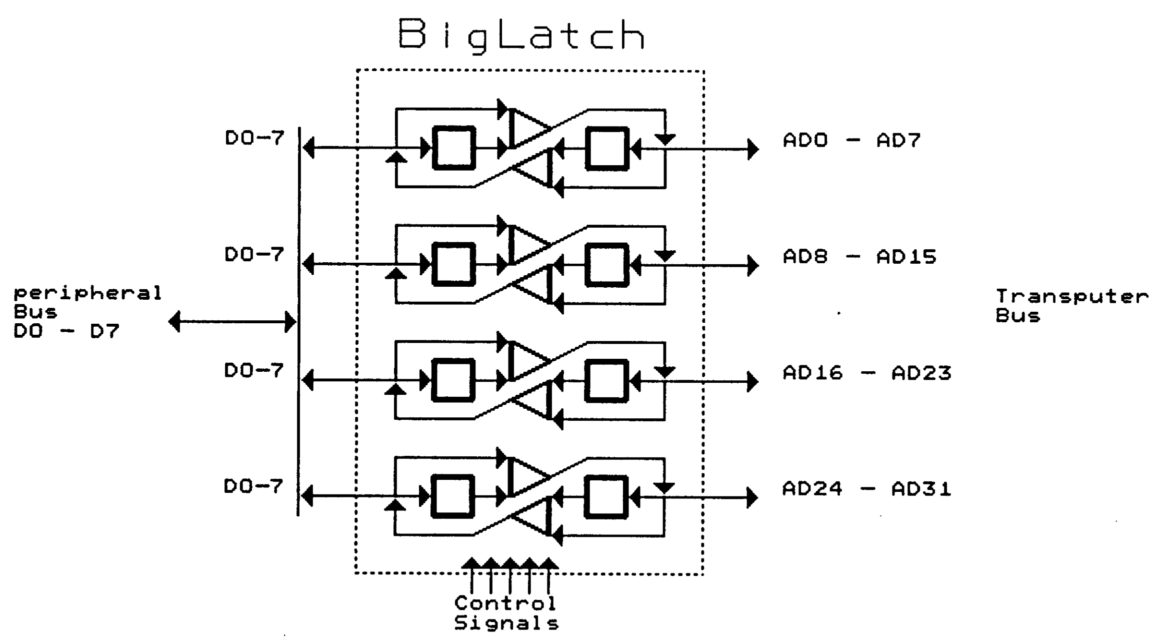
If the transputer reads a register in the SCSI or floppy controler, the BigLatch works like a transparent bus driver. The 8 bit output produced by the accessed controler is transmitted fourfold to the four bytes of the transputer bus.

If the transputer writes to a register in the SCSI or floppy controler, the BigLatch works like a transparent bus driver in the other direction. The contents of the bits 0-7 of the transputer data bus are transmitted to the accessed controler. Bits 8-31 are don't-care bits.

The latch facility of the BigLatch is used only during SCSI data transfers, when the SCSI controler works in DBA mode: The data transfer controler converts the read/write cycles of the SCSI controler to direct the data bytes into/out of the BigLatch. The data transfer controler supervises the SCSI data transfer which takes place between the SCSI controler and the BigLatch and the transputer. The transputer on the other side of the BigLatch gets/supplies the SCSI data by normal memory read/writes accesses.

During SCSI DBA (direct buffer access of WD33C93) read the SCSI controler supplies data bytewise. Up to four bytes can be buffered in the BigLatch. The DBA transfer has to wait (under control of the data transfer controler) until the transputer reads out the four stored data bytes in the Biglatch by a single word read access.

During SCSI DBA write the DBA transfer has to wait until the transputer writes a word (i.e. four bytes) into the BigLatch. Then the DBA proceeds: The data transfer controler directs the BigLatch to send the four stored bytes one after the other to the requesting SCSI controler.



□ 8 Bit Latch

▷ Multiplexer 2x8 Bit -> 8 Bit

PARSYTEC GmbH		
Size	Document Number	REV
A	MSC schematic details	1.1
Date: February 18, 1988	Sheet of	

4.2 The data transfer controler

The main purpose of the data transfer controler is to establish highspeed SCSI data transfer:

- By controling the BigLatch.
- The transputer and the SCSI controler are tightly coupled during high speed data transfer under the control of the data transfer controler.

The data transfer controler operates internally synchronous with respect to the clock of the transputer and the SCSI controler. It operates asynchronous to the floppy controler.

The data transfer controler contains three registers, one state machine, i.e. an automaton, a watch dog timer and 6 user settable flip flops.

The automaton is called **Byte Counter**, which counts how many bytes are written into (during DBA read) or read out (during DBA write) of the BigLatch. It halts DBA transfer or transputer accesses when the BigLatch needs service from the transputer or from the SCSI controller, because it is full or empty.

The automaton is invisible to the programmer and is triggered by transputer and DBA accesses. But the knowledge about it is somewhat helpful in understanding the data transfer controler.

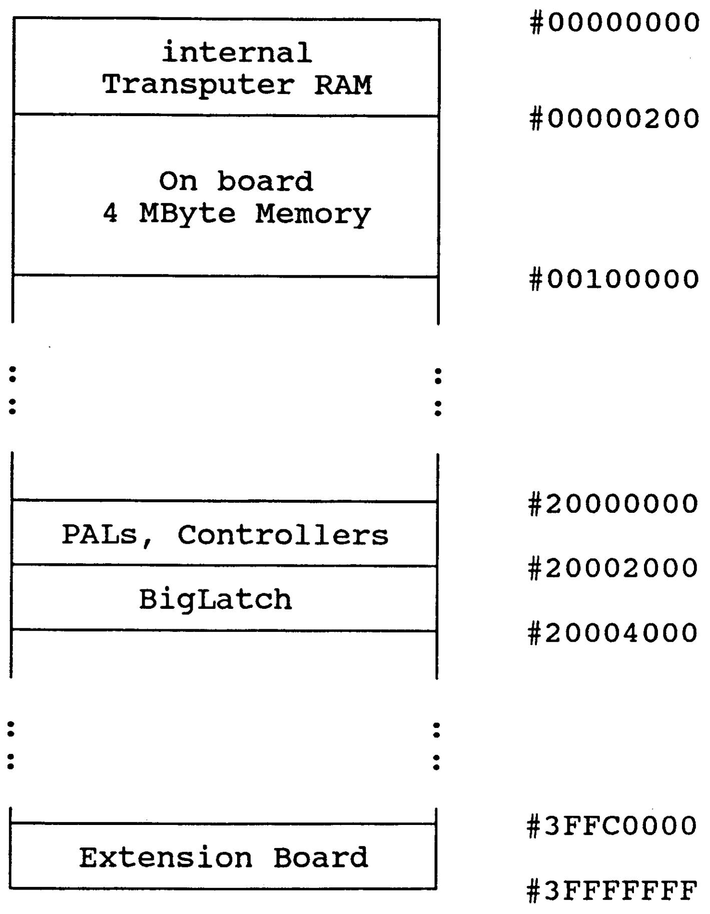
The six flip flops are called PRES, SCSI, READ, DBA, EXTO and DUMMY. For a description see "Overview of OCCAM Address Space".

The state **BigLatch is full/empty** of the Byte Counter Automaton can be polled by the transputer: Bit 5 of the BigLatch Status Register. This enables SCSI data transfer completely under software control in contrast to high speed transfer, where the transputer makes a block move operation between the BigLatch and the memory.

5. Programming the MSC Board

5.1 Overview of the OCCAM Address Space

The T800/T414 transputer can address 1 Giga-words of 32 bit. In OCCAM the address space ranges from #00000000 to #3FFFFFFF (adresses in present OCCAM-2-Implementation PLACEment as word address):



The addresses of all peripherals except BigLatch are located between #20000000 and #20000050. Each peripheral device (controller, PAL) is accessible through the lowest byte of a word (bits 0-7).

Transputer accesses into unused address areas result in an address error (see bit 1 of "Error Register" and "Error and Analyse").

Overview of peripheral addresses

Word address	Bits	Function
#20000000	0-7	Ident bytes from Ident PAL (read only)
#20000020	0-5	Error register of Error PAL (read only)
#20000030	0-3	Link reset out register of Reset PAL (write only)
#20000046 controler	4-7	Status register of data transfer (read only)
#20000046 controler	4-7	Control Register 1 of data transfer (write only)
#20000047 controler	4-7	Control register 2 of data transfer (write only)
#20002000 - #20003FFF	0-31	BigLatch (read/write)

The SCSI and floppy controler share some addresses. They are distinguished by the state of the flip flop SCSI.

——— case flip flop SCSI = 1 ———

#20000040	0-7	Auxiliary Status Register of SCSI controler (read only)
#20000040	0-7	Address Register of SCSI controler (write only)
#20000041	0-7	Register File of SCSI controler (read/write)

——— case flip flop SCSI = 0 ———

#2000040 controler	0-7	Master Status Register of floppy (read only)
#20000041	0-7	Data Register of floppy controler (read/write)
#20000042	0-7	Control Register of floppy controler (write only)
#20000043	0-7	Operations Register of floppy controler (write only)

A read access to any of the floppy or SCSI controler registers, all of which are bytewide, results in a word containing 4 identical bytes. E.g. a read of a status register, which contains #13, returns the word #13131313.

Ident PAL at #20000000

The Ident PAL works like a bytewide PROM and is read only. It can be programmed to hold several bytes for identification or other purposes. There is one default value of #5A.

Error Register of Error PAL at #20000020

This Register latches the following error conditions:

Bit	Error Condition
0	Cleared if ERROR output of transputer was activated.
1	Cleared if transputer accesses an unused address.
2	Cleared if parity error in byte 0 (i.e. bits 0-7).
3	Cleared if parity error in byte 1 (i.e. bits 8-15).
4	Cleared if parity error in byte 2 (i.e. bits 16-23).
5	Cleared if parity error in byte 3 (i.e. bits 24-31).

After the register is initialized (see below) it monitors the 6 error conditions mentioned above. The register latches the first occuring error condition. At the same time the register will be locked, i.e. error conditions which will come up afterwards will not change the status of the error register.

The contents of this register is not destroyed by a transputer reset or analyse/reset. So after a program crash and after resetting the transputer a debug procedure can read out the Error Register.

Initialization of the Error Register

After a power-on-reset there will be random bits in the register, but the register is disabled and con not generate ERRORIN.

The first read of the register will clear the bits and enables it to monitor and latch error conditions, but ERRORIN can not be activated yet. The second read completes the initialization, i.e. ERRORIN can be activated.

ERRORIN is activated after the register has catched an error condition. The register will not be changed by an Analyse/Reset. Two read accesses to the register will clear the register and bring it into proper operation again as mentioned above.

Link Reset Out Register at #20000030

Using this register the transputer can activate the four Link Reset Out signals. To prevent the Link Reset Out signals to be accidently activated by a "crashed" program there is an automaton which monitors this register. The automaton must be brought into the correct state by writing a sequence of values into the register. Writing a #00000000 and any value, which the automaton doesn't expect, will bring it back into the starting state. The value sequence is:

```
#00000000
#00000001
#00000002
#00000003
```

The next written value sets or clears the four Link Reset Outputs:

If bit i, i=0..3, of the value is set then the corresponding Link Reset Output signal of link i is activated. If the bit is cleared the corresponding Link Reset Output is also cleared.

The following OCCAM 2 procedure sends a link reset (i.e. an Analyse/Reset) to some of its neighbours. If bit i in the parameter "neighbour.mask" is set, then neighbour i will get a link reset:

```
PROC  link.reset  (VAL INT  neighbour.mask)
  INT  link.reset.out.reg :
  PLACE link.reset.out.reg   AT  #20000030  :
  TIMER  clock  :
  VAL INT duration  IS  2  :
  SEQ
    link.reset.out.reg  :=  0
    link.reset.out.reg  :=  1
    link.reset.out.reg  :=  2
    link.reset.out.reg  :=  3
    link.reset.out.reg  :=  neighbour.mask
    clock  ?  time
    clock ? AFTER time PLUS duration -- 128 microseconds delay
link.reset.out.reg := 0 -- deactivate all link resets
```

The Control Register 1 at #20000046

This register is write only and contains two flip flops. The transputer must have a copy of the register state somewhere.

Bit	7	6	5	4
	res.	res.	EXTO	DUMMY

Flip flop "EXTO"

EXTO is short for External Output. There is a signal EXTO an the 96 pin connector (pin SBM 22, see Wire Diagram), which can be directly controlled by the transputer. The signal EXTO always follows the state of the flip flop EXTO and can be used for any purpose. This signal is driven by a 74F32 OR gate.

Flip flop "DUMMY"

This flip flop has no defined function yet.

The Control Register 2 at #20000047

This register is write only and contains four flip flops. The transputer must have a copy of the register state somewhere.

Bit	7	6	5	4
	DBA	READ	PRES	SCSI

Flip flop "DBA"

Only used in SCSI mode (i.e. flip flop SCSI = 1). DBA informs the data transfer controler, when the SCSI controller will do data transfer in DBA mode.

DBA = 0: The data transfer controler is not in DBA mode. The transputer can make normal register accesses to the SCSI controler in order to initialize it. BigLatch accesses, although meaningless outside DBA mode, will hang (i.e. wait forever), when WORD = 0, and will succeed, when WORD = 1. DBA = 0 will clear the TO flip flop. The watch dog timer has no effect.

DBA = 1: The data transfer controler is in DBA mode and waits for the SCSI controller to transfer SCSI data bytes via DBA cycles. The transputer cannot access any register of the SCSI controler.
The transputer can access the BigLatch, but has to wait (via WAIT input) until the BigLatch is full (during SCSI Read operation) or empty (during SCSI Write operation). The watch dog timer can set the TO flip flop.

Flip flop "READ"

This flip flop indicates to the data transfer controler the direction of the DBA transfer between the SCSI controler and the BigLatch during DBA mode (DBA = 1).

During floppy operation (SCSI = DBA = 0) the state of READ will be send into the terminal count input of the floppy controler and the inverse state will be send to the DACK input. This feature is necessary for the last byte of a floppy read/write.

Flip flop "PRES"

PRES stands for **peripheral reset.** PRES = 1 will activate the reset input of the SCSI controler or the floppy controler, depending on the state of the SCSI flip flop in the data transfer controler (see flip flop SCSI):

SCSI	PRES	Reset line activated for
0	1	floppy controler
1	1	SCSI controler
X	0	none

The minimal reset duration is controler dependent.

Flip flop "SCSI"

The SCSI controler and the floppy controler both share the same window in the adress space: #20000040-43. The flip flop SCSI in the data transfer controler defines with which of these two controllers the transputer wants to work. The selected controller can be

- accessed ,
- resetted and
- polled via interrupt line (using BigLatch Status Register)

The other controller is not accessible at that time.

SCSI = 1:	This selects the SCSI controler. Accesses to the floppy controler are impossible.
SCSI = 0:	This selects the floppy controler. The SCSI controler is not accessible.

Status Register of data transfer controler at #20000046

This Register is read only. The state of all bits except bits 4 - 7 are undefined.

Bit	7	6	5	4
	TO	WORD	FL_RDY	INT

Flip flop "TO"

TO stands for Time Out. The highest SCSI data transfer rate is achieved, when the SCSI controler works in DBA mode and at the same time the transputer makes a block move on the BigLatch. Transputer BigLatch accesses in DBA mode must wait until the SCSI controler has filled/emptied the BigLatch. This tight coupling can create transputer wait times longer than 16 us, so that refresh cycles will be lost.

The TO flip flop prevents this in DBA mode. TO is cleared at the start of DBA mode and normal block move proceeds. Every BigLatch access starts the watch dog timer. The first access, which lasts longer then 13 us, will set TO, which will cause every subsequent BigLatch access to finish immediately (the transputer is no longer coupled with the SCSI controller). So the data phase is corrupted, but refresh is preserved. The transputer finishes the block move operation and then tests the state of TO. If TO = 1, the transputer knows, that the data phase was broken.

Note: TO must be read before it is cleared by DBA = 0.

Flip flop WORD

WORD must be used on low speed SCSI data transfers, i.e. one cannot be sure that the transputer has never to wait longer than 13us on BigLatch accesses (see TO). In this case the transputer polls the flip flop WORD and if it is set, the transputer can make a BigLatch access.

WORD = 1:	The BigLatch is full or empty. The SCSI controler is halted until the transputer has accessed the BigLatch.
WORD = 0:	The BigLatch is filled or emptied by the SCSI controler and the transputer is waiting until it is full/empty.

FL_RDY line

This bit always reflects the state of the ready line of the floppy bus. The floppy controler does not use the ready line.

INT line

INT = 1 indicates an activated interrupt line from the floppy (SCSI = 0) or SCSI controler (SCSI = 1), depending on the state of the flip flop SCSI.

BigLatch at #20002000 - #20003FFF

Every access into this address area accesses the BigLatch. A transputer read access to the BigLatch reads 4 Bytes at a time. A transputer write access latches 4 Bytes at a time into the BigLatch.

Any transputer access to this address (in DBA mode) clears the automaton **Byte Counter** and the flip flop WORD in the data transfer controler.

A transputer access to the BigLatch lets the transputer wait until WORD becomes set to 1.

Address Register of the SCSI controler at #20000040

If the flip flop SCSI is set a write access at this address writes directly to the SCSI controler Address Register.

Auxiliary Status Register of the SCSI controler at #20000040

If the flip flop SCSI is set a read access at this address reads the SCSI controler Auxiliary Status Register.

Register File of the SCSI controler at #20000041

If the flip flop SCSI is set the transputer can access at this address the SCSI controler Register File.

Master Status Register of floppy controler at #20000040

If the flip flop SCSI is cleared a read access at this address will return the content of the floppy controler Master Status Register.

Data Register of floppy controler at #20000041

If the flip flop SCSI is cleared the data register of the floppy controler can be accessed at this address.

Control Register of the floppy controler at #20000042

If the flip flop SCSI is cleared a write access to this address writes into the Control Register of the floppy controler.

Operations Register of the floppy controler at #20000043

If the flip flop SCSI is cleared a write access at this address writes into the Operations Register of the floppy controler.

5.2 Software Adresses of the Links

After declaration of the channels the following address allocation is valid for the 4 links of the transputer:

```
PLACE  Link0.Output   AT   #0  :
PLACE  Link1.Output   AT   #1  :
PLACE  Link2.Output   AT   #2  :
PLACE  Link3.Output   AT   #3  :
PLACE  Link0.Input    AT   #4  :
PLACE  Link1.Input    AT   #5  :
PLACE  Link2.Input    AT   #6  :
PLACE  Link3.Input    AT   #7  :
```

The event channel has the number #8 (see "Events").

5.3 Events

To interrupt a running process on external events the transputer has a facility called **event.** When the EVENTREQ input of the transputer will be activated the event handling must be supported by software as following: An OCCAM channel must be assigned to the event pin using the construct

```
PLACE  event  AT  #8  :
```

The event process will be inactive and waiting on a statement like:

```
event  ?  signal
```

Within a maximum of 58 processor cycles provided no other high level processes are executed the event process will be restarted after the event pin is triggered. To ensure completion within a minimum time the event process should be a high priority one.

There are five sources which can generate a transputer event:

- The interrupt line of the SCSI controler
- The interrupt line of the floppy controler
- The data transfer controler (see "Flip flop DMATRIG")
- The Error PAL
- The extension board

The event handling can be disabled by Jumper 9.

5.4 Error and Analyse

During development of a program on a transputer network it will sometimes occur that one or more transputers will fail. For debugging purposes it is necessary to find out the cause of the crash after resetting the "crashed" transputers. A special kind of transputer reset is needed which does not destroy the state of the whole transputer system. The Analyse/Reset provides this service. After the Analyse/Reset a special debug procedure can be loaded into the transputer to monitor the internal state.

The Reset PAL executes an Analyse/Reset when a link reset is received by the MSC board. The transputer reacts with an organized shutdown of all processes. Three kinds of error conditions can occur. They are stored in the Error Register of the Error PAL for later inquiry of the error condition (see "Error Register of the Error PAL"):

- Program Error
- Address Error
- Parity Error

Program errors are division by zero, integer overflow, array access outside its boundaries etc. This activates the error flag inside the transputer and the ERROR output of the transputer.

A transputer access to an unused address generates an address error. This will set a bit in the Error Register.

After the Analyse/Reset a debug procedure can monitor the Error Register of the Error PAL.

Jumper J0 decides whether

- both an event and an error condition or
- only an event

can activate the ERRORIN input of the transputer (see "Jumper J0").

J9 connects the ERRORIN input with the otherwise inactive EVENTREQ input ("Jumper J9").

If the MSC board runs as a host the Analyse/Reset will destroy the shutdown status.

The user can start a process when ERRORIN is activated (see "Events"). The activation of the ERRORIN input has the same effect as an internal transputer error.

6. Hardware Details

6.1 Reset signals

There are the following reset signals on the board:

- Power On Reset
- External Reset
- Link Reset In / Out
- Data transfer controler Reset
- SCSI / floppy controler Reset
- SCSI Bus Reset

Power On and External Reset

A Power On Reset of 0.2 seconds will be generated every time the VCC voltage ramps up from below 4.6 volt above 4.6 volt. The External Reset, which is no link reset (pin a24 DIN connector), and the Power On Reset have the same effect:

- The transputer is reset.
- The Reset PAL and the Error PAL is reset.
- The flip flops and automatons in the data transfer controler are reset, i.e. cleared.
- The Link receivers are deactivated during the active reset duration, so there are no incoming link data
- The SCSI and floppy controlers are reset.

Link Reset In

A MULTICLUSTER link consists of two link signals (for both directions) and two link reset signals (for both directions). If one of the four incoming link reset signals are activated, then

- the transputer will get an Analyse/Reset (see "Error and Analyse")
- and the link input receivers will be disabled for the active duration of the link reset signal.

10 microseconds after deactivation of the link reset in signal the transputer is ready to communicate over a link using "Peek and Poke" or for booting.

Link Reset Out

The MSC board can send to any of its four neighbouring transputers a link reset via the MULTICLUSTER links. This is achieved by writing to the Link Reset Out Register (see "Link Reset Out Register"). The state of the MSC board transputer system is not modified by this.

Data transfer controler Reset

The data transfer controler is programmed and reset via its serial EPROM at Power On Reset and External Reset.

SCSI and floppy controler Reset

Both controlers are reset at Power On Reset and External Reset. Each controller can be reset individually by the transputer (see "Flip flop PRES" and "Flip flop SCSI").

SCSI Bus Reset

The RST signal of the SCSI bus is not connected to the SCSI controler. It is terminated with 220/330 resistors and available on the DIN connector, pin b18.

6.2 Jumper Allocation

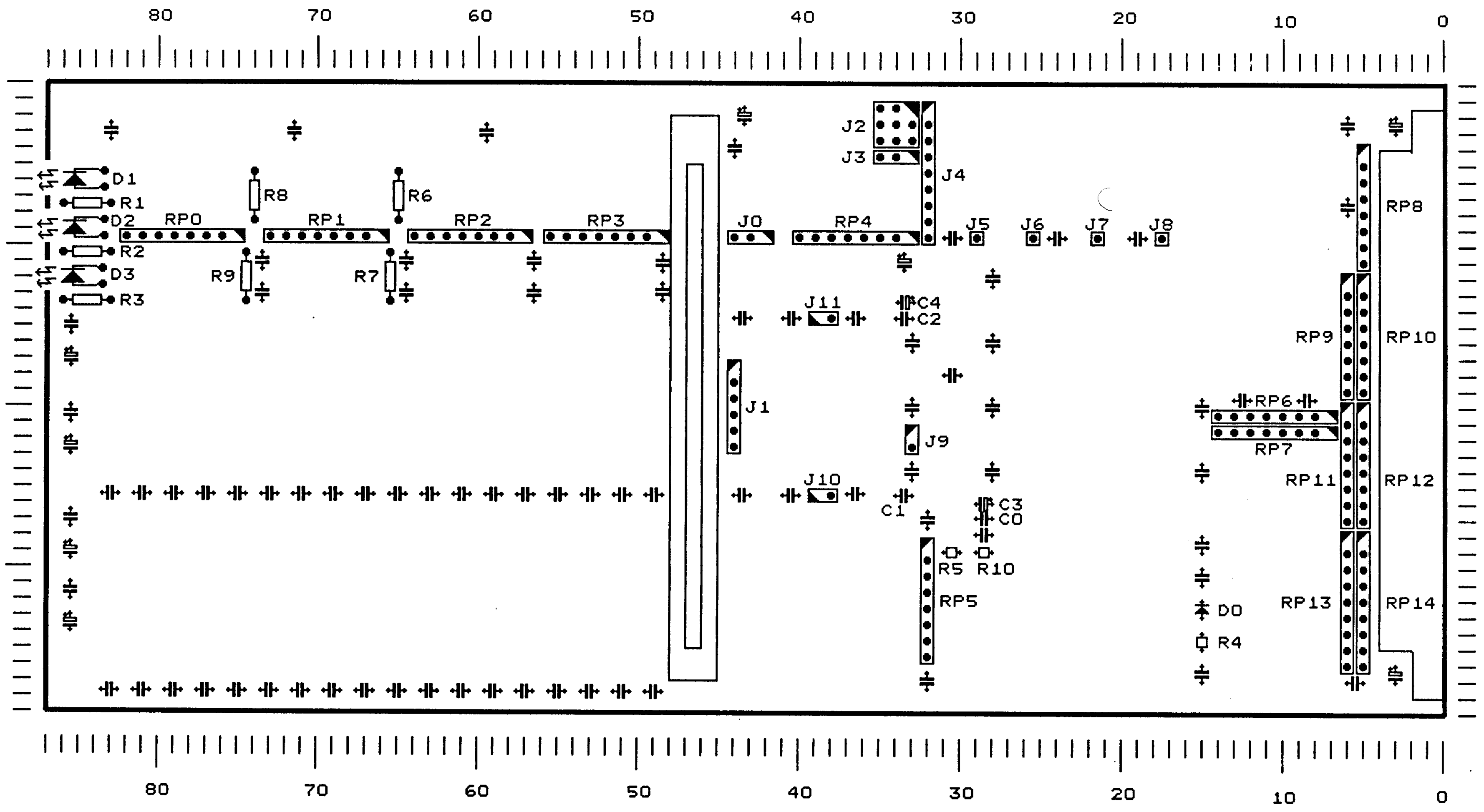
The locations of jumpers on the MSC board are shown at the sheet "MSC Board Layout Discretes".

Jumper J0

The ERROR PAL (U52) activates the ERRORIN input of the transputer when the data transfer controler or the extension board requests an event. J0 enables or disables the activation of the ERRORIN input in cause of an error condition: program error, address error or parity error.

Jumper J1

J1 selects the memory access time of the transputer. The timing of the transputers control signals, which are in memory accesses involved, can be varied (See "External Memory Configuration" in the transputer manual).



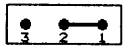
PARSYTEC GmbH

Juelicher Str. 338
D - 5100 Aachen

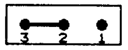
Size	Document Number		REV
A	MSC Board Layout Discretes		1.1
Date: February 10, 1988	Sheet	7 of	8

MSC jumper setting

JØ

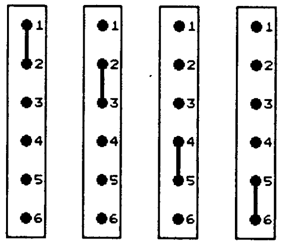


Error conditions can activate ERRORIN.



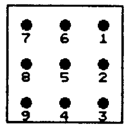
Error conditions can not activate ERRORIN.

J1



Transputer cycles:	3
Access time:	80

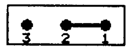
J2



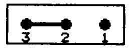
Link speed in MBits/sec.

Link 0	Links 1-3	Jumper		
20	20	2-3	5-6	8-9
20	10	1-2	5-6	8-9
10	10	1-2	4-5	7-8
10	20	2-3	4-5	8-9
10	5	2-3	4-5	7-8
5	5	2-3	5-6	7-8
5	10	1-2	5-6	7-8

J3

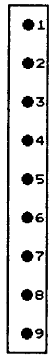


Transputer boots from link.



Transputer boots from ROM.

J4

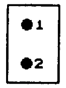


Processor clock speed (MHz)	Jumper		
17.5	2-3	4-5	7-8
20.0	2-3	5-6	8-9
22.5	1-2	5-6	8-9
25.0	2-3	4-5	8-9
30.0	1-2	4-5	8-9
35.0	2-3	5-6	7-8

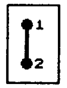
J5, J6, J7, J8

Reserved.

J9

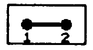


EVENTREQ pin of transputer can not be activated, held low by pull down resistor.

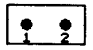


EVENTREQ pin of transputer connected to ERRORIN pin of transputer

J1Ø



There is no extension board.



Extension board installed.

J11

Reserved.

Must be jumpered.

PARSYTEC GmbH		
Title	MSC Jumper Setting	
Size	Document Number	REV
B	MSCJUMP.ORC	1.1
Date:	May 8, 1989	Sheet of

Jumper J2

J2 selects the speed of the transputer links. Link speed can be set to 5, 10 and 20 MBits/sec. Links 1-3 have always the same link speed. The speed of link 0 can be selected separately. Link speed combinations of 5 and 20 MHz simultaneously are not possible.

Jumper J3

J3 defines whether the transputer will boot from ROM or from link.

Jumper J4

J4 defines the internal clock frequency at which the processor part of the transputer runs. This has no influence on the link speed, which is fixed at 5, 10 and 20 MBits/sec.

Jumpers J5, J6, J7, J8

These are 4 soldering pads, which are reserved.

Jumper J9

J9 makes a connection between the ERRORIN pin and the EVENTREQ pin of the transputer. So if this jumper is inserted an activation of ERRORIN of the ERROR PAL U52 also generates an event.

Jumper J10

J10 must jumpered if there is no extension board installed on the MSC board.

Jumper J11

Reserved. Must be jumpered.

6.3 Pin-out of 96-way DIN connector

	a	b	c
1:	Reset Out 0 -	GND	Reset Out 0 +
2:	Link Out 0 -	SCSI DB0	Link Out 0 +
3:	Fl. Revol./min	SCSI DB1	Fl. Head Load
4:	Link In 0 +	SCSI DB2	Link In 0 -
5:	Reset In 0 +	GND	Reset In 0 -
6:	Fl. Ready	SCSI DB3	Fl. Index
7:	Reset Out 1 -	SCSI DB4	Reset Out 1 +
8:	Link Out 1 -	SCSI DB5	Link Out 1 +
9:	Fl. Motor on 2	GND	Fl. Drive Sel. 1
10:	Link In 1 +	SCSI DB6	Link In 1 -
11:	Reset In 1 +	SCSI DB7	Reset In 1 -
12:	Fl. Drive Sel. 2	SCSI DBP	Fl. Motor on 1
13:	Reset Out 2 -	GND	Reset Out 2 +
14:	Link Out 2 -	SCSI ATN	Link Out 2 +
15:	Fl. Direction	SCSI BSY	Fl. Step
16:	Link In 2 +	SCSI ACK	Link In 2 -
17:	Reset In 2 +	GND	Reset In 2 -
18:	Fl. Write Data	SCSI RST	Fl. Write Enable
19:	Reset Out 3 -	SCSI MSG	Reset Out 3 +
20:	Link Out 3 -	SCSI SEL	Link Out 3 +
21:	Fl. Track 00	GND	Fl. Write Prot.
22:	Link In 3 +	Ext. Output	Link In 3 -
23:	Reset In 3 +	SCSI C/D	Reset In 3 -
24:	Ext. Reset	SCSI REQ	Fl. Read Data
25:		SCSI I/O	Fl. Head Select
26:			
27:	VCC	VCC	VCC
28:	VCC	VCC	VCC
29:	VCC	VCC	VCC
30:	GND	GND	GND
31:	GND	GND	GND
32:	GND	GND	GND

6.4 Pin-out of 34-way floppy connector

floppy connector	Signal name	MSC in/out	Function
2			not connected
4			not connected
6	FL_RPM	out	Rounds per minute
8	FL_IDX	in	Index Pulse
10	FL_MO1	out	Motor Select 1
12	FL_DS2	out	Drive Select 2
14	FL_DS1	out	Drive Select 1
16	FL_MO2	out	Motor Select 2
18	FL_DIRC	out	Direction Controll
20	FL_STEP	out	Step Pulse
22	FL_WD	out	Write Data
24	FL_WE	out	Write Enable
26	FL_TR00	in	Track 00
28	FL_WP	in	Write Protect
30	FL_RDD	in	Read Data
32	FL_HS	out	Head Select
34	FL_RDY	in	Ready (used only by DTC)

All odd numbered pins (except pin 1) are connected to Ground.
Pin 1 is not connected.

6.5 Pin-out of 64-way Extension Connector

	a	b
1:	GND	GND
2:	GND	GND
3:	VCC	VCC
4:	VCC	VCC
5:	NOTWB0 *	NOTWB1 *
6:	NOTWB2 *	NOTWB3 *
7:	MEMREQ	MEMGRANT
8:	NOTS0 *	NOTS1 *
9:	NOTS2 *	NOTS3 *
10:	NOTS4 *	NOTRD *
11:	GND	GND
12:	VCC	VCC
13:	AD0	AD1
14:	AD2	AD3
15:	AD4	AD5
16:	AD6	AD7
17:	AD8	AD9
18:	AD10	AD11
19:	AD12	AD13
20:	AD14	AD15
21:	GND	GND
22:	VCC	EVENTACK
23:	EXTENSION SELECT *	EXTENSION WAIT END *
24:	EVENT REQUEST	POWER ON RESET *
25:	AD16	AD17
26:	AD18	AD19
27:	AD20	AD21
28:	AD22	AD23
29:	AD24	AD25
30:	AD26	AD27
31:	AD28	AD29
32:	AD30	AD31

Signals with an asterisk are active low.

The signal EXTENSION SELECT gets active when the transputer accesses the extension address space (see Overview of the OCCAM address space). This signal is NOT validated by any of the strobes NOTS0-4.

During Extension Board accesses the transputer inserts wait states until the signal EXTENSION WAIT END is activated. EXTENSION WAIT END is synchronized for the transputer on the MSC board.

6.6 The LEDs

There are three LEDs on the front panel. When no process is running the only transputer accesses to memory will be refresh accesses. This can be seen as a dark shining memory access LED.

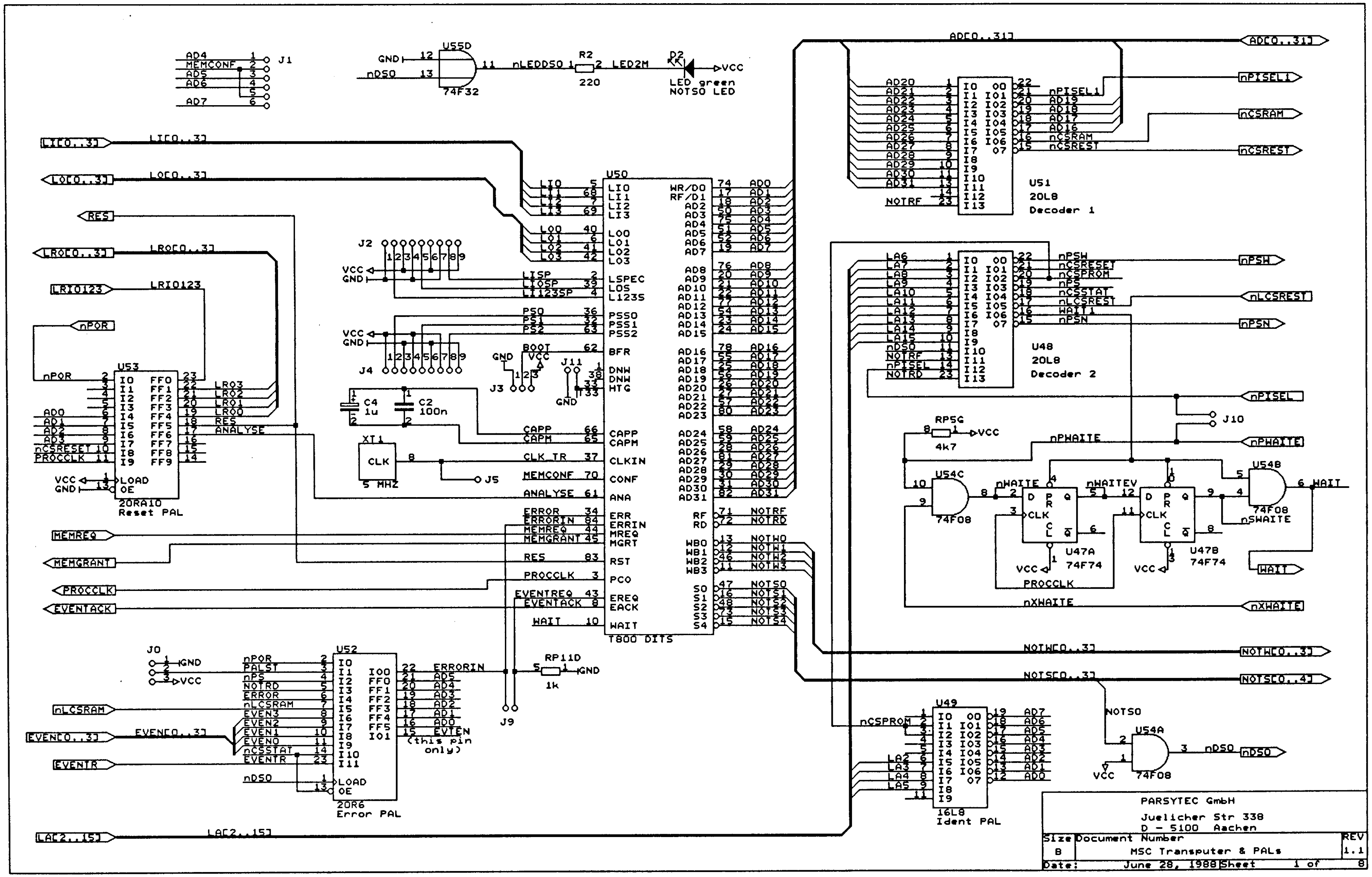
o	Red LED:	5 V Power
o	Green LED:	Transputer access to external memory (NOTS0 signal of transputer)
o	Green LED:	SCSI BSY line

6.7 Technical Data

Size:	Extended Euro card 220mm x 100mm
Layers:	6 layer multilayer (including GND and VCC plane)
Power requirements:	5 Volt 1,7 A standby 4,5 A maximum

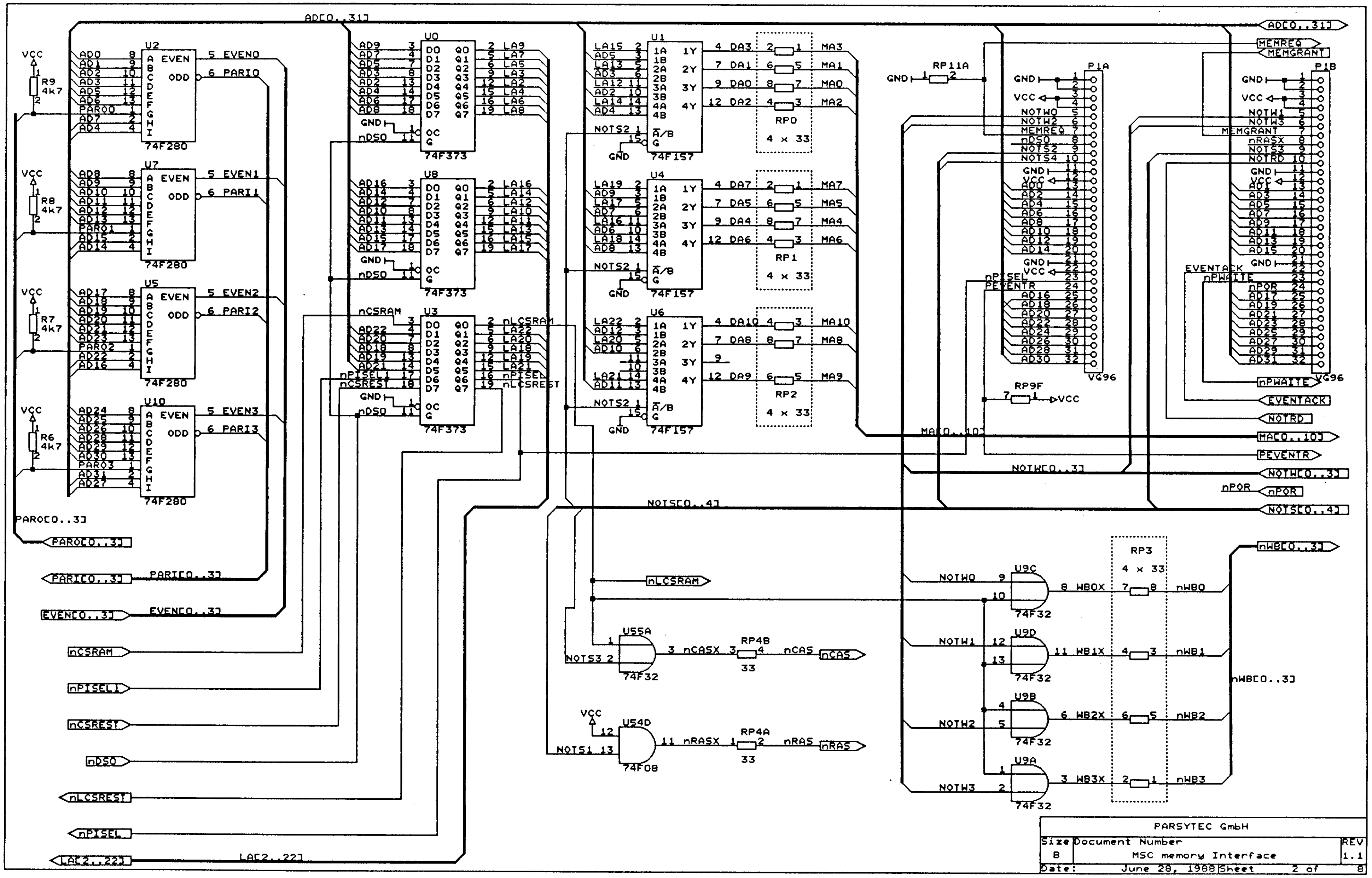
6.8 Wiring diagram

On the following pages the wiring diagrams of the MSC board are shown.

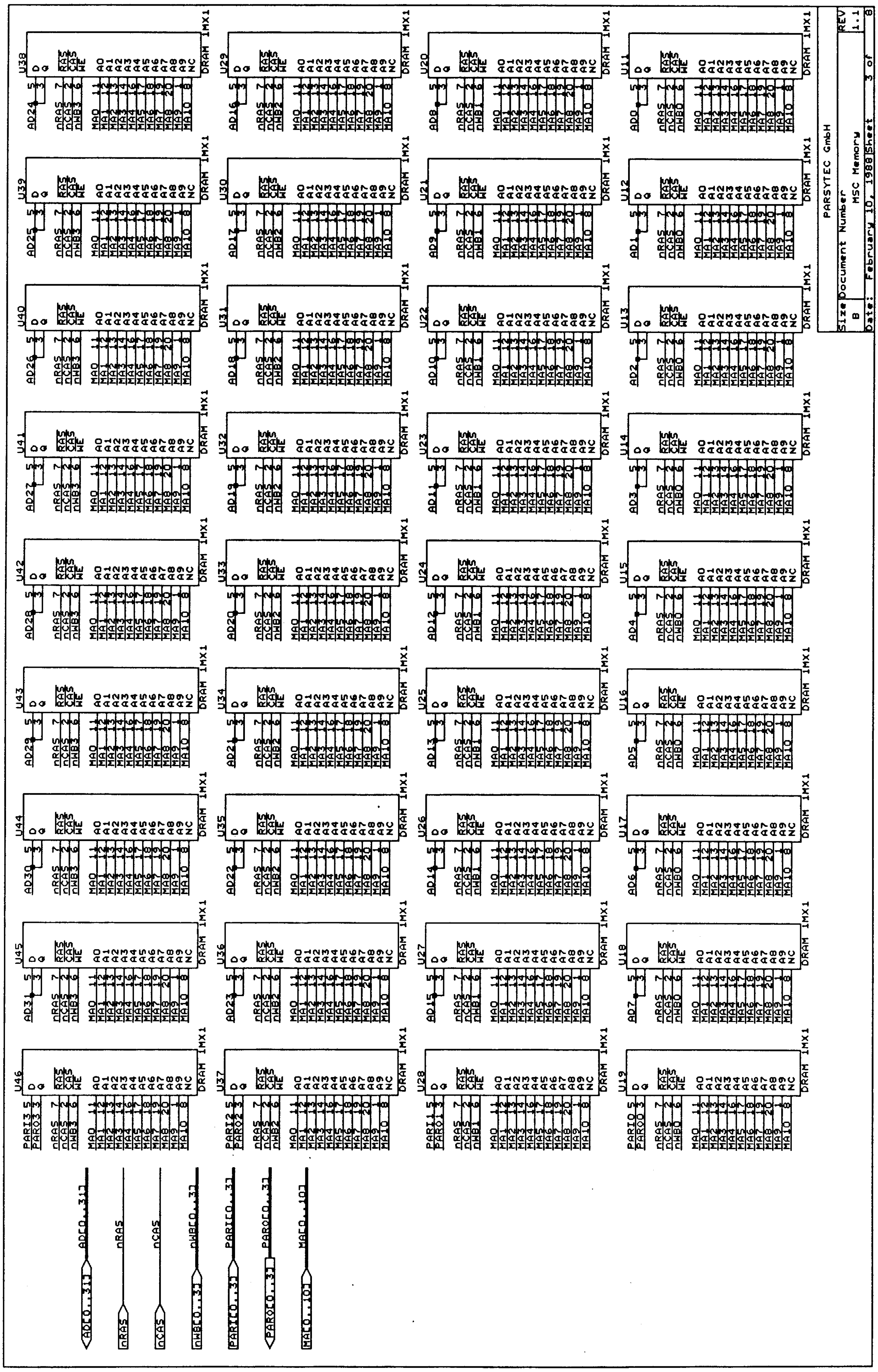


PARSYTEC GmbH
Juelicher Str 338
D - 5100 Aachen

Size	Document Number	REV
B	MSC Transputer & PALs	1.1
Date:	June 28, 1988 Sheet 1 of 8	

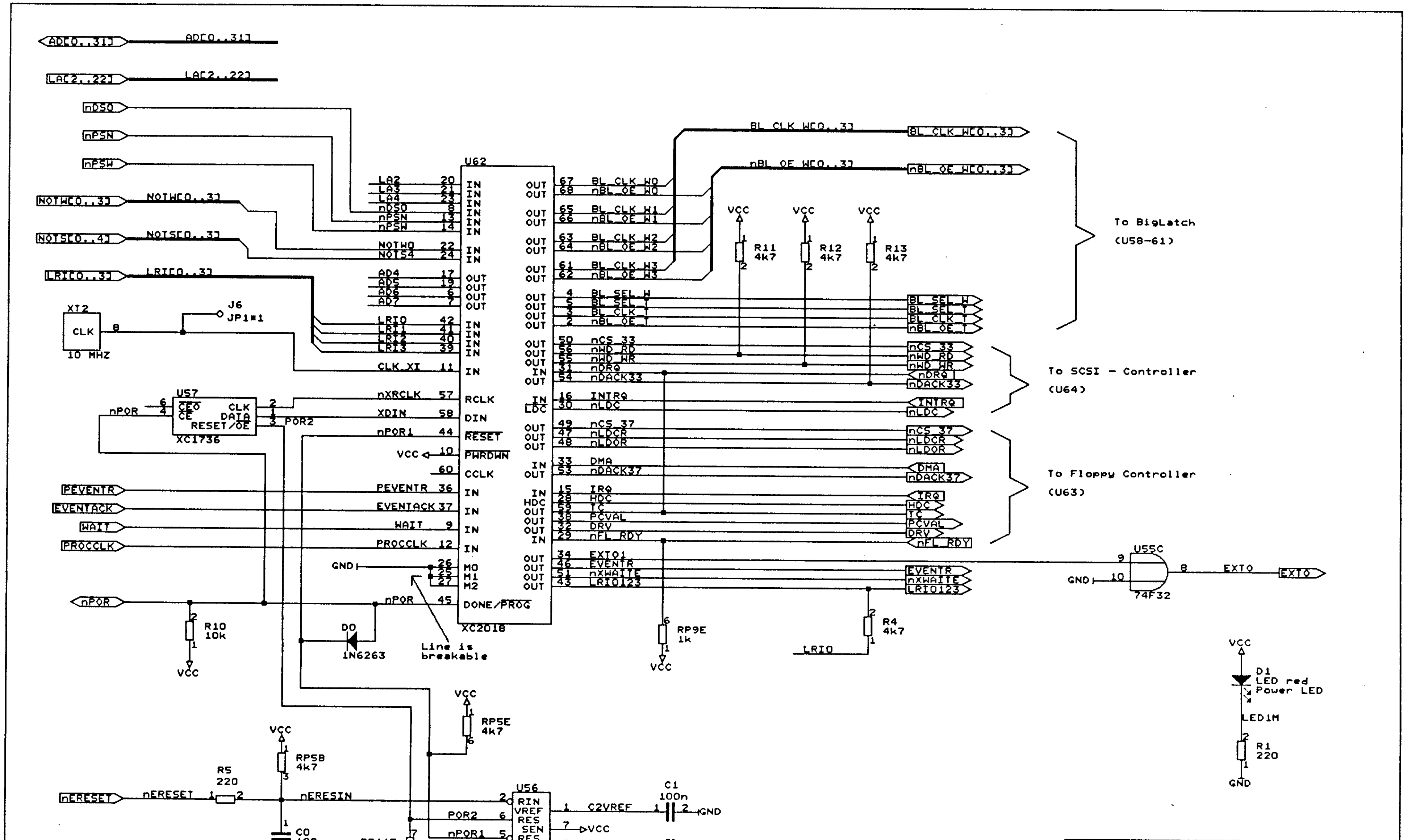


PARSYTEC GmbH		
Size	Document Number	REV
B	MSC memory Interface	1.1
Date:	June 28, 1988	Sheet 2 of 8

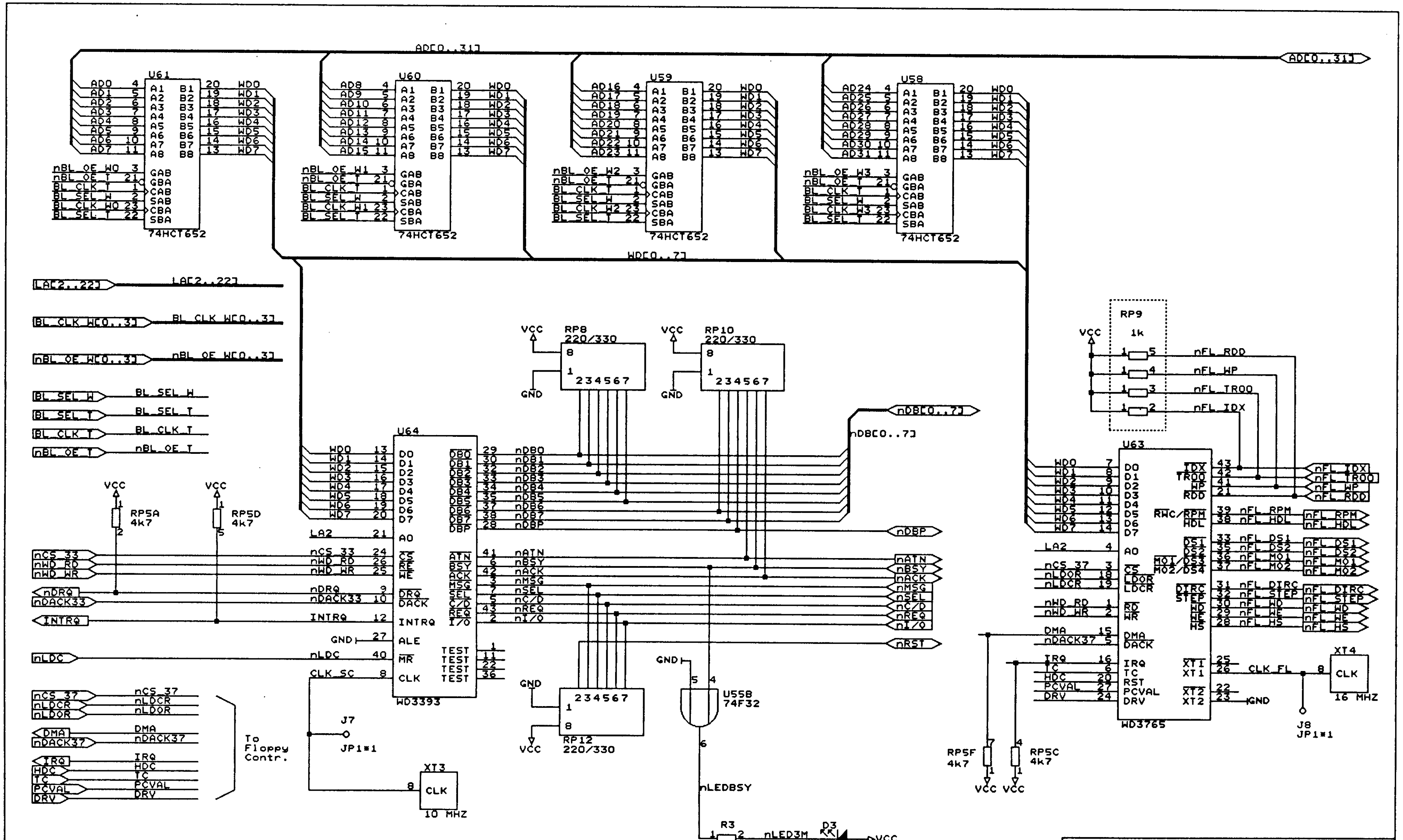

ADC[0..31]
nRAS
nCAS
nWB[0..3]
PARI[0..3]
PARO[0..3]
MA[0..10]
U11 U12 U13 U14 U15 U16 U17 U18 U19
U20 U21 U22 U23 U24 U25 U26 U27 U28
U29 U30 U31 U32 U33 U34 U35 U36 U37
U38 U39 U40 U41 U42 U43 U44 U45 U46
DRAM 1MX1
D
Q
RAS
CAS
WE
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 NC
MA0 MA1 MA2 MA3 MA4 MA5 MA6 MA7 MA8 MA9 MA10
nWB0 nWB1 nWB2 nWB3
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14 AD15 AD16 AD17 AD18 AD19 AD20 AD21 AD22 AD23 AD24 AD25 AD26 AD27 AD28 AD29 AD30 AD31
PARI0 PARO0 PARI1 PARO1 PARI2 PARO2 PARI3 PARO3

PARSYTEC GmbH

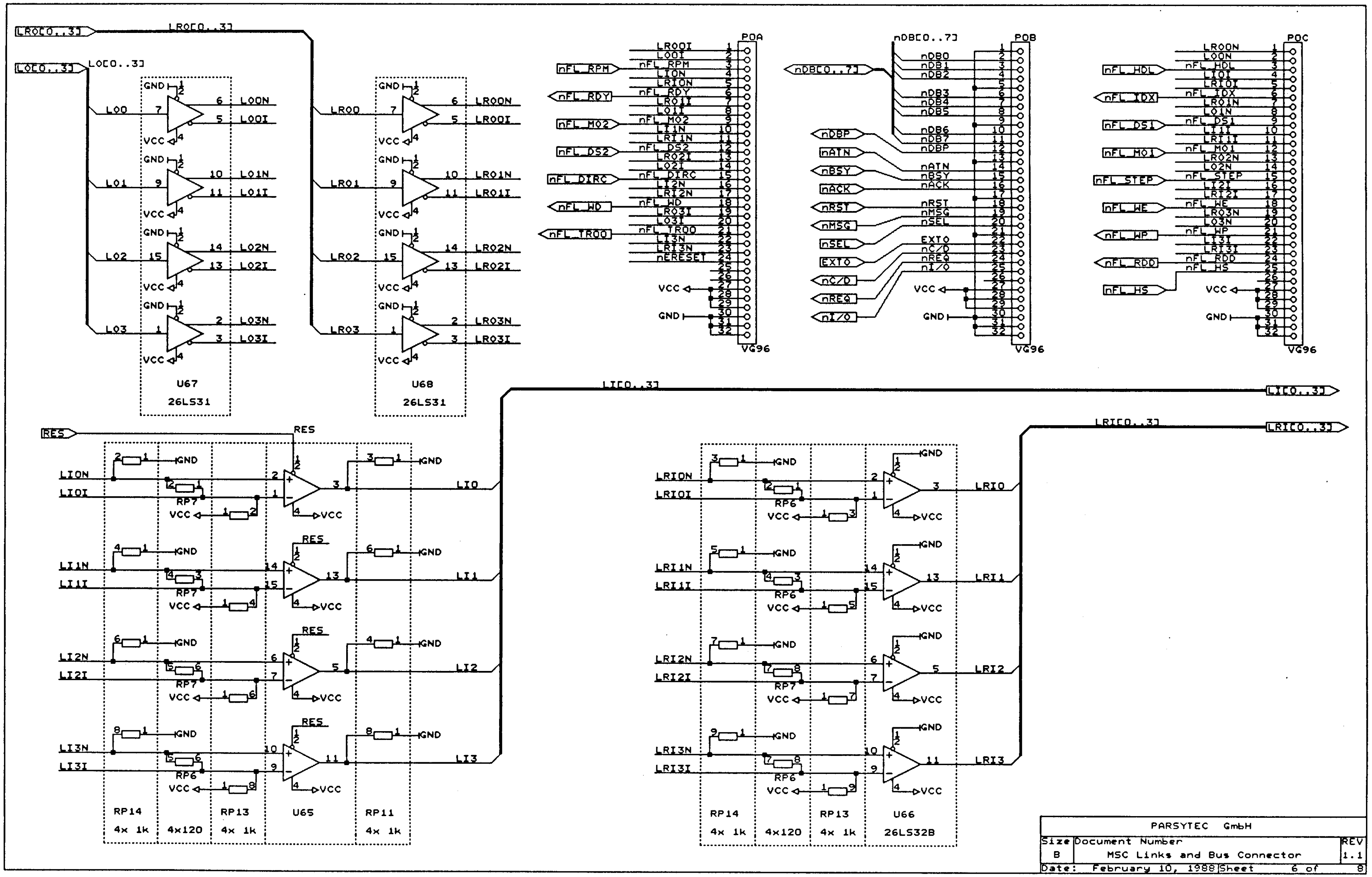
Size	Document Number	REV
B	MSC Memory	1.1
Date: February 10, 1988	Sheet 3 of 8	



PARSYTEC GmbH		
Size	Document Number	REV
B	MSC Xilinx and Reset	1.1
Date: June 28, 1988	Sheet 4 of 8	



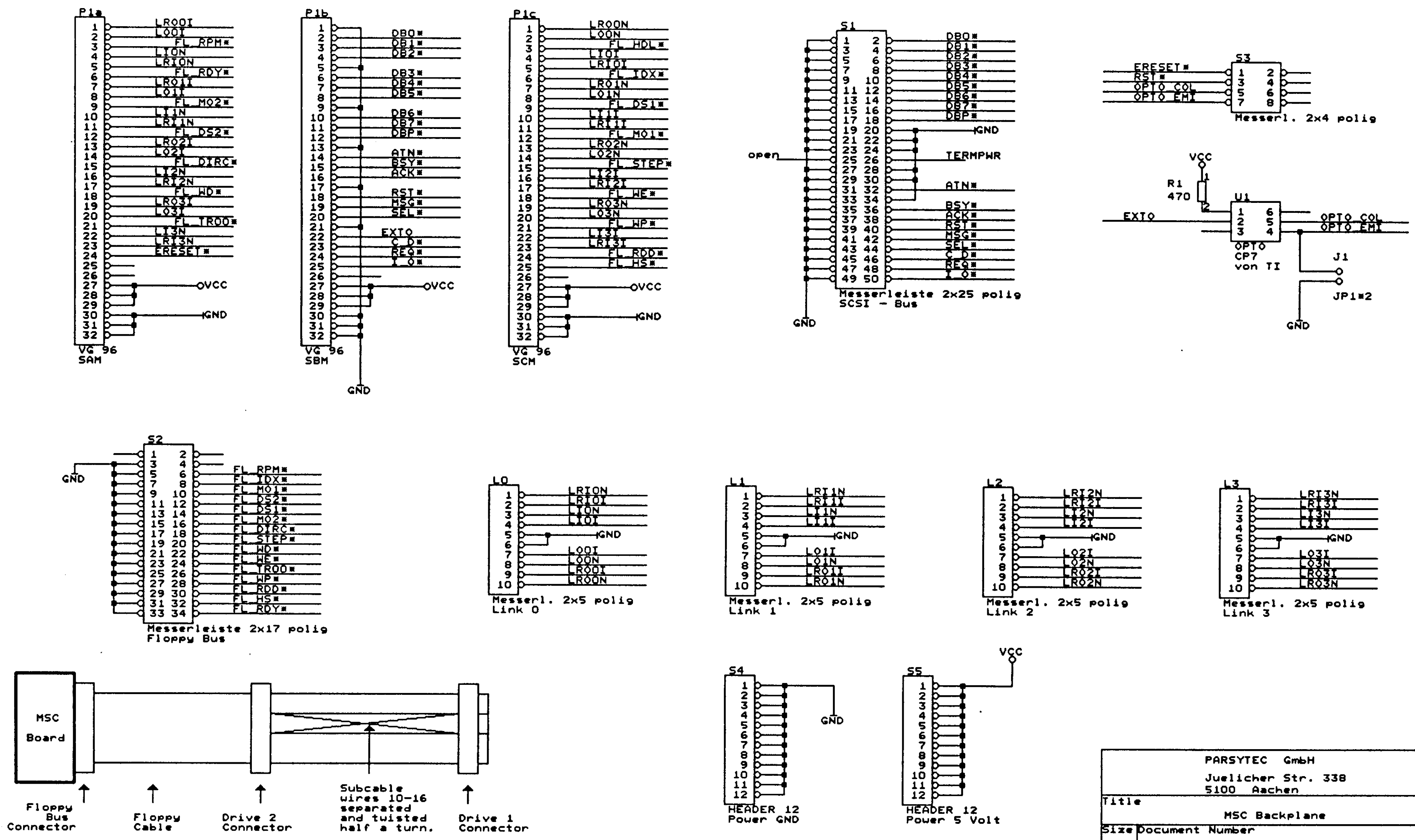
PARSYTEC GmbH		
Size	Document Number	REV
B	MSC SCSI & Floppy Interface	1.1
Date: February 10, 1988	Sheet 5 of 8	



PARSYTEC GmbH

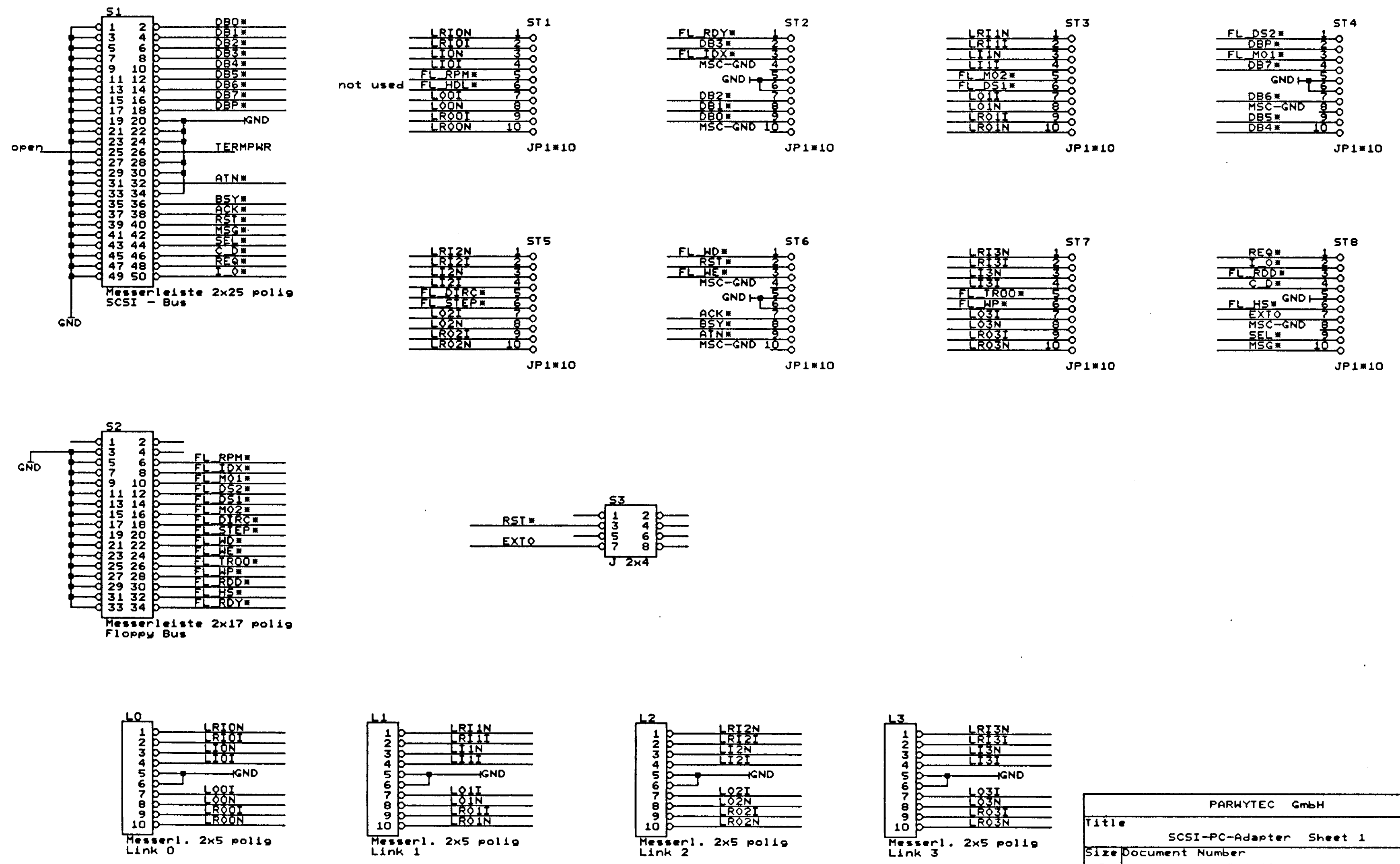
Size	Document Number	REV
B	MSC Links and Bus Connector	1.1

Date: February 10, 1988 Sheet 6 of 8

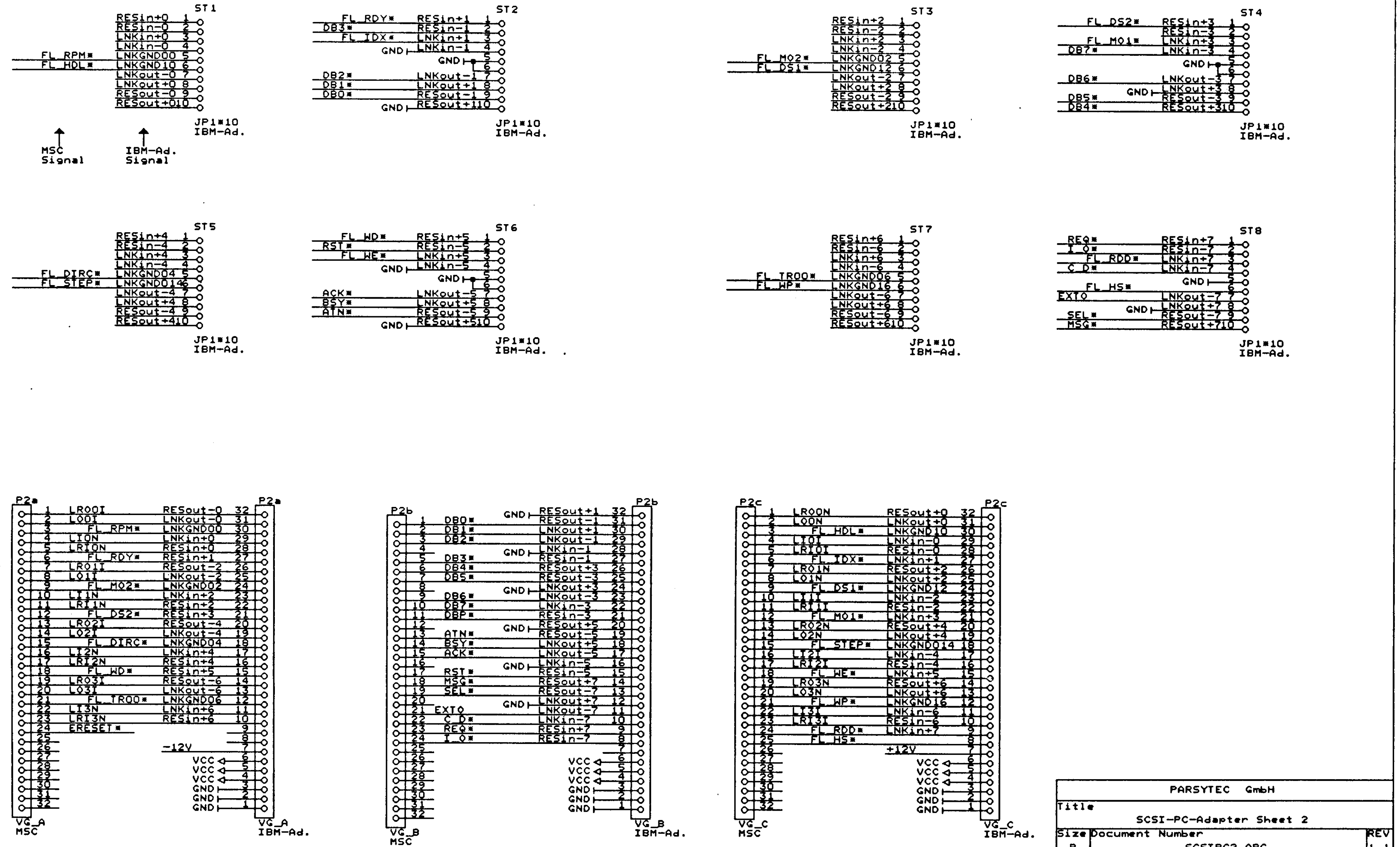


PARSYTEC GmbH
Juelicher Str. 338
5100 Aachen

Title		
	MSC Backplane	
Size	Document Number	REV
B	1	V1.
Date:	April 25, 1988	Sheet 1 of 1



PARWYTEC GmbH		
Title	SCSI-PC-Adapter Sheet 1	
Size B	Document Number SCSIPC1.ORC	REV 1.0
Date: May 8, 1989	Sheet of	

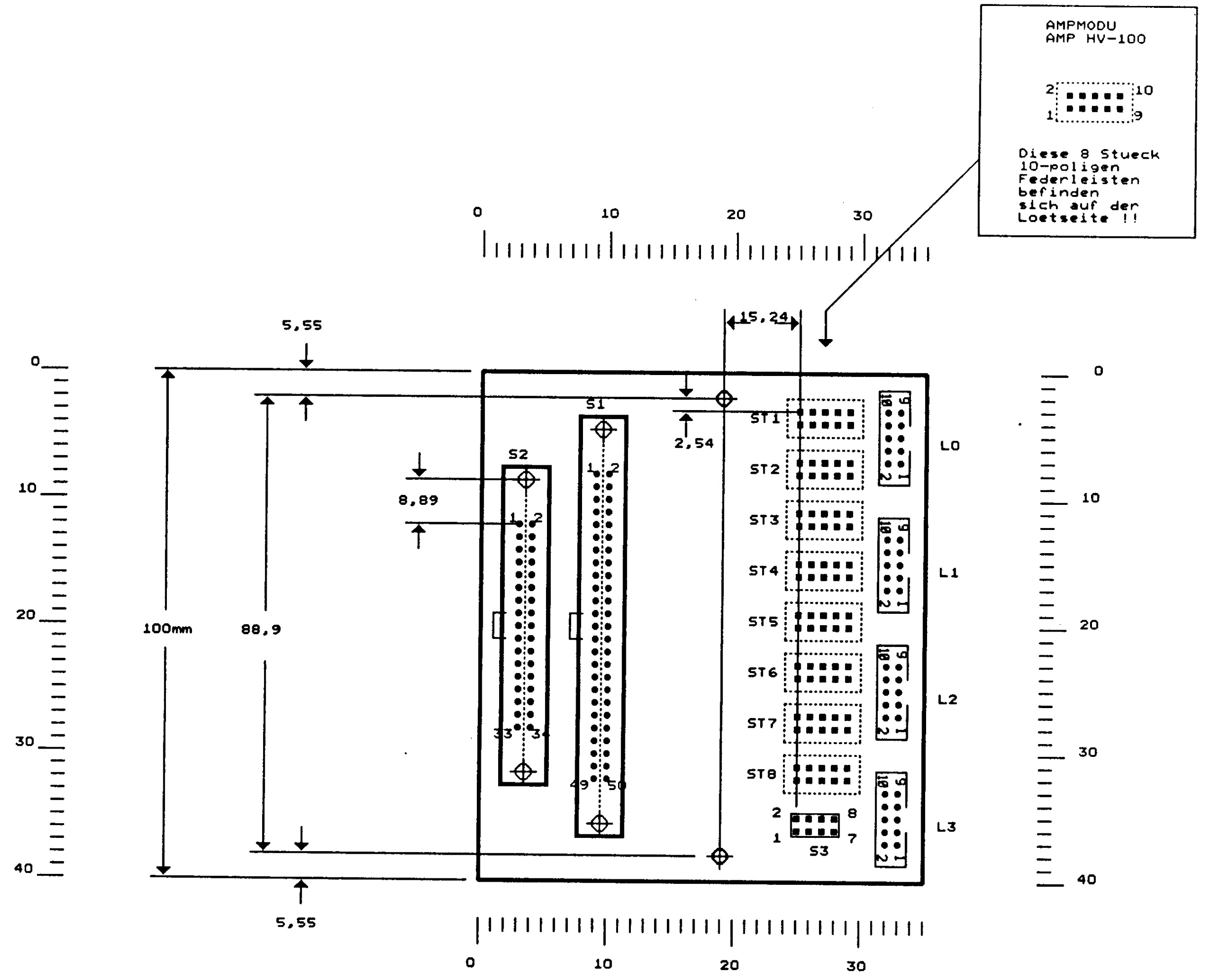


PARSYTEC GmbH

Title: SCSI-PC-Adapter Sheet 2

Size	Document Number	REV
B	SCSIPC2.ORC	1.1

Date: May 8, 1989 Sheet of 1



PARSYTEC GmbH		
Title Layout of SCSI – PC – Adapter		
Size B	Document Number SCSIPCLA.ORC	REV 1.0
Date:	May 8, 1989	Sheet of

PART II Software

7. Introduction

The **msc.driver** module is a software package for the PARSYTEC MSC board to facilitate data input/output to the mass storage devices. It is available as a separate compiled module, called SC, and is fully written in OCCAM. It has an easy to use communication interface. A set of interface procedures is delivered in OCCAM source code (read(), write(), load(), unload(), verify etc.). The user integrates them into his software layer. The msc.driver module hides complicated device features and gives a simple, general and common view of the devices.

This release of the msc.driver software package supports most kinds of

- Winchester Disk Drives
- Tape Storage Devices (Streamers)
- Floppy Disk Drives.

More sorts of devices will be supported in future.

It has the following features:

- Fully written in OCCAM.

- Available as a SC (separately compiled module)

- Communication over four channels: Command Channel, Read Channel, Write Channel and Result Channel.

- A small and easy to use set of communication procedures in OCCAM source is also available:
    Clear driver package
    Init logical unit
    Load medium
    Unload medium
    Read logical block
    Write logical block
    Format, Verify etc.

- SCSI devices are dynamically requested to report their parameters (capacity, sector size, e.t.c.).

- Without recompilation the software package can be dynamically reconfigured at any time:
    o The number and type of devices can be changed.
    o The logical block size can be redefined.

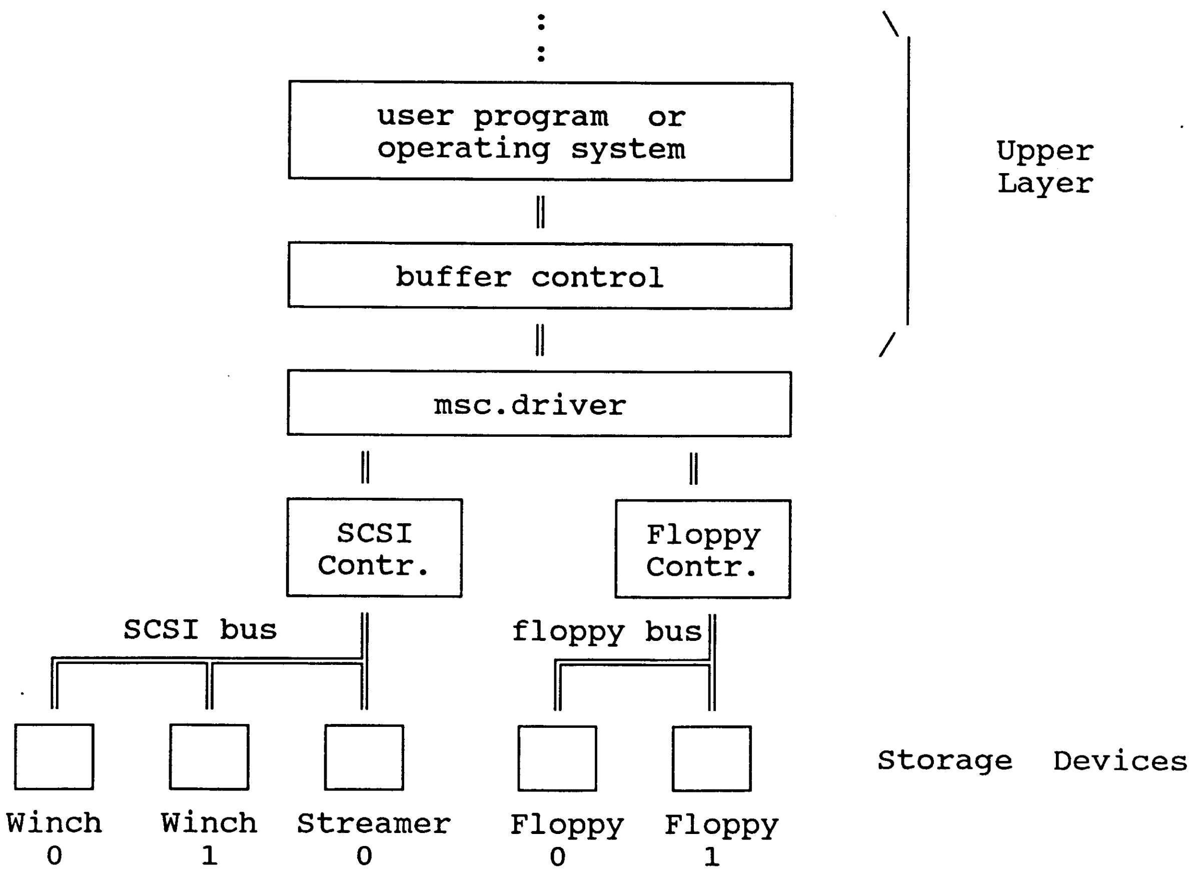
- Makes a trace of all performed internal actions, called the internal protocol. This can be requested by the user and inspected for debugging purposes.

- Supports up to 4 winchesters, 2 streamers and 2 floppies in any combination.

- Does a selftest of SCSI and floppy controlers and devices.

8. The msc.driver software package

The msc.driver module represents the lowest level of a hierarchy of software layers and interfaces directly to the SCSI and floppy controlers:



The msc.driver module is a OCCAM process which is normally started at system start and then running forever.

The layer, with which msc.driver communicates, is a OCCAM process running in parallel with msc.driver and is called the **upper layer.**

The upper layer requests data transfer actions of the msc.driver process by sending command blocks via an OCCAM channel . After msc.driver has received a command block the data is transferred and at last the msc.driver returns a result block to the upper layer process.

The msc.driver process has a single internal buffer of 64 K Byte, which buffers the data of every device on it's way from the devices to the upper layer and vice versa. So the biggest block of data that can be transferred with a single command is 64 K Byte.

The msc.driver process incorporates no buffer strategies. Every block to be read/written will be physically read/written. The block buffering is the task of the upper layer process.

8.1 Interfaces of msc.driver

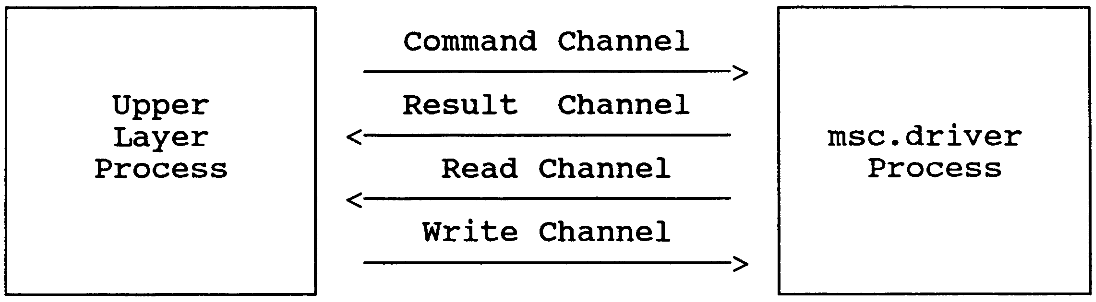
There are two interfaces to the outside world of msc.driver:

- The interface to the SCSI and floppy controller.
- The interface to the upper layer process.

The msc.driver process communicates with the storage devices through the SCSI and floppy controlers. The upper layer can access these controllers also and can do input/output directly with its own procedures. But this should be avoided since the msc.driver process can be confused. In future there will be some additional procedures available for special device handling (testing, formatting etc.).

The msc.driver process communicates with the upper layer process through OCCAM channels. These may be software channels or transputer links. Generally the buffer control software resides on the MSC board itself and not outside in order to use the great amount of local memory for efficiently storing large data sets.

Four channels are provided for communication:



- A **Command Channel** to send commands to the msc.driver.
- A **Result Channel** , over which the msc.driver reports result and status in response of a command.
- Two data channels, called **Read Channel** and **Write Channel** for data transport.

The upper layer process can be viewed as a producer of commands and the msc.driver process as a consumer of commands.

A set of procedures, which handle the communication with the msc.driver process, called **communication procedures,** are supplied in OCCAM source.

8.2 View of the devices

One purpose of the msc.driver process is that the upper layer process has a simplified view of the devices. Hardware details, device parameters, handling of special cases etc. are hidden in the msc.driver process. **The devices should look similar with respect to their controling, data structures and response to facilitate their handling.** The msc.driver communication procedures use descriptive parameters:

- The **class** of a device.
- The **type** of a device.
- The **number** of a device.
- The **type** of a medium.
- The **logical block** of medium.

Devices will be distinguished in floppy drives and SCSI devices because of the following reason: The concept of hiding hardware details is much easier to implement for SCSI devices than for floppy drives.

8.3 The class concept

Devices are categorized in so called **classes**. Devices of the same class have nearly identical behaviour and response, and can therefor be controled by the same set of procedures. At this time there are 4 classes implemented:

- Winchester
- Floppy
- Streamer
- Special

The class **Special** is a class of those commands, which are not related to a certain device (e.g. clear() command).

8.4 The device type concept

Although two devices of a class are nearly identical there will be sometimes minor but important differences between them. Inside the msc.driver process a sort of device is represented by a set of descriptive constants (e.g. number of tracks of a floppy drive etc.). This set is called **device record**.

For every class there exists inside the msc.driver process a small library of common used device records. The device records are numbered starting from 0. The **device type** is the number of a device record. This library will be updated from time to time to include new sorts of devices. The type numbers will be compatible with earlier versions of msc.driver.

Two devices are of the same type if they are descripted by the same device record and such by the same device type.

class	device type	device features
Winchester	0	all SCSI winchesters
Floppy	0	2 Heads, 80 tracks, 250 KBit/sec
Streamer	0	all SCSI streamers

8.5 The device number concept

In every class the devices, which are physically present, are assigned logical numbers from 0 to the maximum value, the **device numbers.** E.g. if there are n winchesters connected to the SCSI bus the logical winchester numbers must range between 0 and n-1.

A certain device is fully addressed by the pair <class,device.number>.

The device number must be distinguished from the physical address of a device: The physical address is assigned once at process start of msc.driver through the init() command.

8.6 The medium concept

To make use of orthogonally command structures all devices are considered to have a changeable medium. So you can even for winchesters send commands to load or unload its medium (which is not considered as an error).

8.7 The medium type concept

The term medium is used e.g. for

- floppy disks
- fixed disk (!) of a winchester
- streamer cartridges
- otical disks   etc.

Analogous to the type concept of devices there is a **type concept** of storage media. That is because a certain device can work on different kinds of media, which must be distinguished of

- storage size
- formatting type etc.

Inside the msc.driver process a medium is represented by a set of descriptive constants (e.g. physical sector size of a floppy disk etc.). This is called **medium record.** In msc.driver there exists for every class a small library of common used medium records. This library will be updated from time to time to include new sorts of media. The media records are represented by numbers, so called **medium types**. The medium type numbers will be compatible with earlier versions of msc.driver.

class	medium type	medium features
Winchester	0	all SCSI winchesters
Floppy	0	2 sided, 80 tracks, 250 KBit/sec, interleave 1:1, sector size 512 Byte starting sector number is 0, 9 sectors per track
	1	same as 0. Starting sector number is 1.
Streamer	0	all SCSI streamers

8.8 The logical block concept

Every medium of a device has an individual physical block length (e.g. a physical sector on a winchester). The physical blocks are ordered in some unique way. So at this lowest level the user sees a sequence of physical blocks numbered in some unique way.

Definition: A **logical block** (of a certain device with a certain medium) or **block** for short, contains exactly one or more consecutive physical blocks. At this level the device is considered to contain a sequence of logical blocks numbered from 0 to some maximum value.

As a consequence there will be some physical sectors unusable (those with the highest sector numbers) for many logical block sizes.

The msc.driver process has the ability to work with **individual** and **dynamic** block sizes:

-> Individual block size means, every physical device or medium can be defined to work with its own block size, e.g. two winchesters can have different block sizes.

-> Dynamical block size means, the block size can be changed at run time (!) by issuing another init() command. The user can experiment with block sizes to find one that suits him best. Every given block length must be less or equal to the internal buffer size of 64 K byte of the msc.driver.

Danger: The block size, with which a medium is written the first time, must be the same for later read/write operations to avoid corrupting the stored data. In general if you decide to use another block size the data on a device or medium will be lost. So the dynamic block size feature is useful during tuning phases.

Example: The n sectors on a medium are numbered from 0 to n-1. If the block size is defined to contain k sectors, then there will be

```
    L :=  n / k      (integer division)
```

logical blocks. The rest of R := n mod k sectors will be unused. The logical block with number B contains the sectors from (B*k) to (B*k + k-1).

8.9 The buffer concept

It is assumed that the upper layer process organizes the handling of the logical blocks, i.e it will implement one or more class dependent buffer algorithms.

Definition: A **buffer** is an array of integers, which can hold the data of a logical block. In OCCAM notation

```
        [block.size] INT  buffer :
```

This defines a buffer to hold blocks of up to 4*block.size bytes.

Note: The logical block size can be less than the buffer size. And

```
    [10] [block.size] INT  buffer :
```

defines a **buffer array** which can hold ten logical blocks.

Implementation examples:

1. -> There is only one buffer, which every class uses.

2. -> There is one buffer array, which all classes share. This means the buffer array will contain in a mixed manner logical blocks of different classes and devices.

3. -> Every class has its own array of buffer arrays.

The communication procedures are designed for case 2. The procedures are doing data transfer on the buffer array defined as "BUffer". The buffer structure can be easily changed by the programmer.

8.10 The internal protocol

The execution of a msc.driver command can be regarded as a large sequence of elementary actions. Msc.driver makes an internal protocol of every action it performs. This protocol can be transferred to the upper layer by the get.protocol() command.

Every elementary action has a unique number, the **action number.** Every performed action is entered into a cyclical buffer. Unsuccessful actions are marked. An entry of the protocol buffer consists of four integers:

action code	parameter 1	parameter 2	parameter 3

If bit 31 of an action code is set, then it is marked to be unsuccessful.
Parameters 1 - 3 further describe the action code. They are not explained for the user.

The command get.protocol() will return the protocol buffer and for every entry a text string which explains the action.

Even after a system crash of the MSC board and after system restart the old internal protocol is available for debugging purposes of the crash. In this case the get.protocol() command must be send before the clear() command. The clear() command clears also the internal protocol.

8.11 Error handling

The general problem of error handling is the following:

To make use of the information hiding principle the upper layer process has no knowledge of the internal state and structure of msc.driver. This is allright as long as everything works ok.

On the other side if something goes wrong with a device the upper layer process wants more error information than just

"Bad Device Access", for example. There are often cases where the error information must be quite detailed.

Another problem is the interpretation of error information. Which layer makes the decision what a severe and what a harmless error will be ?

Msc.driver handles these problems as follows:

-> Every minor action, which msc.driver will perform during the execution of a command, has associated a so called **action number.** If all actions of a command, and hence the whole command, are successful, a so called **ok result** is returned via the result channel. If some actions failed then the action number of the **first** action is returned (see "The result structure"). Additional result and status codes are returned depending on the result.

-> Msc.driver makes no error interpretation, but gives a description of the error.

9. msc.driver commands

9.1 The communication procedures

There is a small set of procedures in OCCAM source available which manage the communication protocol with msc.driver. The user of msc.driver is recommended to use these procedures, but it is also possible to write one's own set. The procedures are:

```
- clear          (...)
- init           (...)
- load           (...)
- unload         (...)
- read           (...)
- write          (...)
- w.format       (...)
- s.format       (...)
- f.format       (...)
- verify         (...)
- nop            (...)
- msc.driver.finish          (...)
- w.mode.sense               (...)
- w.mode.select.sector.l     (...)
- w.mode.select.parity       (...)
- reserve                    (...)
- release                    (...)
```

Every procedure uses the predefined communication protocol of msc.driver. After a command has been send to the msc.driver process data will be transferred and at last a result structure will be returned, which contains in coded form the success or some error number of the requested command.

At process start msc.driver performs nothing but is waiting for the first command. At this state (awaiting a command) the process uses no CPU time, because it waits for channel communication.

The meaning of the result structure is explained in "The result structure".

9.2 Parameters of the interface procedures

To simplify parameter description all parameters of the interface procedures are explained here. Parameters "pid" and "pri" are reserved for future enhancements.

pid	process identifier, an 32 bit integer, by wich the sender process of a command can be uniquely identified.
pri	must range between 0 and 1 and is the priority of a command according to transputer convention: 0 = low, 1 = high priority.
msc.scsi.addr	is the SCSI bus address of the MSC board, which is directly related to the priority for gaining the SCSI bus. It must be between 0 (lowest priority) and 7 (highest priority).
num.w	is the number of winchester drives connected to the SCSI bus. 0 means there is no winchester. Maximum is 7.
num.f	is the number of floppy drives connected to the floppy bus. 0 means there is no floppy drive. Maximum is 2.
num.s	is the number of streamer drives connected to the SCSI bus. 0 means there is no streamer drive. Maximum is 2.
class	is the class of a device.
device	is the device number of a device. Must be between 0 and (number of devices - 1).
block.l	is the defined logical block length (in bytes) of a device. It must be less or equal to the internal buffer length of 64 K Byte and must contain exactly one or more physical sectors of the medium.
type	is the type of a device.
addr	is the physical address of a device. SCSI bus device addresses must be between 0 and 7. Floppy drive addresses must be between 0 and 1. The type parameter must be consistent to the physical device with respect to addressing.
block	is the logical block number.
buffer	is the buffer number.

med.type	is the type of the loaded medium. It is valid only between load() and unload() commands.
pcf	Page control field for winchester mode sense command. 0 <= pcf < 4. See "Mode Sense" Command of your winchester manual.
sector.l	Physical sector length for w.mode.select.sector.l() command.
parity	Parity parameter for w.mode.select.parity() command.

9.3 The **clear()** - procedure

Invocation:

```
clear  ( pid , pri , num.w , num.f , num.s , msc.scsi.adr )
```

After the msc.driver process is started the first command to be executed must be a clear() command to initialize general and device independent variables. Device dependent variables are set up by the init() command. The clear() command is also used to tell the msc.driver process how many devices in every class are connected to the MSC board. It performs a hardware reset of the SCSI and floppy controler and after that the controlers are checked for correct internal function.

clear() can be executed at any time. It tests the SCSI controller and the floppy controller. All prior internal states and data of msc.driver are lost.

9.4 The **init()** - procedure

Invocation:

```
init  ( pid, pri, class , device , block.l , dev.type , addr ,
        lun )
```

After the clear() command an init() command must be executed for every logical unit "lun" of every device in every class to initialize device dependent variables. Medium dependent variables are initialized by the load() command. The hardware and software features of the device are considered to be compatibel with type "type". A logical unit is physically addressed by the pair (scsi address, logical unit number). After the init() command has been issued for a logical unit, it is subsequently addressed simply by it's device number "device".

init() can be executed at any time. All prior internal states and data of the msc.driver process associated with this device are lost.

9.5 The **load()** - procedure

Invocation:

load (pid , pri , class , device)

The load() command assumes that a medium is inserted into device "device" of class "class". Load() forbids medium exchange until an unload() command is given which allows medium exchange. Load() tries to find out what sort of medium is loaded and returns over the result channel the medium type of the medium.

The msc.driver process will use this medium type for all subsequent operations on this medium until an unload() command makes the medium type invalid.

Danger: There is no prevention for medium exchange on floppy drives or even some SCSI devices. So msc.driver will not notice the changing of the medium after a load() command has been executed. This will result in intermixed and destroyed data in the buffers and on the medium.

If the load() command cannot find out the medium number or if no medium is inserted an appropriate result code is returned.

Note: The load() command must also be executed for devices with a fixed medium, because of the general medium concept !

Load() does a test of the device without altering the medium (if inserted). Other devices of this class or of other classes are not influenced.

class	performed actions by load()
Winchester	- WD33C93 internal function test - waiting until motor speed ok - SCSI Mode Sense - computing actual medium dependent variables from mode sense data - SCSI inquiry command - winchester buffer test - reading the first 10 logical blocks - reading the last 10 logical blocks - illegal sector read test

Floppy	- WD36C65 internal function test - reads block 0 of floppy with different medium type numbers, until correct (or no) medium type number found.
Streamer	to be defined.

9.6 The **unload()** - procedure

Invocation:

unload (pid , pri , class , device)

The unload() command in some way has the reverse effect of the load() command. It allows medium exchange in device "device" of class "class", but it doesn't check whether the medium has really changed. Msc.driver no longer assumes a certain media type associated with the device.

9.7 The **read()** - procedure

Invocation:

read (pid , pri , class , device , block , num , buffer)

The read() command reads the "num" numbers of blocks starting with logical block "block" from the device "device" and puts it into the buffer "buffer". The block length is device dependent as defined by the init() command. The read is physically done in the device.

9.8 The **write()** - procedure

Invocation:

write (pid , pri , class , device , block , num , buffer)

The write() command sends "num" logical blocks in buffer "buffer" to the device "device" where it overwrites data starting at block "block". The block length is device dependent as defined by the init() command. The write is physically done in the device.

9.9 The **w.format()** - procedure

Invocation:

```
w.format    ( pid, pri, device, cdb1, cdb2, cdb3, cdb4, cdb5,
              buffer , list.1 )
```

The w.format() command formats the winchester "device". In general there is more than one way to format a winchester (using the primary and/or growing defect list). So the procedure gets 5 integer parameters, called cdb1 - cdb5, which msc.driver will convert to 5 bytes and insert them into the Command Descriptor Block (CDB Byte 1 - 5) of the SCSI command, which is send to the winchester. The procedure will always send the list in buffer "buffer" of length "List.1" to msc.driver, whether it will be used there or not. The block contains the appropriate defect list, if any. The init() command can be used to temporarily define a block size of appropriate size to hold the lists.

The user must handle the defect lists. For explanation of the various kinds of formatting see "Command Description for direct Access Devices" in the ANSI SCSI manual.

W.format() does no verification. This must be done with the verify() command.

After the procedure has been started, it returns when the winchester has been formatted.

To change the physical sector size of a winchester, the w.select.sector.1() command must be issued and after that the w.format() command.

9.10 The **f.format()** - procedure

Invocation:

```
f.format  ( pid , pri , device , medium.type )
```

The f.format() command assumes there is a floppy disk inserted into floppy disk drive "device". The floppy will be formatted according to the parameters of "medium.type". The floppy is not verified. A write protected floppy will return "not.loaded" as medium status (according to WD37C65). A successful f.format() will return medium type "no.medium.type".

9.11 The **s.format()** - procedure

To be defined.

9.12 The **verify()** - procedure

Invocation:

```
verify  ( pid , pri , class , device , media.type )
```

The verify() command checks whether all physical sectors on a medium are readable. The whole medium is read, but no data transfer is involved. The purpose of the verify() command is to find out bad sectors. Result structure is the same as the read() command, except that result integer 3 contains the media type.

9.13 The **msc.driver.finish()** - procedure

Invocation:

```
msc.driver.finish ( pid , pri )
```

This command tells the msc.driver process to finish itself. This command is normally not used except for test purpose, because the msc.driver process is assumed to run steadily. No actions on devices are performed.

9.14 The **nop()** - procedure

Invocation:

```
nop  ( pid , pri )
```

The nop() command has no effect. The communication protocol is executed and msc.driver returns status ok.

9.15 The **w.mode.sense()** - procedure

Invocation:

w.mode.sense (pid , pri , w.nr , pcf , buffer)

W.mode.sense() reads from the winchester all (up to 4) sets of mode sense data. The pcf value (page control field) defines which set of mode sense pages is returned as a block into buffer "buffer". So to get all 4 sets of mode sense pages you have to issue w.mode.sense() 4 times with pcf=0..3.

The first 4 byte integer in the returned block contains the length in bytes of the mode sense data. The bytes after the first integer contain the mode sense data exactly as the winchester has supplied them.

A side effect of w.mode.sense() is to perform internally a load() command.

9.16 The **w.mode.select.sector.l()** - procedure

Invocation:

w.mode.select.sector.l (pid , pri , w.nr , sector.l)

W.mode.select.sector.l() enables the user to change the physical sector length of a winchester. The specified sector length comes into effect only after a w.format() command. Sector.l specifies the length in bytes.

9.17 The **w.mode.select.parity()** - procedure

Invocation:

w.mode.select.parity() (pid , pri , w.nr , parity)

w.mode.select.parity().

9.18 The **get.protocol()** - procedure

Invocation:

```
get.protocol   ( pid , pri , buffer.size , buffer , init ,
                 more )
```

Msc.driver will copy its internal protocol into the buffer "buffer". See "The internal protocol". One entry of the protocol occupies 64 bytes:

Byte 0 - 39:	Text string describing the action	
Byte 40 - 43:	The action number	(4 byte integer)
Byte 44 - 47:	Parameter 1	(")
Byte 48 - 51:	Parameter 2	(")
Byte 52 - 55:	Parameter 3	(")
Byte 56 - 63:	unused	will be 0

buffer.size is the size of buffer "buffer". In general the total protocol will fill more than one buffer. In this case the procedure returns the boolean variable "more" = TRUE, whether there are more protocol entries, which must be transferred by another get.protocol - call. "more" = FALSE indicates that the last entries have been transferred.

The first get.protocol() call must initialise the protocol transfer by setting "init" to TRUE. This first call will not fill entries into the buffer. Subsequent calls must set init to FALSE, so get.protocol() will return filled buffers.

9.19 The **get.params()** - procedure

Invocation:

```
get.params   ( pid , pri , class , device , buffer )
```

After loading a medium and sending the load() command, the user will get some parameters describing the medium and the device by sending the get.params() command. Get.params() will return a block into buffer, the first words containing some information:

Word number	Device / medium parameter
0	psysical sector length in bytes
1	total number of sectors on medium
2	total number of blocks (depends on block size)

9.20 The **start.stop.unit()** - procedure

Invocation:

```
start.stop.unit  ( pid , pri , class , device , start )
```

Some direct access devices, mainly winchesters, can switch on and off the spindle motor. This may be used to lengthen the life time of the drive. The boolean start causes the motor to run, when true. The motor will halt otherwise.

9.21 The **send.command()** - procedure

Invocation:

```
send.command  ( [] INT command )
```

The array paramater "command" is transferred to the msc.driver process, where it is interpreted as a command.

9.22 The **reassign.blocks()** - procedure

Invocation:

```
reassign.blocks  ( pid , pri , class , device , buffer )
```

This procedure is used to reassign bad sectors of devices, normally winchester devices. Bytes 2 and 3 in buffer "buffer" comprise a list length (defect list) of bytes. This list sends the msc.driver process to the device. Zero length is allowed.

9.23 The **reserve()** - procedure

Invocation:

```
reserve  ( pid , pri , class , device , free )
```

The communications procedure tries in a loop to reserve the device. If this will not be successful for 1 minute, then result parameter will be returned false.

9.24 The **release()** - procedure

Invocation:

release (pid , pri , class , device)

The device "device" is released.

9.25 **Time relationship of commands**

After the msc.driver process has been started, commands in the following order must be issued:

-> The first command must be a clear().

-> After that for every device in every class an init() must be issued.

-> After that, before accessing the medium of a device, a load() must be executed (with inserted medium of course).

-> Now the medium of the device can be accessed by giving read(), write() and verify() commands in any order.

-> Medium changes are only possible after an unload() and before a load() command.

-> The format() command needs no proir load() command.

10. The communication protocol

A simple protocol is used to communicate with the msc.driver process. This protocol is hidden in the communication procedures, but will be explained here.

One aspect is protocol safety. For example the upper layer process wants to read a block and sends a miss-spelled command. Then it waits for the data on the read channel. The msc.driver rejects the command. If it would not send any data the upper layer process will hang.

Definitions

A **command** is a small integer array of variable length, which contains the requested action for the msc.driver process and parameters, if any.

A **result** is a small integer array of variable length, which contains at least two integers:

- The **process identifier.** To this process the result must be transferred. The process is waiting for the result.

- The **error code** says whether or not the execution of the command was successful or produced a catastrophic failure.

10.1 Channel types

All four communication channels of msc.driver are defined to transfer dynamic arrays, i.e. first the array length is transferred and then the array:

The command and result channels are defined as following:

```
PROTOCOL  command.type  IS  INT :: [] INT :
```

The data channels are defined:

```
PROTOCOL  data.type     IS  INT :: [] INT :
```

The maximum array sizes msc.driver can receive or will send:

Channel	Maximum Array Size
c.command	32 Integers
c.result	32 Integers
c.read	16384 Integers (= 64 K Bytes)
c.write	16384 Integers (= 64 K Bytes)

10.2 The phases of communication

Communication is always initiated by the upper layer process.

1. Step: The upper layer process sends a command over channel c.command. The msc.driver process, waiting for a command, receives the command.

2. Step: After the command transfer is done, there is always a data transfer over the c.read channel. If there is no read data involved a null array is send by msc.driver.

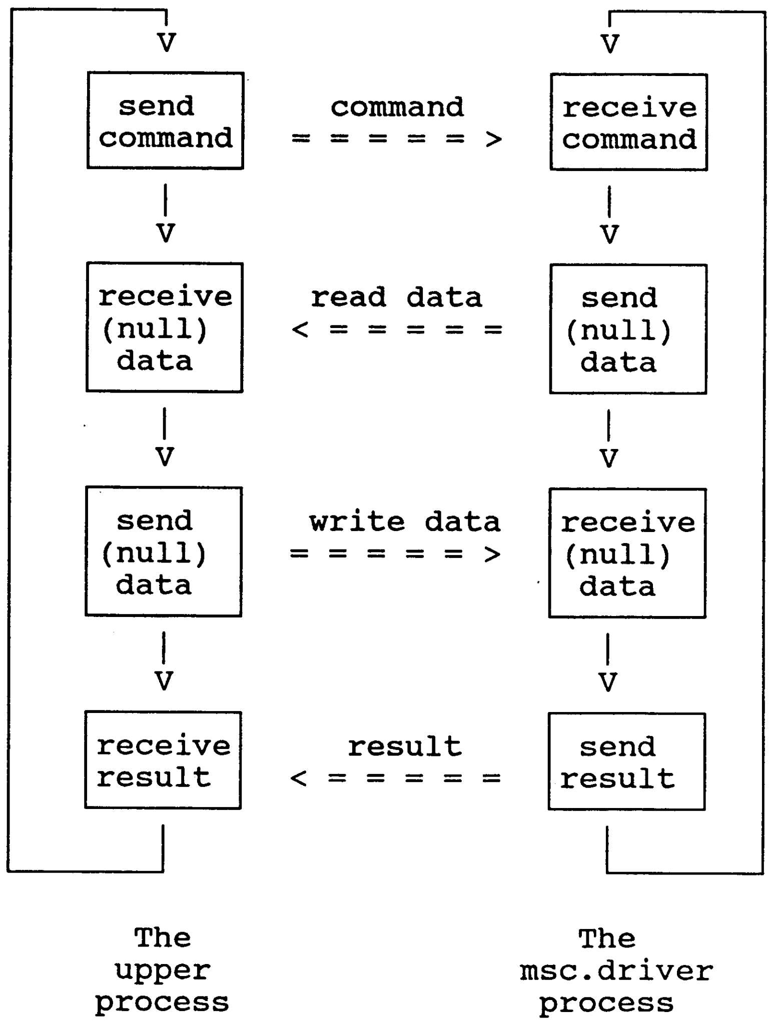
3. Step: After the read data transfer is done, there is a data transfer over the c.write channel. If there is no write data a null array must be send by the upper layer.

4. Step: The msc.driver process sends a result over the c.result channel. After that it waits for the next command. The upper layer process receives the result. The protocol is done.

The msc.driver process decodes the command between steps 1 and 2. If there is read data involved command execution takes place after command decoding and before step 2. If write data is involved command execution takes place between step 3 and 4.

The procedure dummy.c.read() is used to receive null, wrong or unexpected data just to fulfill the protocol phases.

The procedure dummy.c.write() is used to send null data just to fulfill the protocol phases.



Timing of protocol execution.

10.3 The result structure

A result is a small sequence of integer values in the array "result". Its length is 1 at least. The length is contained in the variable "result.l". The interpretation of a result is command dependent. After the execution of a command result[] and result.l are valid.

The first integer of result (result [0]) contains the identifier of the process, to which the result belongs.

result [1] has a common function for all results. It contains the error number of the first error of the internal protocol, or 0 if no error occurred.

If the command performed well, result [1] = 0 , meaning "ok status". Otherwise the first unsuccessful action number of the failed command is returned. The meaning of action numbers is shown in "Action Numbers". The procedure wait.for.result() sets the global variable "ok", if the received result is successful.

If there are more than two integers in result , the meaning of the rest is command and class dependent:

Winchester result structure

If the msc.driver process performs a winchester command successfully, the result length is 2 and result [1] = 0.

If anything is to be reported, the result structure has the following form:

result [0] :	process identifier
result [1] :	internal msc.driver action number
result [2] :	sense key
result [3] :	first byte of request sense data
result [4] :	second byte of request sense data
:	:

result [2] contains the sense key of the winchester (according to the ANSI SCSI manual), which caused the malfunction:

result [2] = 0:	No error.
result [2] = 2:	Winchester not ready. Perhaps the motor has not the correct speed.
result [2] = 3:	Medium error. The winchester can not read or write a physical sector (bad sector).
	etc.

After that the request sense data are following: The integer result [3] contains the first sense data byte and so forth. In the case of a medium error the request sense data normally contains the bad sector number (see your winchester manual).

Floppy result structure

Floppy commands return a result structures of length 2 and 3. If msc.driver performs a floppy command successfully result [1] has the value "medium.ok" (=0).

If anything is to be reported, the result structure has the following form:

result [0] :	internal msc.driver action number
result [1] :	Medium status
result [2] :	Medium type

Medium Status Values

Medium status has the following meaning:

0	Floppy command was successful, medium is inserted
1	No medium is loaded
2	Medium is write protected
3	Sector not found
4	CRC Error
5	Medium is not formatted

10.4 The msc.driver library

There is a fold in the msc.driver software package, called mscdrive.tsr, which contains the msc.driver library. The interface procedures and the msc.driver source use this library for command and result constants etc.

10.5 Action Numbers

Action numbers and their meaning can be seen in the SC (separate compiled module) "text.of.action" in the source code of the upper layer of the msc.driver software package. The action numbers are normally not needed by the user.

PART III Installation

1. Hardware Installation

The MSC board is delivered with one of two kinds of backplanes:

For operation inside a PC:
The SCSI-PC backplane. See sheet "SCSI-PC Adapter" for mounting. This needs a BBK-PC adapter. The MSC and the BBK-PC are connected with their 96-pin DIN connectors, and the SCSI-PC adapter is plugged onto the BBK-PC adapter. The 4 MSC links, the SCSI bus and the floppy bus are routed through the 8 link connectors of the BBK-PC to the SCSI-PC adapter.

For all other environments:
The Multicluster backplane, which interfaces directly to the 96 pin DIN connector of the MSC. It provides 4 10-pin Multicluster link connectors, 1 50-pin SCSI connector and 1 34-pin floppy connector.

Normally the first steps to test the operation of the MSC involves a winchester:

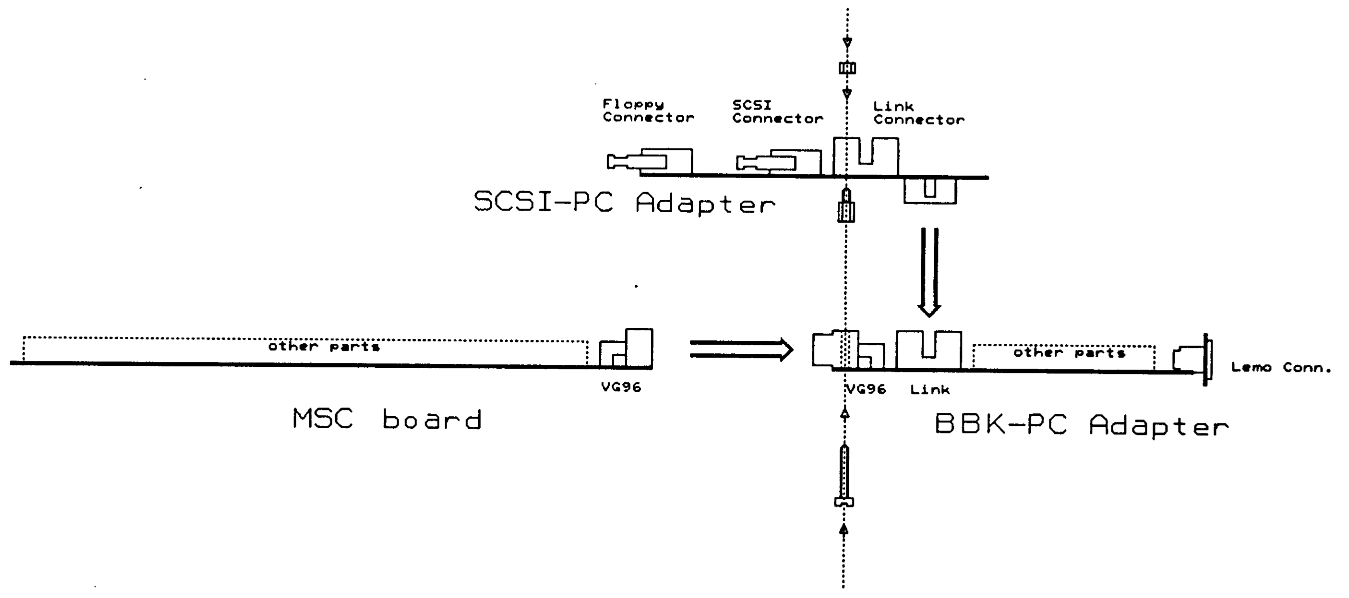
1.1 PC users

- Mount the MSC, the BBK-PC and the SCSI-PC adapter (if not already done).
- Make sure, the BBK-PC is correctly set up for operation with busless transputer modules under MULTITOOL. No other adapters in the PC should interfere with the BBK-Q adresses.
- The link speed of MSC and BBK-PC must be the same. Connect link 0 of the MSC (which is the top link connector on th SCSI-PC adapter) with the BBK-PC link via a flat link cable.
- Connect the 50-pin SCSI bus flat cable with one end to the SCSI connector on the SCSI-PC adapter and the other end to the winchester. Note the orientation of pin 1. The winchester must have it's termination resistors installed in this configuration. The winchester SCSI address must be 0.
- Insert the BBK-PC into a free slot of your PC.
- Apply power to your PC and to the winchester.
- See "Software Installation".

1.2 Using the MULTICLUSTER backplane

- Connect a Ground and a 5 Volt power cable to the backplane.
- Insert the MSC into the backplane.
- Connect link 0 of the MSC (top link connector at the backplane) to the IOS-1 module or to the bus bridge head, you are using (BBK-PC, BBK-1, etc.). Make sure that the MSC and the bus bridge head at the other end operate at the same link speed. The bus bridge head must be set up to run MULTITOOL on the MSC transputer.
- Connect the 50-pin SCSI bus flat cable with one end to the SCSI connector on the backplane and the other end to the winchester. Note the orientation of pin 1. The winchester must have it's termination resistors installed in this configuration. The winchester SCSI address must be 0.
- Apply power to your host, to the MSC and to the winchester.
- See "Software Installation".

Mounting of the SCSI - PC - Adapter



PARSYTEC GmbH Juelicher Str. 338 D-5100 Aachen		
Title SCSI - PC - Adapter		
Size B	Document Number SCSIPCMO.ORC	REV 1.0
Date: May 9, 1989	Sheet 1 of 1	

1.3 Software installation

When the hardware is set up, you can start MULTITOOL on the MSC board.

PC users:
- Get the "File Utilities" of MULTITOOL.
- Create an empty fold where the msc.driver software shall reside.
- Insert the discette labeled "msc.driver" into drive A: .
- Start function "COPY IN" of the File Utilities by pressing ALT-8.
- Set the "DestinationFileName" of the "directory parameters" to "mscdrive".
- Press the "Exit Fold" - key: The msc.driver software will be copied into the new fold. Now see "Starting a test".

MULTITOOL users:
- The same as for PC users, except that the "COPY IN" is replaced by a "STREAM RETRIEVE".

Starting a test:

- After the copy is ok, enter the fold and get and run the EXE. A menu should appear.
- Execute command "Help".
- Execute command "Clear". Set the following parameters:

"Process ID"	to	0
"Priority"	to	0
"Number of winchesters"	to	1
"Number of floppies"	to	0
"Number of streamers"	to	0
"MSC SCSI address"	to	7

- Execute the command "Test". Answer the first question with blank.

Now the msc.driver initialises and loads the winchester using different block sizes as printed. "W" means writing some sectors, "R" means reading the written sectors and "v" means a compare of the written and read sectors. The test can be aborted by pressing any key.

MTM-PC : Software Controlled Configuration

The MTM-PC has software controlled link connections via C004 linkswitches. This allows to configure the processors and the external link connectors of the board automatically by Helios or the configurer/loader of MEGATOOL's Occam Utility Set before or during a session.

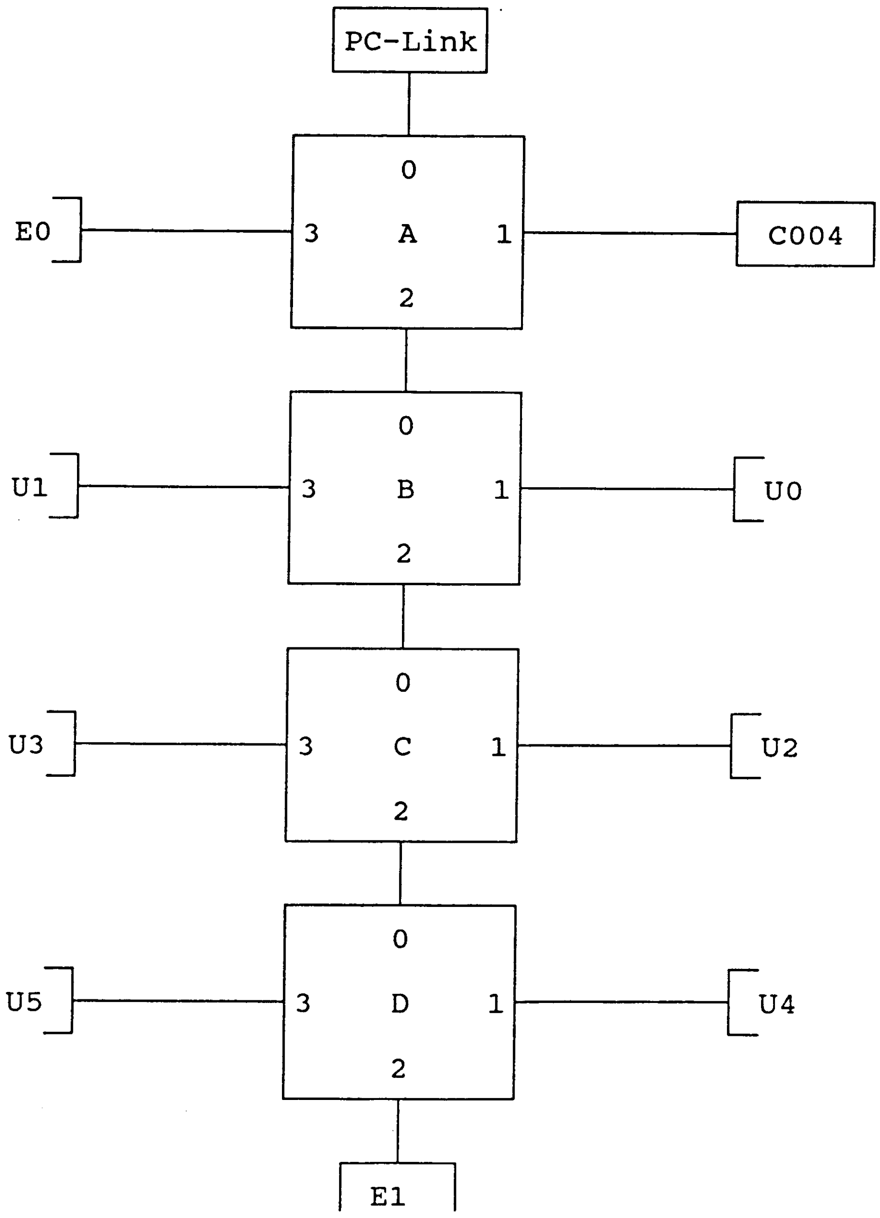
The file "MTM-PC.CDE" on the supplied disc contains the transputer code to configure the MTM-PC board with its C004 linkswitches from DOS level according to a default configuration structure as shown below. The program has been written in OCCAM under MEGATOOL and made bootable to be used with the alien file server (AFSERVER) which is an executable DOS program and supplied too. The program assumes the transputer A of the board connected via its link 0 to the PC-link section and via its link 1 to the configuration input of the C004 linkswitches (standard jumper configuration of the MTM-PC). To configure the MTM-PC board with the default configuration, the MTM-PC.CDE file has to be send to that transputer using the following command from DOS level :-

```
        AFSERVER -:b MTM-PC.CDE
```

This will reset the transputer and load the file MTM-PC.CDE into the processor. The program automatically starts and gives you some information about the configuration process on your screen. The alien file server used to boot the transputer is described in full in the server part of the MEGATOOL documentation. See below for a short description of how to invoke the AFSERVER.

Default configuration

The default configuration of the MTM-PC with the mentioned MTM-PC.CDE program builds a pipeline as follows :-



The notation of the processors and the connectors correspond to the technical documentation of the MTM-PC board.

Note that the connections between transputer A and the PC-Link as well as the C004 configuration input are hardwired by use of the appropriate jumpers JP21 and JP22 on the MTM-PC module.

Alien file server

The program to boot and load the transputer is the alien file server (AFSERVER) which is an executable DOS program and part of the MEGATOOL release for MS-DOS/PC-DOS systems. It's full description can be found in the server part of the MEGATOOL documentation. In this section only the necessary options for invoking the alien file server for configuring the MTM-PC will be mentioned.

The syntax of the alien file server call is as follows :-

```
              AFSERVER [command.line]
```

where command.line is defined as follows :-

```
      command.line          =    option
                            |    option command.line

      option                =    - options
                            |    / options

      options               =    :b boot.file.name
                            |    :l [#]link.address
                            |    :i

      boot.file.name        =    standard DOS file name

      link.address          =    decimal or hexadecimal number
```

Boot Transputer option (:b)

If the option ':b' is used the server will try and use the given file name after the ':b' to boot the transputer with. If the file name is not a valid file or the server is unable to load the file, appropriate error messages are given. When the server boots up, the transputer is reset. If this option is not specified the server will try and communicate with a program that has been previously loaded onto the transputer. If no program is loaded on the transputer, the server will hang-up. This is because the server does not test the board to see if a program is resident.

Link Address option (:l)

The use of the ':l' option enables you to change the address which the aerevr uses to communicate with the transputer board. If a '#' is used as a prefix of the following number then the number is taken as a hexadecimal number. If no number is specified an error will occur.
The option need only be used to change the link address of the board to other addresses than the default ones, as the server defaults to #150 for use with AT-like systems or #300 on XT's.

Server Information option (:i)

If the option ':i' is used the server will display a copyright message and it's version date.

Installation

The supplied disc (360 Kb, IBM-PC format) contains the software in copy format. To install the programs on your harddisk you can copy it back to any directory you want. Here we assume a subdirectory called 'MTM-PC' which is directly below the root directory. To create the subdirectory turn to the root directory of your system and issue at DOS level :-

```
MD \MTM-PC
```

Then enter the following command to copy all the files from the disc to that subdirectory :-

```
COPY A:*.* \MTM-PC
```

To configure the MTM-PC board from any directory the following command can be used :-

```
\MTM-PC\AFSERVER -:b \MTM-PC\MTM-PC.CDE
```

If you have installed the programs in other directories change the appropriate pathnames in the above command.
See the AFSERVER description above for other sometimes necessary options like setting the linkaddress.
→

MTM-PC : Software Controlled Configuration

The MTM-PC has software controlled link connections via C004 linkswitches. This allows to configure the processors and the external link connectors of the board automatically by Helios or the configurer/loader of MEGATOOL's Occam Utility Set before or during a session.

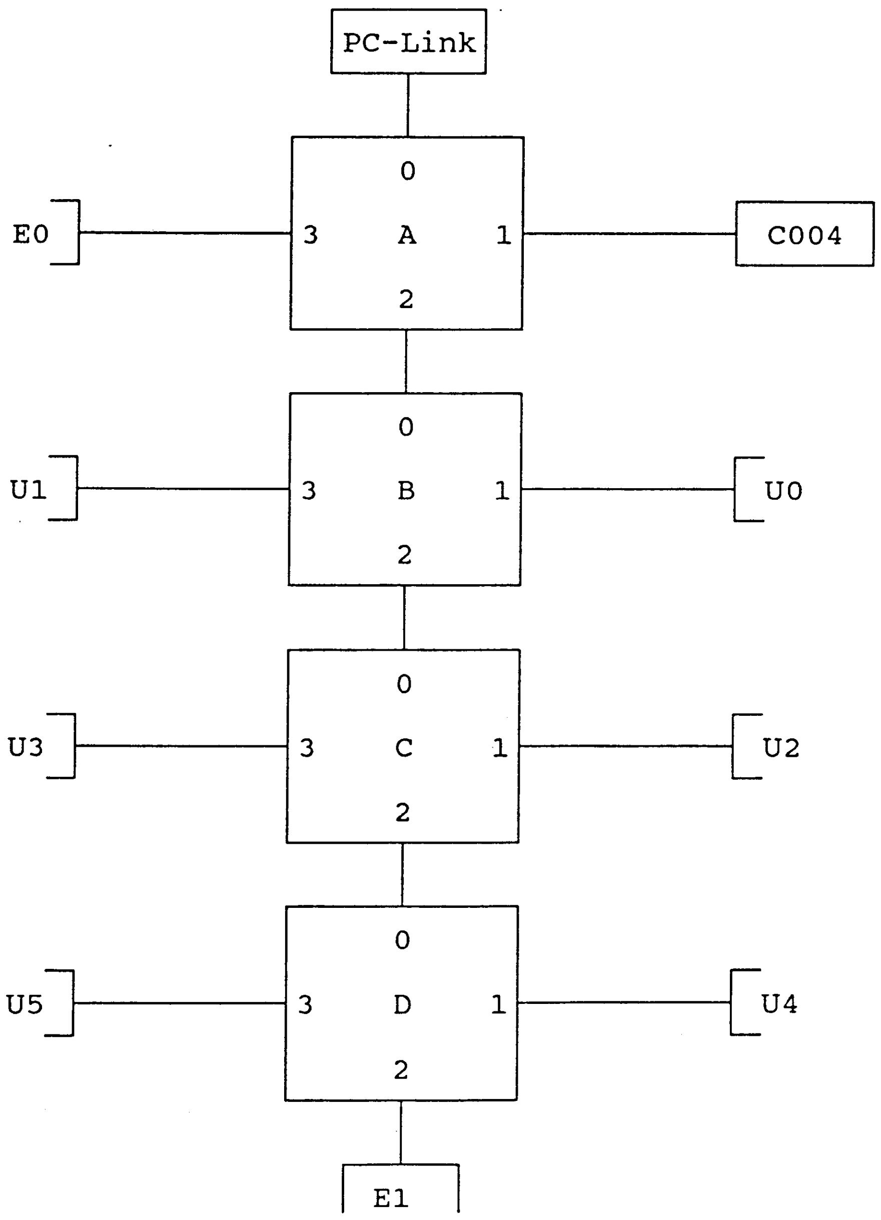
The file "MTM-PC.CDE" on the supplied disc contains the transputer code to configure the MTM-PC board with its C004 linkswitches from DOS level according to a default configuration structure as shown below. The program has been written in OCCAM under MEGATOOL and made bootable to be used with the alien file server (AFSERVER) which is an executable DOS program and supplied too. The program assumes the transputer A of the board connected via its link 0 to the PC-link section and via its link 1 to the configuration input of the C004 linkswitches (standard jumper configuration of the MTM-PC). To configure the MTM-PC board with the default configuration, the MTM-PC.CDE file has to be send to that transputer using the following command from DOS level :-

```
        AFSERVER -:b MTM-PC.CDE
```

This will reset the transputer and load the file MTM-PC.CDE into the processor. The program automatically starts and gives you some information about the configuration process on your screen. The alien file server used to boot the transputer is described in full in the server part of the MEGATOOL documentation. See below for a short description of how to invoke the AFSERVER.

Default configuration

The default configuration of the MTM-PC with the mentioned MTM-PC.CDE program builds a pipeline as follows :-



The notation of the processors and the connectors correspond to the technical documentation of the MTM-PC board.

Note that the connections between transputer A and the PC-Link as well as the C004 configuration input are hardwired by use of the appropriate jumpers JP21 and JP22 on the MTM-PC module.

Alien file server

The program to boot and load the transputer is the alien file server (AFSERVER) which is an executable DOS program and part of the MEGATOOL release for MS-DOS/PC-DOS systems. It's full description can be found in the server part of the MEGATOOL documentation. In this section only the necessary options for invoking the alien file server for configuring the MTM-PC will be mentioned.

The syntax of the alien file server call is as follows :-

```
              AFSERVER [command.line]
```

where command.line is defined as follows :-

```
      command.line          =    option
                              |  option command.line

      option                =    - options
                              |  / options

      options               =    :b boot.file.name
                              |  :l [#]link.address
                              |  :i

      boot.file.name        =    standard DOS file name

      link.address          =    decimal or hexadecimal number
```

Boot Transputer option (:b)

If the option ':b' is used the server will try and use the given file name after the ':b' to boot the transputer with. If the file name is not a valid file or the server is unable to load the file, appropriate error messages are given. When the server boots up, the transputer is reset. If this option is not specified the server will try and communicate with a program that has been previously loaded onto the transputer. If no program is loaded on the transputer, the server will hang-up. This is because the server does not test the board to see if a program is resident.

Link Address option (:l)

The use of the ':l' option enables you to change the address which the aerevr uses to communicate with the transputer board. If a '#' is used as a prefix of the following number then the number is taken as a hexadecimal number. If no number is specified an error will occur.
The option need only be used to change the link address of the board to other addresses than the default ones, as the server defaults to #150 for use with AT-like systems or #300 on XT's.

Server Information option (:i)

If the option ':i' is used the server will display a copyright message and it's version date.

Installation

The supplied disc (360 Kb, IBM-PC format) contains the software in copy format. To install the programs on your harddisk you can copy it back to any directory you want. Here we assume a subdirectory called 'MTM-PC' which is directly below the root directory. To create the subdirectory turn to the root directory of your system and issue at DOS level :-

```
        MD \MTM-PC
```

Then enter the following command to copy all the files from the disc to that subdirectory :-

```
        COPY A:*.* \MTM-PC
```

To configure the MTM-PC board from any directory the following command can be used :-

```
        \MTM-PC\AFSERVER -:b \MTM-PC\MTM-PC.CDE
```

If you have installed the programs in other directories change the appropriate pathnames in the above command.
See the AFSERVER description above for other sometimes necessary options like setting the linkaddress.
→