apeNEXT operating system OS7

01 June 2005

Abstract

This document describes the services provided by the apeNEXT operating system OS7, the routines running on the apeNEXT nodes, and the protocol for communication with the host operating system nose. All high-level services are based on a small set of low-level data transport operations, which are provided by the *transport layer*. The details of the high-level services are described in the section *protocol layer*.

Contents

List of Tables

1 Terminology

- In this document a “word” always refers to 128 bit.

2 Transport Layer

2.1 Low-level data transport operations

Data between the host interface boards (HIB) attached to the front-end systems and the host system can either be transported via the I2C channels[1] or the LVDS links[2] (so-called 7th links). The latter case might also require transport operations within the host system since only one or a few nodes are connected to the HIB(s) via LVDS links. These nodes are referred to as *HIB nodes*. If more than one HIB node exists within a given host partition, one of them is the *master HIB node*.

HIB nodes

2.2 Transport via the LVDS links

Communication between HIB nodes and the HIB proceeds along the following steps:

- HIB node sends message of length 1 word which contains the *7th-link header*.
- HIB decodes relevant parts of the header (i.e. `hib_cmd` and `hib_len`) to retrieve information about the following communication.
- HIB prepares for reading/sending message.
- When communication finished the HIB will become ready to receive the next header.

7th-link header

Note that the machine is not switched into I2C-mode for transport operations via the LVDS links.

The following transport operations via the 7th-link are available:

Slice write (SWR): Private data from a slice of nodes is written to the HIB(s). This operation requires when using the LVDS links the data from all nodes within the specified slice being moved to the HIB node(s).

Global write (GWR): Global data[3] is written directly from the master HIB node to the corresponding HIB. This operation does not involve any transport operation within the host system.

Multi-data read (MRD): Private data is read from the HIB(s) to a slice of nodes. When using the LVDS links the data for all nodes needs to be send to the HIB node(s). From there the data is moved to all nodes within the specified slice.

Broadcast read (BRD): Data is read from the HIB(s) and broadcasted to all nodes within the specified slice.

Broadcast load (BLD): Data is read from the HIB(s) and broadcasted to all nodes (independent of slice).[4]

[1] Here the front-end acts as master.
[2] Here the host system acts as master.
[3] It is assumed that an identical copy of this data exists on all nodes.
[4] This operation is handled by separate routines and is mentioned here only for completeness.

All transport operations act on 1 or more words. Flags on the protocol level might control use or initialisation of high or low part of all words of one particular transport operation.

The header used for communication between HIB nodes and the HIB contains the following fields:

`hib_cmd`: The command field encodes the following commands to the HIB:

Table 2.1: *List of HIB commands (`hib_cmd`).*

0x0	NOP
0x2	read
0x3	write
0x7	write 64 words
0x8	reset

`hib_len`: The length field encodes the number of words transmitted during the following communication. In this document length is always understood as length per node.[5] Each node can contain up to 256 MBytes of memory, therefore the maximum length is 2^{24} words.

`sys_svnop`: Combining the information from this field and the `hib_cmd` defines the transport operation being used.

Table 2.2: *`sys_svnop` bits.*

sys_svnop	hib_cmd	
0x1	0x2	SVNOP_BRD
0x1	0x3	SVNOP_GWR
0x2	0x2	SVNOP_MRD
0x2	0x3	SVNOP_SWR

`sys_svntag`: As the HIB is in case of transmissions via the 7th-link not able to handle burst of length larger than 64 words, transport layer operations have to be translated into several transmissions from node to HIB and vice versa. Only the first burst will contain information on the full length. The `sys_svntag` is used to mark the first and last burst being transmitted.

Table 2.3: *`sys_svntag` bits.*

0x0	intermediate transmission
0x1	first transmission
0x2	final transmission
0x3	first and final transmission

`sys_req`: The request field bits trigger a particular protocol layer service. See Tab. 3.1 for encoding.

`sys_rval`: The bits of this field control the handling of the return value on the front-end system.

Table 2.4: *`sys_rval` bits.*

0x1	RVAL_GET	fetch return value
0x2	RVAL_RST	reset return value to zero

[5] Depending on the type of operation the total amount of words transmitted during a data transport operation has to be multiplied with the number of nodes within a particular slice.

`sys_arg`: The system argument field provides optional arguments for a requested service.

`sys_rev`: The system revision field encodes the current version of the header encoding.

Table 2.5: *7th-link header encoding.*

3: 0	0000 0000 0000 0000 0000 0000 0000 000F	hib_cmd
27: 4	0000 0000 0000 0000 0000 0000 0FFF FFF0	hib_len
29: 28	0000 0000 0000 0000 0000 0000 3000 0000	sys_svnop
31: 30	0000 0000 0000 0000 0000 0000 C000 0000	sys_svntag
37: 32	0000 0000 0000 0000 0000 003F 0000 0000	sys_req
39: 38	0000 0000 0000 0000 0000 00C0 0000 0000	sys_rval
125: 40	3FFF FFFF FFFF FFFF FFFF FF00 0000 0000	sys_arg
127:126	C000 0000 0000 0000 0000 0000 0000 0000	sys_rev

Comments:

- Only the fields `hib_cmd` and `hib_len` are used by the HIB hardware itself. All other fields should be ignored by the hardware.
- The header is written to all HIBs, even if some of them will not be involved in the data transport.[6]

2.3 Transport via the I2C links

System services via the I2C links will proceed along the following steps:

- The `sys_arg` is written into the I2C-argument register (`REG_SYS_ARG`).
- The machine, i.e. all nodes, are stopped using the following halt code:

Table 2.6: *I2C-halt encoding.*

0x3F	sys_req
0xC0	sys_rval

- The front-end system reads `sys_req`, `sys_rval` and `sys_arg` from a configuration register and the register file, respectively.
- After completion of the system service the front-end system puts the machine back into run mode (if necessary).

Table 2.7: *I2C-argument register encoding.*

123: 0	0FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF	sys_arg
127:126	C000 0000 0000 0000 0000 0000 0000 0000	sys_rev

2.4 Transport layer routines

Before switching from protocol layer to the transport layer the following set of arguments are written to a memory area starting at `MEM_OS7_TPARG`. The arguments and their order are shown in Tab. 2.8.

When all arguments have been written to memory the transport layer routines are invoked in the following way:

[6] This is required in order to ensure synchronisation of I/O operations.

t

Table 2.8: *Transport layer arguments.*

MEM_OS7_TPARG+0	Number of transport operations
MEM_OS7_TPARG+1	Lower/upper limit of slice in x-direction
MEM_OS7_TPARG+2	Lower/upper limit of slice in y-direction
MEM_OS7_TPARG+3	Lower/upper limit of slice in z-direction
	transport operation #0
MEM_OS7_TPARG+4	Pointer to transport routine
MEM_OS7_TPARG+5	Header (sys_rev|sys_arg|sys_req|sys_svnop)
MEM_OS7_TPARG+6	Data source/destination memory address
MEM_OS7_TPARG+7	Data length
MEM_OS7_TPARG+8	Slice flag (full slice / user defined slice)
MEM_OS7_TPARG+9	I2C request code
	transport operation #1
MEM_OS7_TPARG+10	Pointer to transport routine
...	

- Execute transport startup routine
- For-each transport layer operation:
  - Load arguments
  - Call transport layer routine
- Execute transport finish routine

2.4.1 Transport startup and finish routines

Transport layer operations typically require a number of startup and finish operations like:

- Spilling and restoring registers.
- Switching from user to system network topology and vice versa.[7]

The following routines are defined as macros:

- `$os7_i2cstart`
- `$os7_svnstart`
- `$os7_i2cend`
- `$os7_svnend`

[7] Note that this is not required for GWR.

3 Protocol Layer

This chapter describes the details of all services provided by OS7 and the corresponding assembly macros.

Note that during one system service it only possible to use either the I2C links or the 7th links, even if the transport operations for this service could be done via either of these links.

A group of system services which have a particular argument encoding in common and use the same transport link type are grouped into so-called requests. Different services using different transport link types might use the same request code.

3.1 Overview on system requests

The following table lists all system requests and the corresponding request codes and transport link types:

Table 3.1: *System service request codes.*

Request	I2C	7th	description
REQ_NOP	0x00	0x00	no operation
REQ_EXIT	0x01	–	program exit
REQ_IO	0x02	0x02	basic I/O operations
REQ_FSE	0x03	0x03	file seek operation
REQ_SET	0x04	0x04	set system variable
REQ_PARGS	0x05	0x05	read command line arguments
REQ_RVAL	0x06	0x06	return value handling

3.2 Caveats about using protocol layer routines

All source and destination addresses provided to the macros have either to be known at compile-time or computable by a MOP at run-time. It is, e.g., not allowed to use the address of a virtual register.

By default OS7 uses virtual registers for the protocol layer routines.[1] Special care has to be taken if code is inserted inside a composition of OS7 macros, i.e. I/O packet lists started by $io_start and completed by $io_end. Within I/O packets loops might be used after setting $io_setloopflg. In this case counters are stored in memory.

3.3 Return value handling

System services may consist of several protocol level operations. In these cases the return value is composed at the front-end system and reset on request. The host has to always explicitely request returning the return value.

The return value consists of one word where high and low bank contain different information:

[1] When switching to transport layer a spilling of all user registers is performed. For the transport layer routines physical registers are used.

- low bank: status information (zero if successful)
- high bank: additional information (depends on operation)

The return value is stored at the memory address `MEM_SYS_RVAL`. Depending on whether the internal error handler is enabled (see `$sys_errhandler_enable`, `$sys_errhandler_disable`).

3.4 Exit request (REQ_EXIT)

Table 3.2: *sys_arg encoding for sys_req==REQ_EXIT.*

status	15:0	0000 0000 0000 0000 0000 0000 0000 00FF

3.4.1 User program exit service

Description: This service handles end of program execution. When the machine stopped with exit code `sys_req==REQ_EXIT`. The user defined exit code has to be read via I2C from the `REG_SYS_ARG` register.

Assembler macro: `$sys_exit()`

3.5 I/O operations request (REQ_IO)

Table 3.3: *sys_arg encoding for for sys_req==REQ_IO.*

IOADDR[2]	23: 0	0000 0000 0000 0000 0000 0000 00FF FFFF
IOLEN[2]	37: 24	0000 0000 0000 0000 0000 003F FF00 0000
IODEV[2]	39: 38	0000 0000 0000 0000 0000 00C0 0000 0000
IOCMD	47: 40	0000 0000 0000 0000 0000 FF00 0000 0000
IOFD	55: 48	0000 0000 0000 0000 00FF 0000 0000 0000
IOBS	57: 56	0000 0000 0000 0000 0300 0000 0000 0000
IORDFLG	59: 58	0000 0000 0000 0000 0C00 0000 0000 0000
IOMODE	62: 60	0000 0000 0000 0000 7000 0000 0000 0000
IOSLICE	95: 64	0000 0000 FFFF FFFF 0000 0000 0000 0000
IOFMT	99: 96	0000 000F 0000 0000 0000 0000 0000 0000
IOBSEP	103:100	0000 00F0 0000 0000 0000 0000 0000 0000
IOWSEP	107:104	0000 0F00 0000 0000 0000 0000 0000 0000
IOPSEP	111:108	0000 F000 0000 0000 0000 0000 0000 0000
IOCNT	119:112	00FF 0000 0000 0000 0000 0000 0000 0000

IOADDR: Source or destination address used for I2C operations. When using 7th-link this field must be zero.

IOLEN: Number of words to be read or written via I2C link. When using 7th-link this field must be zero.

IODEV: Defines the device from/to which data is to be read/written. Two devices are supported: memory and register file. When using 7th-link this field must be zero.

Table 3.4: *List of I/O devices.*

0x0	IODEV_MEM
0x1	IODEV_RF

IOCMD: I/O command code encodes the commands listed in Tab. 3.5. Details on

[2] Only when using I2C-link.

each of the commands is described in the following (sub-)sections.

Table 3.5: *List of I/O commands.*

0x00	IOCMD_NOP
0x01	IOCMD_SWR
0x02	IOCMD_GWR
0x03	IOCMD_MRD
0x04	IOCMD_BRD
0x05	IOCMD_FOP
0x06	IOCMD_FCL
0x07	IOCMD_TIM
0x09	IOCMD_PRF
0x0a	IOCMD_SCF
0x0b	IOCMD_MRA
0x0c	IOCMD_BRA
0x0d	IOCMD_SPRF
0x0e	IOCMD_MSCF
0x0f	IOCMD_SYSTEM

IOFD: File descriptor is a small, non-negative integer 0...255. Reserved file descriptors are shown in the following table:

Table 3.6: *List of reserved file descriptors.*

0x00	IOFD_STDIN
0x01	IOFD_STDOUT
0x02	IOFD_STDERR

IOBS: Selects which part of the word will be used. Possible values are listed in the following table:

Table 3.7: *Bank selector bits.*

0x1	IOBS_L
0x2	IOBS_H
0x3	IOBS_V

IORDFLG: Defines flags needed for initialisation of high bank during read operations:

Table 3.8: *List of read flags.*

0x0	IORDFLG_NOP	high bank unchanged
0x1	IORDFLG_HZERO	initialise high bank with zero
0x2	IORDFLG_HMIRROR	mirror content low bank to high bank

IOMODE: Modes used for file open operation as listed in Tab. 3.9. Ignored unless IOCMD==IOCMD_FOP.

Table 3.9: *List of file open modes with the corresponding argument used by `fopen()`.*

0x1	IOMODE_R	r
0x2	IOMODE_RPLUS	r+
0x3	IOMODE_W	w
0x4	IOMODE_WPLUS	w+
0x5	IOMODE_A	a
0x6	IOMODE_APLUS	a+

IOSLICE: The topology of any apeNEXT machine consisting of at least 256 nodes is $4 \cdot n \times 8 \times 8$, where n is the number of crates. Any slice $[k_x, k_y, k_z][l_x, l_y, l_z]$ can therefore be encoded in $2 \times (8 + 4 + 4) = 32$ bits. The special value IOSLICE_ALL is used to select all nodes of the machine. This field is ignored for IOCMD=IOCMD_GWR.

IOFMT: Format of the data to be written/read. In case of a formatted write operation IOFMT is translated into a standard C format as indicated in the last column of Tab. 3.10. In case of a formatted read operation IOFMT is used to convert numbers which have been identified by pattern matching.[3]

Table 3.10: *List of formats and the corresponding standard C formats (last column).*

0x0	IOFMT_BIN	binary (=unformatted)	
0x1	IOFMT_INT	integer	%d
0x2	IOFMT_UINT	unsigned integer	%u
0x3	IOFMT_HEX	hexadecimal integer	0x%x
0x4	IOFMT_STR	string	%s
0x5	IOFMT_VARSTR	variable string	%s
0x6	IOFMT_DBL	double	%.15e

IOBSEP: Defines token which is written between high and low part of a word. This field is ignored unless IOBS==IOBS_V.

Table 3.11: *List of separators.*

0x00	IO[BWP]SEP_NOP	”
0x01	IO[BWP]SEP_SPACE	’ ’
0x02	IO[BWP]SEP_TAB	\t
0x03	IO[BWP]SEP_NL	\n
0x04	IO[BWP]SEP_KOMMA	,
0x08	IOBSEP_RBRACKET[4]	(#,#)
0x09	IOBSEP_SBRACKET[4]	[#,#]

IOWSEP: Defines token which is written between different words. It is not printed after writing the last word. A list of all separators is listed in Tab. 3.11.

IOPSEP: Defines token which is written at the end of the packet, irrespective of whether there is a next packet. A list of all separators is listed in Tab. 3.11.

IOCNT: Counts the number of packets in case of composed commands. Counter will be decremented for consecutive operations and refers to the number of transport operations still to follow.

3.5.1 I/O packet and I/O packet list

Description: An *I/O packet* defines the elementary protocol level I/O operation. Details of the available commands are described below.

Any I/O operation defined in the high-level language user program usually translates into several I/O packets which are grouped into one *I/O packet list*. E.g. there may be various write operations and a final read operations to obtain the return value.

[3] All non-matching characters are ignored.
[4] Only usable as bank separator.

Assembler macros: `$io_start`, `$io_packet()`, `$io_end`, `$io_fd`, `$io_cnd_set()`, `$io_cnd_decr`, `$io_cnd_clear`

Notes: An I/O packet list may not be empty.

3.5.1.1 Slice write command (IOCMD_SWR)

Description: Write slice of data using the format defined by `IOFMT`, `IOBSEP`, `IOWSEP` and `IOPSEP`.

Return value:

status	number
0 = success	number of words written
1 = failure	number of words written

3.5.1.2 Global write command (IOCMD_GWR)

Description: Write global data using the format defined by `IOFMT`, `IOBSEP`, `IOWSEP` and `IOPSEP`. As the data is assumed to be global, the data is written from just one node.

Return value:

status	number
0 = success	number of words written
1 = failure	number of words written

3.5.1.3 Multi-data read command (IOCMD_MRD)

Description: Read private data to a slice of nodes. A binary read is performed when `IOFMT=IOFMT_BIN`. Otherwise a formatted read is done with data being converted according to `IOFMT`. During a formatted read operation the following characters are considered as separators: '`(`', '`)`', '`[`', '`]`', '`,`', '`\t`', '`\n`'. Initialisation of the high bank depends on the setting of `IORDFLG`.

Return value:

status	number
0 = success	number of words read
1 = failure	number of words read

3.5.1.4 Broadcast read command (IOCMD_BRD)

Description: Same as `IOCMD_MRD`, but the same data is broadcasted to all nodes of the specified slice.

Return value:

status	number
0 = success	number of words read
1 = failure	number of words read

3.5.1.5 Read time command (IOCMD_TIM)

Description: Read the number of non-leap seconds since the epoch (i.e. 00:00:00 UTC, January 1, 1970) using a broadcast read operation.

Return value:

status	number
0 = success	number of words read
1 = failure	number of words read

3.5.1.6 Multi-data read of random numbers (IOCMD_MRA)

Description: Read random numbers from the front-end system using a multi-data read operation. The random numbers are generated using a simple, 64-bit linear congruential generator. If `fmt` = `SYS_IOFMT_DBL` floating point

numbers in the range $[0.0, 1.0]$ will be returned, otherwise unsigned 64-bit fix point numbers are read.

The seed can be set using the request `REQ_SET`.

Return value:

`status`	`number`
0 = success	number of words read
1 = failure	number of words read

3.5.1.7 Broadcast read of random numbers (`IOCMD_BRA`)

Description: Same as `IOCMD_MRA`, but a broadcast read operation is used instead.

Return value:

`status`	`number`
0 = success	number of words read
1 = failure	number of words read

3.5.1.8 C-printf command (`IOCMD_PRF`)

Description: Send next argument of a `printf()` function to the front-end system using a global write operation. The first I/O packet has to provide the total number of arguments in `IOCNT`. Before each following packet this counter has to be decremented.

Return value:

`status`	`number`
0 = success	number of bytes written
1 = failure	number of bytes written

3.5.1.9 Slice C-printf command (`IOCMD_SPRF`)

Description: Same as `IOCMD_PRF` but, a slice write operation is used.

Return value:

`status`	`number`
0 = success	number of bytes written
1 = failure	number of bytes written

3.5.1.10 C-scanf command (`IOCMD_SCF`)

Description: Send next argument of a `scanf()` function to the front-end system and read requested value by a broadcast read operation to all nodes. The first I/O packet has to provide the total number of arguments in `IOCNT`. Before each following packet this counter has to be decremented.

Return value:

`status`	`number`
0 = success	number of items read
1 = failure	number of items read

3.5.1.11 Multi-data C-scanf command (`IOCMD_MSCF`)

Description: Same as `IOCMD_MSCF`, but a multi-data read is used instead.

Return value:

`status`	`number`
0 = success	number of items read
1 = failure	number of items read

3.5.2 File open command (`IOCMD_FOP`)

Description: Open a file with a given path and mode and file descriptor. Predefined file descriptors are listed in Tab. 3.6. The available file modes are defined in Tab. 3.9.

Assembler macros: `$io_fopen()`, `$io_fopenp()`, `$io_setmode()`

Return value:

status	number
0 = success	–
1 = failure nose	–
2 = failure fopen	errno returned by fopen() on front-end

3.5.3 File close command (IOCMD_FCL)

Description: Close a file with a given file descriptor.

Assembler macros: `$io_fclose()`

Return value:

status	number
0 = success	–
1 = failure nose	–
2 = failure fclose	errno returned by fclose() on front-end

3.5.4 System command (IOCMD_SYSTEM)

Description: Perform a given command in a shell on the front-end system. The command string is written to the front-end system using a global write operation.

Assembler macros: `$io_system()`, `$io_packet()`

Return value:

status	number
0 = success	status returned by system() on front-end
1 = failure nose	–

3.6 File seek request (REQ_FSE)

Table 3.12: *sys_arg encoding for sys_req==REQ_FSE.*

FSE_FD	55:48	0000 0000 0000 0000 00FF 0000 0000 0000
FSE_WHENCE	57:56	0000 0000 0000 0000 0300 0000 0000 0000
FSE_OFFSET	127:64	FFFF FFFF FFFF FFFF 0000 0000 0000 0000

3.6.1 File seek command

Description: This command allows to do a file seek operation on a file previously opened with file descriptor FSE_FD. The new file position is obtained by adding FSE_OFFSET bytes to the position specified by FSE_WHENCE. FSE_WHENCE can have the following values: FSE_SET, FSE_CUR, or FSE_END.

Assembler macro: `$sys_fseek()`

Return value:

status	number
0 = success	–
1 = failure	–

3.7 Set variable request (REQ_SET)

Table 3.13: *sys_arg encoding for sys_req==REQ_SET.*

IOADDR[2]	23:0	0000 0000 0000 0000 0000 0000 00FF FFFF
IODEV[2]	39:38	0000 0000 0000 0000 0000 00C0 0000 0000
SET_VAR	63:48	0000 0000 0000 0000 FFFF 0000 0000 0000

IOADDR: See section "I/O Operations Request".

IODEV: See section "I/O Operations Request".

SET_VAR: The following variables are encoded in this field:

Table 3.14: *List of system variables which can be modified through REQ_SET*

0x0001	SETVAR_SEED	Random number generator seed

For this request no return values are defined. Program execution is aborted in case of a failure, e.g. when setting of an unknown variable is attempted.

3.7.1 Set random number command

Description: Two 64-bit unsigned integers are communicated to the front-end system for being used as new seed for the random number generator.

Assembler macro: `$sys_srand()`

3.8 Run-time program arguments request (REQ_PARGS)

Table 3.15: *sys_arg encoding for sys_req==REQ_PARGS.*

IOADDR	23:0	0000 0000 0000 0000 0000 0000 00FF FFFF

Description: This service allows to fetch command line arguments to a given memory address. The front-end system will load starting at the given memory address the following items:

- Number of arguments
- Total number of words stored to memory
- Pointer to first argument
- ...
- Pointer to last argument
- First argument terminated by zero word
- ...
- Last argument terminated by zero word

Notes: OS7 does not check whether a sufficient amount of memory is available.

Currently this request will always be handled via the I2C-links.

Assembler macro: `$sys_pargs()`

3.9 Return value request (REQ_RVAL)

This request is used when a separate protocol layer operation is required to fetch the return value. No additional arguments are required except `sys_rval`.

4 Protocol Layer Assembly Macros

4.1 $io_cnt_clear

Request: REQ_IO

Description: Indicate end of using I/O packet counter within a I/O packet list.

$io_cnt_clear

4.2 $io_cnt_decr

Request: REQ_IO

Description: Decrement the I/O packet counter. Use of this macro has to be preceeded by a use of $io_cnt_set().

$io_cnt_decr

4.3 $io_cnt_set()

Request: REQ_IO

Description: Copy the initial value of the counter to a OS7 internal location.

$io_cnt_set(reg)		
reg	REG	Input register

4.4 $io_end

Request: REQ_IO

Description: Mark the end of an I/O packet list. See also $io_start.

$io_end

4.5 $io_fclose()

Request: REQ_IO

Description: Close file with given file descriptor.

$io_fclose(fd)		
fd	MOP	File descriptor

4.6 $io_fd()

Request: REQ_IO

Description: Set the file descriptor for the following I/O packets.

$io_fd(fd)		
fd	MOP	file descriptor

4.7 $io_fopen()

Request: REQ_IO

Description: Open file using file name, open modes and file descriptor provided on input. This macro implies use of $io_fd().

$io_fopen(fd, mode, fname)		
fd	MOP	File descriptor
mode	CONST	File open modes (generic encoding)
fname	CONST	File name (string)

4.8 $io_fopenp()

Request: REQ_IO

Description: Open file using file name stored at given location, open modes and file descriptor provided on input. The argument dev is for backward compatibility with os1. The argument len allows to provide the length of the file name in units of words. If the length of this expape variable is zero then code will be inserted to determine the length at run-time.

This macro implies use of $io_fd().

$io_fopenp(fd, mode, fnptr, dev, len)		
fd	MOP	File descriptor
mode	REG	Register containing open mode
fnptr	MOP	Pointer to file name in memory
dev	CONST	Must be equal to $SYS_IODEV_MEM
len	CONST—void	Length or empty

Notes: Register mode has to contain the modes using OS7 encoding. The macro $io_setmode() should be used to initialise this register.

4.9 $io_getd(), $io_getdv(), $io_gete(), $io_getz()

Request: REQ_IO

Description: Read a word as an integer, vector integer, double, or complex number from the front-end system. If used outside a I/O packet list, an I/O packet list is generated, i.e. wrapping these macros by $io_start and $io_end is not required.

When using these macros outside an explicit I/O packet list, a broadcast read is performed. If macros are used within an I/O packet list and a slice has been defined then a slice write operation is performed.

When using these macros within an explicit I/O packet list, a file descriptor may be specified using $io_fd().

$io_get*(maddr)		
maddr	MOP	Destination memory address

4.10 $io_node()

Request: REQ_IO

Description: Short-cut for $io_slice() when slice consists of one node only.

$io_node(x, y, z)		
x	MOP	Coordinate of node (x-direction)
y	MOP	Coordinate of node (y-direction)
z	MOP	Coordinate of node (z-direction)

4.11 $io_packet()

Request: REQ_IO

Description: Perform a I/O operations using the commands describe in section 3.5. One or more uses of $io_packet() have to be wrapped by $io_start and $io_end.

$io_packet(cmd,fmtsep,rdflg,dev,len,bs,addr)		
cmd	CONST	I/O command
fmtsep	CONST	Format and separators
rdflg	CONST	Read flag
dev	CONST	Source/destination device
len	MOP	Length
bs	CONST	Bank selector
addr	MOP|REG	Source/destination address

4.12 $io_puts()

Request: REQ_IO

Description: Write a string to the front-end system using a global write operation. If used outside a I/O packet list, an I/O packet list is generated, i.e. wrapping this macro by $io_start and $io_end is not required.

When using this macro within an explicit I/O packet list, a file descriptor may be specified using $io_fd().

$io_puts(str)		
str	CONST	Output string

4.13 $io_putd(), $io_putdv(), $io_putx(), $io_putxv(), $io_pute(), $io_putz()

Request: REQ_IO

Description: Write a word stored in memory as an decimal, vector decimal, hexadecimal, vector hexadecimal, double, or complex number to the front-end system. If used outside a I/O packet list, an I/O packet list is generated, i.e. wrapping these macros by $io_start and $io_end is not required.

When using these macros outside an explicit I/O packet list, a global write is performed. If macros are used within an I/O packet list and a slice has been defined then a slice write operation is performed.

When using these macros within an explicit I/O packet list, a file descriptor may be specified using $io_fd().

$io_put*(maddr)		
maddr	MOP	Source memory address

4.14 $io_rgetd(), $io_rgetdv(), $io_rgete(), $io_rgetz()

Request: REQ_IO

Description: Similar to $io_getd() etc., but word is written to the register file.

$io_rget*(reg)		
reg	CONST	Destination register address

4.15 $io_rputd(), $io_rputdv(), $io_rputx(), $io_rputxv(), $io_rpute(), $io_rputz()

Request: REQ_IO

Description: Similar to $io_putd() etc., but word is taken from register file.

$io_rput*(reg)		
reg	CONST	Source register address

4.16 $io_setloopflg

Request: REQ_IO

Description: When used the I/O macros will assume a loop in the code between $io_start and $io_end. If the loop flag is set the transport routine argument counter (TARGCNT) and IOCND (if used) will be stored to memory after each change. The loop flg is reset by $io_end.

$io_setloopflg

4.17 $io_setmode()

Description: Generates file open mode bits in register translating generic encoding used on input into OS7 encoding.

$io_setmode(reg, mode)		
reg	REG	Output data register
mode	MOP	File open mode (generic encoding)

4.18 $io_slice()

Request: REQ_IO

Description: Define slice for I/O operations. Within an I/O packet list the slice may only be defined once. Each I/O operation can use either this slice or the slice extending over the full partition. This is controlled by the macros $io_slice_set and $io_slice_all. The macro $io_slice() implies using $io_slice_set.

$io_slice(x0, y0, z0, x1, y1, z1)		
x0	MOP	Lower bound of slice (x-direction)
y0	MOP	Lower bound of slice (y-direction)
z0	MOP	Lower bound of slice (z-direction)
x1	MOP	Upper bound of slice (x-direction)
y1	MOP	Upper bound of slice (y-direction)
z1	MOP	Upper bound of slice (z-direction)

4.19 $io_slice_all

Request: REQ_IO

Description: When used the slice containing all nodes of the partition will be used for the following I/O packets.

$io_slice_all

4.20 $io_slice_set

Request: REQ_IO

Description: When used the slice defined by $io_slice() will be used for the following I/O packets.

$io_slice_set

4.21 $io_start

Request: REQ_IO

Description: Marks the start of an I/O packet list.

$io_start

4.22 $sys_errhandler_enable

Description: Enable the internal OS7 error handler. If error handler is enabled, a SEX will be raised when a system services ends with a return value status unequal to zero.

$sys_errhandler_enable

4.23 $sys_errhandler_disable

Description: Disable the internal OS7 error handler.

$sys_errhandler_disable

4.24 $sys_exit()

Request: REQ_EXIT

Description: End program execution using program exit status status.

$sys_exit(status)		
status	MOP	User program exit status

4.25 $sys_fseek()

Request: REQ_FSE

Description: Sets the file position indicator for the file pointed to by the file descriptor. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to FSEWHENCE_SET, FSEWHENCE_CUR, or FSEWHENCE_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

$sys_fseek(fd, offset, whence)		
fd	MOP	File descriptor
offset	MOP	Offset with respect to whence
whence	REG	Whence bits (generic encoding)

4.26 $sys_pargs()

Request: REQ_PARGS

Description: Loads the run-time program arguments to the given memory address.

$sys_pargs(addr)		
addr	MOP	Destination address

4.27 $sys_rtime()

Description: Obtain wall-clock time since last use of this macro in units of seconds as a double.

$sys_rtime(reg)		
reg	REG	Output register

4.28 $sys_setvar()

4.29 $sys_setlink()

Description: Define link to be used for all following transport operations.

$sys_setlink(hiblink)		
hiblink	CONST	$OS7_HIBLINK_SVN or $OS7_HIBLINK_I2C

4.30 $sys_setvar()

Request: REQ_SET

Description: Set a variable on the front-end system to a given value. The settable variables are defined in Tab. 3.14.

$sys_setvar(var, addr, dev)		
var	CONST	Variable identifier
addr	MOP	Source address
dev	CONST	Source device (MEM or RF)

Notes: The value has to fit in a single word.

4.31 $sys_setwhence()

Request: REQ_FSE

Description: Generate file seek whence bits in register using OS7 encoding. The constant provided on input has to contain the whence bits using generic encoding.

$sys_setwhence(reg, whence)		
reg	REG	Output data register
whence	MOP	Whence bits (generic encoding)

4.32 $sys_system()

Request: REQ_IO

Description: Perform the given command in a shell on the front-end system.

$sys_system(cmd)		
cmd	CONST	Command string

Notes: This macro will generate an I/O packet list. If the command string has

already been generated or is generated at run-time, an I/O packet list has to generated explicitely.

5 OS7 Configuration and Initialisation

5.1 Configuration

The following expape variables allow to change the behaviour of the OS7. Default values are defined in the file `os7_config`.

`$OS7_ASSERT`	If set to a non-zero value code making additional run-time checks will be inserted.
`$OS7_HIBLINK`	Defines the default link type (either $OS7_HIBLINK_SVN or $OS7_HIBLINK_I2C) used for data transport.
`$OS7_USEVREGS`	If set to a non-zero value code OS7 will (almost always) use virtual registers.

5.2 Initialisation

The following memory locations are expected to be initialised:

`MEM_HIB_FLG`	Defines whether node is connected to no HIB (`OS7_HIBFLG_NONE`), a slave HIB (`OS7_HIBFLG_SLAVE`), a master HIB (`OS7_HIBFLG_MASTER`)
`MEM_HIB_X`, `MEM_HIB_Y`, `MEM_HIB_Z`	Coordinates of (nearest) node attached to a HIB.
`MEM_SYS_USRNETCONF`, `MEM_SYS_SYSNETCONF`	User and system network configuration. The low and high bank should contain the setting for `CR_LINKCTRL` and `CR_COMCTRL`, respectively.
`MEM_OS7_HIBSLICEX`, `MEM_OS7_HIBSLICEY`, `MEM_OS7_HIBSLICEZ`	Defines the slice of nodes for which system service operations are performed via the node with coordinates (`MEM_HIB_X`,`MEM_HIB_Y`,`MEM_HIB_Z`).
`MEM_OS7_ROUTE_START` ...	Information needed to perform worm routing.