3.8.9 MM_PWRITE

NAME

mm_pwrite -- "Physical write"

SYNOPSIS

```
#include <memory.h>
uint mm_pwrite ( paddr, laddr, length )

        uint paddr;     /* physical start address */
        char *laddr;    /* logical start address */
        uint length;    /* length in bytes */
```

DESCRIPTION

The *mm_pwrite* directive reads from the logical address in the calling task's address space, and writes to a physical address. The length may not span a section boundary.

RETURN VALUE

If *mm_pwrite* was successful, then 0 is returned.

If the call was not successful, no data is transferred and an error code is returned.

ERROR CONDITIONS

Unmapped logical address.

Length spans section boundary.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.8.10 MM_PTCREATE

NAME

mm_ptcreate -- "Create a Logical Partition"

SYNOPSIS

```
#include <memory.h>
uint mm_ptcreate ( name, paddr, length, bsize, laddr, flags, &ptid, &bnum)

        uint name;      /* user defined 4-byte partition name */
        char *paddr;    /* physical start address of partition */
        uint length;    /* physical length in bytes */
        uint bsize;     /* size of buffers in bytes */
        char *laddr;    /* physical start address of partition */
        uint flags;     /* partition attributes */
        uint ptid;      /* partition id - returned by this call */
        uint bnum;      /* number of buffers in partition - returned by this call */
```

Flags field values:

GLOBAL	set	to indicate the partition is a multiprocessor global resource.
	clear	to indicate the partition is local

DESCRIPTION

This directive allows the user to create a logical partition of fixed size buffers from a contiguous memory area. The partition is mapped into the caller's address space at the logical address specified in *laddr*. By creating logical partitions at the same logical addresses, partitions can be easily shared between processors.

The partition id will be returned in *ptid* by the executive to use for *pt_getbuf* and *pt_retbuf* directives for the partition.

The partition physical start address must be on the pagesize boundary.

The number of buffers created by the executive will be returned in *bnum*. The executive may use memory within the partition for partition and buffer data structures. Therefore, the product of the buffer count and buffer size will be slightly less than the length of the partition.

By setting the GLOBAL value in the flags field, the *ptid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors.

The maximum number of partitions that may exist at any one time is a configuration parameter.

RETURN VALUE

If *mm_ptcreate* successfully created the partition, the *ptid* and *bnum* are filled in and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Too many partitions.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.9 Dual-ported Memory

Dual-ported memory is commonly found in multiprocessor systems. The executive provides a method for converting internal addresses to external, and external addresses to internal, to accommodate the use of dual-ported memory and allow tasks to exchange addresses between processors.

The *internal* address will be defined as the address of a memory resource, relative to the local node which needs to access the memory resource.

The *external* address will be defined as the address of a memory resource, relative to a remote node which needs to access the memory resource.

The directives provided for dual-ported memory are:

Directive	Function
m_ext2int	Convert external address
m_int2ext	Convert internal address

3.9.1 M_EXT2INT

NAME

m_ext2int -- "Convert external address to internal address"

SYNOPSIS

uint m_ext2int (external, &internal)

```
char *external;    /* external address */
char *internal;    /* internal address - returned by this call */
```

DESCRIPTION

The *m_ext2int* call is used to convert the physical address contained in *external* into an internal address, so it can be used by the local node. The internal address is returned to the caller in *internal.*

The external (VMEbus) address is normally an address received by the local node, and the requester may not know whether its internal (local) or not. If the address contained in *external* is internal, the returned address will be same as the address in *external.*

RETURN VALUE

The *m_ext2int* directive always succeeds, the internal address is returned in *internal,* and 0 is returned.

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

In a MMU system, a task will need to execute *mm_p2l* following this call to obtain a logical internal address.

3.9.2 M_INT2EXT

NAME

m_int2ext -- "Convert internal address to external address"

SYNOPSIS

uint m_int2ext (internal, &external)

```
char *internal;    /* internal address */
char *external;    /* external address - returned by this call */
```

DESCRIPTION

The *m_int2ext* call is used to convert the physical address contained in *internal* into an external address, so it can pass the address to a remote node within the system. The external address is returned to the requester in *external.*

The internal address is a physical address accessible by the local node within its dual-ported memory, and the external (VMEbus) address will be different.

RETURN VALUE

The *m_int2ext* directive always succeeds, the external address is returned in *external,* and 0 is returned.

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

In a MMU system, a task will need to execute *mm_l2p* preceding this call to obtain a physical address.

4. I/O INTERFACE

This section describes a set of I/O Interface services for the RTEID. These services provide a well defined mechanism for installing and calling device drivers. They provide a structured methodology for writing drivers which both simplifies and assists in the development of drivers and enhances their portability between RTEID based ~~system~~ systems. The RTEID does not make any assumptions about the construction or operation of a driver itself.

The directives provided by the I/O Interface are:

Directive	Description
de_init	Initialize a device driver
de_open	Open a device for I/O
de_close	Close a device
de_read	Read from a device
de_write	Write to a device
de_cntrl	Special device services

4.1 Driver Properties

Device drivers shall have the following properties:

1. A driver is always called by a task and is considered to run on behalf of the task which called it.
2. A driver can make any and all RTEID calls, including additional I/O calls. I/O calls may not be called from within the driver's ISR.
3. If the driver makes a blocking service call, (e.g. *q_receive*), the calling task blocks.
4. Drivers always execute in supervisor mode regardless of the mode of the caller. Designers should account for driver stack usage when determining supervisor stack sizes for new tasks.
5. A driver may temporarily enter user mode but must return to supervisor mode prior to exiting.
6. Other than item (4) above, drivers retain the *mode* of the calling task. Thus on entry they have the same interrupt mask level, preemption, asr and time-slicing status as the caller. The driver may change any or all of these but is responsible for restoring them prior to exiting.

4.2 Data Structures

The data structures used by drivers which are supported by the I/O Interface are:

- **Driver Address Table**
  - Used by the I/O Interface to locate the driver's INIT, OPEN, CLOSE, READ, WRITE, and CNTRL routines.

- **Device Data Area Table**
  - Used by the I/O Interface to locate the driver's data area for the driver's OPEN, CLOSE, READ, WRITE, and CNTRL routines.

4.2.1 Driver Address Table

When a task makes an I/O Interface call, the executive must locate the driver associated with the specified device (major number) and operation (i.e. READ). It does so via a **Driver Address Table** provided by the user. The physical address of the table and the number of devices are specified to the executive via configuration parameters.

The **Driver Address Table** for a system with N devices can be described by the following declarations:

```
struct drvaddr drvatab[N];

struct drvaddr
{
int (*init_driver)();
int (*open_driver)();
int (*close_driver)();
int (*read_driver)();
int (*write_driver)();
int (*cntrl_driver)();
int resvd1;
int resvd2;
}
```

As shown, the **Driver Address Table** is an array of N structures, one for each device. Each structure contains eight entries. The first six entries contain pointers to functions (routines) within the driver associated with the device. The last two entries are reserved for future use.

4.2.2 Device Data Area Table

Many, if not most, devices need a data area where the device driver can store information specific to the device. Although a statically allocated area can be used, it is usually more convenient to dynamically allocate this area when the device is initialized. The I/O Interface contains services to support such dynamic allocation.

The **Device Data Area Table** is supplied and maintained by the I/O Interface. The table contains one long word entry for each device in the system. The entry is used to maintain the address of the data area for the device.

The device driver's INIT routine is responsible for allocating the device's data area and returning its address to the I/O Interface. This memory can come from any source - static data, a region, or a partition. On exit, the INIT routine must return the address of the data area to the I/O Interface. The I/O Interface saves this address in the **Device Data Area Table**. Whenever a device driver routine (other than INIT) is called, the I/O Interface passes the data area address to the driver.

4.3 Device Initialisation

During system initialisation, the executive automatically calls the driver's INIT routine for each device. They are called sequentially, beginning with device 0 and ending with the last device in the system.

Since drivers can only be called by tasks, the executive calls the driver's INIT routine on behalf of a system initialisation task, defined by configuration parameters. The *mode* of the system initialisation task (also a configuration parameter) is used as the *mode* while the executive calls the INIT routines of the drivers. If the driver's INIT routine makes a RTEID call which blocks, control is passed to an idle task provided by the executive until an interrupt unblocks the driver.

Although the driver's INIT routine is always called at system startup, it may also be called by a task, either to re-initialise a driver or when a new device driver is dynamically loaded.

4.4 Parameter Passing

All directives except *de_init* require a user provided parameter block. The format and content of the parameter block depends on and is determined entirely by the particular driver and device it controls. Its function is to pass input parameters to the driver.

In a system with an MMU, the address of the parameter block is a logical address. The I/O Interface will convert it to a physical address before passing it to the driver. Within the parameter block, addresses may be either logical or physical, as defined by the driver. The I/O Interface does not examine or translate any fields within the parameter block.

4.5 I/O Interface in C Language

The I/O Interface may be called in the C language as follows:

Function	Parameters
de_init	(dev)
de_open	(dev, argp, &rval)
de_close	(dev, argp, &rval)
de_read	(dev, argp, &rval)
de_write	(dev, argp, &rval)
de_cntrl	(dev, argp, &rval)

dev is a 32-bit device number formatted as follows:

bits 31-16 = major device number
bits 15-0 = minor device number

argp is a pointer to a parameter block which contains device and operation specific parameters. The format and contents of the block is determined by the driver.

rval is an output parameter in which READ, WRITE and CNTRL routines may return information about the call.

4.6 I/O Interface in Assembly Language

The I/O Interface may be called by loading parameters into specific CPU registers and executing a TRAP instruction. The following assembly language interface is used:

INPUT

D0.W = function number as follows:
1 = INIT
2 = OPEN
3 = CLOSE
4 = READ
5 = WRITE
6 = CNTRL
7 = RESVD1
8 = RESVD2

D1.L = Device number (major and minor)
A0.L = Pointer to parameter block (except INIT)

OUTPUT

D0.L = Error code - 0 indicates successful return
D1.L = Return value from OPEN, CLOSE, READ, WRITE and CNTRL
A1.L = Address of device data area (INIT only)

4.7 Driver Interface in Assembly Language

The I/O Interface calls the user provided driver using the following assembly language convention:

INPUT

D0.L = tid
D1.L = Device number (major and minor)
A0.L = Physical address of parameter block (except INIT)
A1.L = Physical address of device data area (except INIT)

OUTPUT

D0.L = Error code - 0 indicates successful return
D1.L = Return value from OPEN, CLOSE, READ, WRITE and CNTRL
A1.L = Address of device data area (INIT only)

4.8 Error Handling

There are a number of errors which can occur during a driver call. In general, there are two types:

1. Errors detected by the I/O Interface.
2. Errors detected and returned by the driver.

All I/O Interface generated errors are detected prior to calling the driver. In these cases, the I/O supervisor loads register D0 with an error code and returns to the caller without ever passing control to the driver. To distinguish between I/O Interface errors and driver errors, error codes below 10000H (16-bit values) are reserved for use by the I/O Interface. Below is a list of the errors which are detected by the I/O Interface:

Illegal Function Code
Illegal Major Device Number
Illegal to call driver from ISR
Illegal parameter block address (MMU version only)

Drivers should always return error codes which are greater than 10000H (non-zero in the upper 16-bits).

Error codes returned from the driver's INIT routine are ignored by the executive. If a driver's INIT encounters a fatal error during system startup, the *k_fatal* directive may be used.

4.9 I/O Interface Routines in C Language

The I/O Interface routines as called in the C language are described in the following pages.

4.9.1 INIT

NAME

de_init – "Initialise a Device Driver"

SYNOPSIS

```
uint de_init ( dev )

uint dev;    /* 32-bit device number */
```

DESCRIPTION

The INIT routine will be called during system initialization. The function of INIT is to setup the hardware as necessary and to initialise the driver dependent variables. If the driver needs to allocate a data area for its use, it would do so in the INIT routine. The address of this data area is saved in the Device Data Area Table by the I/O Interface.

RETURN VALUE

If the call succeeds, then 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.2 OPEN

NAME

de_open -- "Open a device for I/O"

SYNOPSIS

uint de_open (dev, argp, &rval)

```
uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The OPEN routine is generally used in conjunction with the CLOSE routine. An example is the implementation of mutual exclusion. OPEN can define the start of a task's exclusive access to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.3 CLOSE

NAME

de_close -- "Close a device"

SYNOPSIS

```
uint de_close ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The CLOSE routine is generally used in conjunction with the OPEN routine. An example is the implementation of mutual exclusion. CLOSE can define the end of a task's exclusive access to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.4 READ

NAME

de_read -- "Read from a device"

SYNOPSIS

uint de_read (dev, argp, &rval)

```
uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The READ routine transfers data from a device to a user's buffer.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.5 WRITE

NAME

de_write – "Write to a device"

SYNOPSIS

```
uint de_write ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The WRITE routine transfers data from a user's buffer to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.6 CNTRL

NAME

de_cntrl -- "Special device services"

SYNOPSIS

```
uint de_cntrl ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* Return value - returned by this call */
```

DESCRIPTION

The function of the CNTRL routine is driver dependent. For serial I/O drivers, this routine can define tty parameters such as the baud rate. For a disk driver, the CNTRL routine can include disk formatting.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.10 Driver Interface in C Language

In addition to the assembly language interface, a driver may support the C language interface as defined below:

Function	Parameters
dx_init	(dev, &datap, tid)
dx_open	(dev, pargp, datap, tid, &rval)
dx_close	(dev, pargp, datap, tid, &rval)
dx_read	(dev, pargp, datap, tid, &rval)
dx_write	(dev, pargp, datap, tid, &rval)
dx_cntrl	(dev, pargp, datap, tid, &rval)

dev is the 32-bit device number.

bits 31-16 = major device number
bits 15-0 = minor device number

pargp is the physical address of the parameter block which contains device and operation specific parameters. The format and contents of the block is determined by the driver.

datap is the physical address of the device's data area for all calls except INIT. *datap* is an output parameter in which the INIT routine returns the address of the device's data area.

tid is the task id of the calling task.

rval is an output parameter in which READ, WRITE and CNTRL routines may return completion information about the call.