Please refer to the *FGA-002 User’s Manual*, chapter 10, which includes a guide to the latest revision of the boot software.
VMEPROM SYSTEM OVERVIEW
# TABLE OF CONTENTS

## 1. General Overview .............................................. 2-1

1.1 VMEPROM Modules ............................................ 2-1  
1.2 Features of VMEPROM .......................................... 2-1  
1.2.1 Debugging Functions ........................................ 2-1  
1.2.2 System Functions ........................................... 2-2  
1.2.3 System Calls ............................................... 2-2  
1.2.4 Application Command Interface .............................. 2-2  

## 2. Starting VMEPROM ............................................ 2-3

2.1 Power Up Sequence ............................................ 2-3  
2.2 Front Panel Switches ........................................... 2-4  
2.2.1 RESET Switch ............................................... 2-4  
2.2.2 ABORT Switch ............................................... 2-4  
2.3 The Rotary Switches ............................................ 2-5  
2.3.1 Lower Rotary Switch ....................................... 2-5  
2.3.2 Executing the Startup File ................................ 2-5  
2.3.3 Program Execution ......................................... 2-6  
2.3.4 Upper Rotary Switch ....................................... 2-7  
2.4 Memory Configuration ......................................... 2-7  
2.5 Terminal/RAM Port ............................................ 2-8  
2.6 Start Application Command Interface ........................... 2-8  
2.6.1 Default Memory Usage .................................... 2-9  
2.6.2 Default EPROM Usage ....................................... 2-10  

## 3. Details of the IBC Board ...................................... 2-11

3.1 EPROM/RAM Address and Device Table ........................ 2-11  
3.2 On-board I/O Devices ........................................ 2-12  
3.3 On-board Interrupt Sources .................................. 2-12  
3.4 The On-board Real Time Clock ................................ 2-13  
3.5 Off-board Interrupt Sources .................................. 2-13  

## 4. VMEPROM Kernel ............................................ 2-14

4.1 Special VMEPROM commands for the IBC-20 board .......... 2-15  
4.2 CONFIG - Search VMEbus for Hardware ....................... 2-15  
4.3 FLUSH - Set Buffered Write Mode ............................. 2-16
LIST OF TABLES

Table 1: Lower Rotary Switch ................................................ 2-5
Table 2: Program Execution .................................................. 2-6
Table 3: Upper Rotary Switch ................................................ 2-7
Table 4: Memory Configurations .......................................... 2-7
Table 5: Memory layout of the on-board RAM ............................. 2-9
Table 6: Layout of the Default EPROM Usage ............................ 2-10
Table 7: EPROM/RAM Addresses ........................................... 2-11
Table 8: On-board I/O Devices ............................................. 2-12
Table 9: On-board Interrupt Sources .................................... 2-12
Table 10: Off-board Interrupt Sources ................................. 2-13
1. General Overview

1.1 VMEPROM Modules

VMEPROM is a PDOS based real time Monitor. It consists of two basic parts:

1) PDOS Kernel with BIOS modules
2) User Interface

The first part, the PDOS Kernel with the BIOS modules, consumes around 32 Kbytes. This part is responsible for all the system calls and the hardware interface.

The second part is much bigger and contains the complete user interface, the built-in functions and debugging facilities. The size of this part is about 256 Kbytes.

The remaining space in the EPROM is reserved for future expansions.

The kernel features over 100 system calls and is 100% identical to PDOS.

The user interface gives the user both a debugging tool and an interface to the system functions. It includes breakpoint setting, tracing, a powerful line assembler/disassembler, task management, and event control.

1.2 Features of VMEPROM

1.2.1 Debugging Functions

- Line assembler/disassembler with full support of all 68020/68881 instructions.
- Over 20 commands for program debugging, including breakpoints, tracing, processor register display and modify.
- Display and modify floating point data registers.
- S-record up/downloading from any port in the system.
- Time stamping of user programs.
- Built-in Benchmarks
- Support of RAM disk, floppy and Winchester disks, including disk formatting and initialization.
Debugging Functions Continued

- Serial I/O support for up to two SYS68K/SIO-1/2 or SYS68K ISIO-1/2 boards in the system.
- Support for EAGLE modules and base boards equipped with an Application Command Interface (ACI).
- EPROM programming utility using the SYS68K/RR-2/3 board.
- Full screen editor.
- Numerous commands to control the PDOS kernel and file manager.

1.2.2 System Functions

- EPROM programming utility, using the SYS68K/RR-2/RR-3 boards.
- FLASH EPROM programming utility
- Complete task management.
- I/O redirection on the command line.

1.2.3 System Calls

- Over 100 system calls.
- Data conversion and file management functions.
- Task management system calls.
- Terminal I/O functions.

1.2.4 Application Command Interface

Depending on the state of the front panel rotary switch, two additional tasks are started together with VMEPROM. The Application Command Interface software is started as a second task and a third task - the DMA task - is started by the ACI.

Please refer to the "ACI Programming Guide" in this manual for further details about the Application Command Interface (ACI).
2. Starting VMEPROM

2.1 Power Up Sequence

The first executed software after powerup is the IBC Boot Software. Immediately after this, the VMEPROM will be started. Control is given to the BIOS modules of VMEPROM to perform all the necessary hardware initialization of the IBC. The real time kernel is started and the user interface of VMEPROM is invoked as the first task. This sequence also reads the Real Time Clock (RTC) of the IBC board and initializes the software clock of the kernel.

If a terminal is connected to the terminal port of the IBC board, the VMEPROM banner and the VMEPROM prompt ("? ") will be displayed upon powerup or reset. Otherwise the VMEPROM installs a RAM port to be used as an alternate I/O port of the VMEPROM.

The default terminal port setup is as follows:

**Asynchronous communication**
- 9600 Baud
- 8 data bits
- 1 stop bit
- no parity

**Hardware handshake protocol**

If the above message does not appear, check the following:

1. Baud rate and character format setting of the terminal (default upon delivery of the IBC board is 9600 Baud, 8 data bits, 1 stop bit, no parity).
2. Cable connection from the IBC board to the terminal (refer to the Hardware User’s Manual for the pinning of the DSub connector and the required handshake signals).
3. Power supply, +5V, +12V, -12V must be present. See the Hardware User’s Manual for the power consumption of the IBC board.

If everything goes well, the header and prompt are displayed on the terminal and VMEPROM is now ready to accept commands.

The Application Command Interface provides a unique interface to access the physical devices on available EAGLE modules.
2.2 Front Panel Switches

2.2.1 RESET Switch

Pressing the RESET switch on the front panel causes all programs to terminate immediately and resets the 68020 processor and all I/O devices.

When the VMEPROM kernel is started, it overwrites the first word in the user memory after the task control block with an EXIT system call. If breakpoints were defined and a user program was running when the RESET button was pressed, the user program could possibly be destroyed.

Pressing reset while a program is running should only be done as a last resort when all other actions (such as pressing ^C twice or the ABORT Switch) have failed.

2.2.2 ABORT Switch

The ABORT Switch is defined by VMEPROM to cause a level 7 interrupt. This interrupt cannot be disabled and is therefore the appropriate way to terminate a user program and return to the command level of VMEPROM.

If ABORT is pressed while a user program is under execution, all user registers are saved at the current location of the program counter and the message "Aborted Task" is displayed along with the contents of the processor register.

If ABORT is pressed while a built-in command is executed or the command interpreter is waiting for input, only the message is displayed and control is transferred to the command interpreter. The processor register are not modified and are not displayed in this case.

NOTE: Tasks with port 0 as its input port will not be aborted.
2.3 The Rotary Switches

Two rotary switches are available on the IBC-20 base board: The rotary switch accessible on the front panel is called the "Upper Rotary" switch; whereas the second rotary switch on the base board is called the "Lower Rotary" switch. The state of these two rotary switches are read by VMEPROM after RESET and control the options described below:

2.3.1 Lower Rotary Switch

Table 1: Lower Rotary Switch

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Controls whether a startup file is executed (typically SY$STRT)</td>
</tr>
<tr>
<td>2,3</td>
<td>Selects one of four addresses where to continue execution after the kernel has been initialized.</td>
</tr>
</tbody>
</table>

2.3.2 Executing the Startup File

The second bit (bit 1) of the lower rotary switch controls whether a given startup file has to be executed by the VMEPROM shell. The name of this startup file is included in the configuration table at offset 0_{16} and is called "SY$STRT".

If the second bit is set (bit 1), then no startup file is executed by the VMEPROM shell; otherwise, when the bit is cleared (0), the VMEPROM shell executes the startup file specified by the appropriate entry in the configuration table.
2.3.3 Program Start

Bits 2 and 3 are used to control where the firmware continues execution after the VMEPROM kernel is initialized. Depending on the state of bits 2 and 3, the firmware fetches one of four addresses from a table within the configuration table. This table contains the addresses where the firmware continues execution after the kernel is initialized. The table below lists the relationship between the state of the two bits of the lower rotary switch and the address being fetched.

Table 2: Program Execution

<table>
<thead>
<tr>
<th>Lower Rotary Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 3</td>
<td>Bit 2</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

As shown in the table above, the firmware continues execution in all four cases at predefined locations. In the first, third and fourth cases, the firmware continues execution at the appropriate addresses via Jump-Subroutine instruction (JSR); when the application returns to the firmware by a Return-From-Subroutine instruction (RTS), the firmware continues to start the VMEPROM shell.

In the second case, the firmware fetches the stack pointer and the program counter, located at offset $0_{16}$ and $4_{16}$ of the binary image addressed by the entry, and continues execution at the location specified by the program counter (the firmware uses a Jump instruction (JMP) to continue execution).

In general, it is wise to keep at least the fourth entry of the table located at offset $40_{16}$ within the configuration table; however, the first three entries of the table at offset $40_{16}$ are to be modified accordingly.
2.3.4 Upper Rotary Switch

Table 3: Upper Rotary Switch

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>Memory Configuration</td>
</tr>
<tr>
<td>2</td>
<td>Terminal/RAM port</td>
</tr>
<tr>
<td>3</td>
<td>Start Application Command Interface (ACI)</td>
</tr>
</tbody>
</table>

2.4 Memory Configuration

These two bits are used to describe how the on-board memory has to be shared between the VMEPROM and the task supporting the Application Command Interface. In the table below all possible memory configurations are stated:

Table 4: Memory Configurations

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Memory Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Three fourths of the on-board memory is available to VMEPROM. The remaining memory is available to ACI.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>One half of the on-board memory is available to VMEPROM. The remaining memory is available to ACI.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>One fourth of the on-board memory is available to VMEPROM. The remaining memory is available to ACI.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>One eighth of the on-board memory is available to VMEPROM. The remaining memory is available to ACI.</td>
</tr>
</tbody>
</table>

NOTE: The state of these two bits are only evaluated when the most significant bit (bit 3) of the upper rotary switch is set.
2.5 Terminal/RAM Port

This bit specifies whether the I/O port of VMEPROM will be either the RAM port provided by the Application Command Interface or the first serial communications channel (channel A) of the base board’s SCC Z85C30.

If the bit is set, then the port 1 is selected to be the I/O port of VMEPROM. The firmware verifies whether a terminal is connected with the serial interface of the first port (CTS and DCD must be active for the firmware to assign the first port as I/O port to VMEPROM.) In case a terminal attached to the serial interface, port 1 is used as I/O port; otherwise, the RAM port is being used as the task’s I/O port. If the bit is cleared, then the RAM port is specified to be the task’s I/O port.

**NOTE:** The state of this bit is only evaluated when the most significant bit (bit 3) of the upper rotary switch is set.

2.6 Start Application Command Interface

This bit specifies whether the Application Command Interface (ACI) has to be provided by the base board. If this bit is set, then the firmware starts a second task - supporting the ACI - in addition to VMEPROM. The *on-board* memory is shared between the two tasks as specified by the first and second bits (bit 0 and 1) of the rotary switch.

If this bit is cleared, then the base board does not provide the ACI and no further tasks are started by the firmware. The entire *on-board* memory is available to VMEPROM.
2.6.1 Default Memory Usage

By default, VMEPROM uses the following memory assignment for the IBC board:

Table 5: Memory layout of the on-board RAM

<table>
<thead>
<tr>
<th>MEMORY LAYOUT OF THE ON-BOARD RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00000</td>
</tr>
<tr>
<td>$00400</td>
</tr>
<tr>
<td>$00800</td>
</tr>
<tr>
<td>$01000</td>
</tr>
<tr>
<td>$07000</td>
</tr>
<tr>
<td>$080000</td>
</tr>
<tr>
<td>Highest On-board Memory Address</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### 2.6.2 Default EPROM Usage

Table 6: Layout of the Default EPROM Usage

<table>
<thead>
<tr>
<th>EPROM USAGE LAYOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFE00000</td>
</tr>
<tr>
<td>$FFE00004</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$FFE10000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
3. Details of the IBC Board

3.1 EPROM/RAM Address and Device Table

Table 7: EPROM/RAM Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>Local RAM</td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>........ *</td>
<td></td>
</tr>
<tr>
<td>FFC0 0000</td>
<td>SRAM Area</td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>FFC7 FFFF</td>
<td></td>
</tr>
<tr>
<td>FFC8 0000</td>
<td>FLASH EPROM Area</td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>FFCF FFFF</td>
<td></td>
</tr>
<tr>
<td>FFE0 0000</td>
<td>EPROM Area</td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>FFEF FFFF</td>
<td></td>
</tr>
</tbody>
</table>

* → Highest On-Board Memory Address
3.2 On-board I/O Devices

The following table shows the base addresses of the on-board I/O devices.

**Table 8: On-board I/O Devices**

<table>
<thead>
<tr>
<th>BASE ADDRESS</th>
<th>DEVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FF803000</td>
<td>RTC 72423</td>
</tr>
<tr>
<td>$FF802000</td>
<td>SCC Z8530 (Channel B)</td>
</tr>
<tr>
<td>$FF802020</td>
<td>SCC Z8530 (Channel A)</td>
</tr>
<tr>
<td>$FF800C00</td>
<td>CIO Z8536</td>
</tr>
<tr>
<td>$FFD00000</td>
<td>FGA-002A</td>
</tr>
</tbody>
</table>

3.3 On-board Interrupt Sources

The following table is used for the on-board interrupt sources and levels which are defined by VMEPROM. All interrupt levels and vectors of the on-board I/O devices are software programmable via the FGA-002A Gate Array.

**Table 9: On-board Interrupt Sources**

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>INTERRUPT LEVEL</th>
<th>INTERRUPT VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort Switch</td>
<td>7</td>
<td>232</td>
</tr>
<tr>
<td>CIO</td>
<td>5</td>
<td>242</td>
</tr>
<tr>
<td>SCC</td>
<td>4</td>
<td>245</td>
</tr>
<tr>
<td>Application Command Interface</td>
<td>2</td>
<td>192</td>
</tr>
<tr>
<td>EAGLE UART Driver</td>
<td>5</td>
<td>196</td>
</tr>
<tr>
<td>EAGLE Disk Driver</td>
<td>5</td>
<td>198</td>
</tr>
</tbody>
</table>
3.4 The On-board Real Time Clock

During the powerup sequence, the on-board real time clock of the IBC board is read and loaded in the VMEPROM. This sequence is done automatically and requires no user intervention. If the software clock of VMEPROM is set by the ID command, the RTC is set automatically to the new time and date values.

3.5 Off-board Interrupt Sources

VMEPROM supports several VMEbus boards. As these boards are interrupt driven the level and vectors must be defined for VMEPROM to work properly. The following table shows the default setup of the interrupt levels and vectors of the supported hardware. For a detailed description of the hardware setup of the boards, please refer to the Appendix of this manual. The supported I/O boards together with the base addresses and the interrupt level and vector are summarized in Table 9. In order for these boards to work correctly with VMEPROM, the listed interrupt vectors may not be used.

Table 10: Off-board Interrupt Sources

<table>
<thead>
<tr>
<th>Board</th>
<th>Interrupt Level</th>
<th>Interrupt Vector</th>
<th>Board Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIO-1/2</td>
<td>4</td>
<td>169-181</td>
<td>$FCB00000</td>
</tr>
<tr>
<td>ISIO-1/2</td>
<td>4</td>
<td>161-168</td>
<td>$FC960000</td>
</tr>
<tr>
<td>ISCSI-1</td>
<td>4</td>
<td>160</td>
<td>$FCA00000</td>
</tr>
<tr>
<td>IBC UART Driver</td>
<td>5</td>
<td>197</td>
<td>---</td>
</tr>
<tr>
<td>IBC Disk Driver</td>
<td>5</td>
<td>199</td>
<td>---</td>
</tr>
</tbody>
</table>
4. VMEPROM Kernel

The functions of the VMEPROM kernel are described here in detail. There are three main sections of VMEPROM; namely, the BIOS, kernel, and the user interface.

VMEPROM is based on the powerful PDOS real time kernel.

Features of the kernel:

1. Multitasking scheduling
2. System clock
3. Memory allocation
4. Task synchronization
5. Task suspension
6. Event processing
7. Character I/O including buffering
8. Support primitives

The PDOS kernel is the multitasking, real time nucleus of the VMEPROM. Tasks are the components comprising a real time application. It is the main responsibility of the kernel to see that each task is provided with the support it requires in order to perform its designated function.

The main responsibilities of the kernel are the allocation of memory and the scheduling of tasks. Each task must share the system processor with other tasks. The kernel saves the task’s context when it is not executing and restores it again when it is scheduled. Other responsibilities of the kernel are maintenance of a 24 hour system clock, task suspension and rescheduling, event processing, character buffering and other support functions.
4.1 Special VMEPROM commands for the IBC-20 board

The commands described below are provided by VMEPROM in addition to the commands described in Section 3 of this manual.

4.2 CONFIG - Search VMEbus for Hardware

Format: CONFIG

This command searches the VMEbus for available hardware. It is useful if VMEPROM is started and bit 0 of the lower rotary switch on the front panel is set to "1", so that VMEPROM does not check the configuration by default.

In addition this command allows the user to install additional memory in the system. Additional memory can ONLY be installed with this command.

The following hardware is detected:

1. ISIO-1/2
2. SIO-1/2
3. ISCSI-1
4. Boards providing the Application Command Interface
5. Contiguous memory starting at the highest on-board memory address

The boards must be set to the default address for 32 bit systems. This setup is summarized for all supported boards in the Appendix of this manual.

Additional memory must be contiguous to the on-board memory of the CPU board. This memory is cleared by the config command to allow DRAM boards with parity to be used. Please remember that the installation of additional memory does not effect the RAM size of the running task. However, VMEPROM identifies this installed memory area and every time memory is required (i.e. with CT or FM) it is taken from this area as long as there is enough free space.

The CONFIG command also installs Winchester disks in the system and initializes the disk controller (if available). So if a SYSFAIL is active on the VMEbus (which can come for example from the ISIO-1/2 or ISCSI-1 controller during selftest), the command is suspended until the SYSFAIL signal is no longer active.

An example follows on the next page.
Example:

? CONFIG<cr>
UART FORCE ISIO1/2 (U3) INSTALLED
ISCSI-1: 1 boards available
ISIO-1/2: 1 boards available

?_

4.3 FLUSH - Set Buffered Write Mode

Format: FLUSH
FLUSH <disk number>,<time>

The first command flushes all buffers on all disks in the system.

The second command sets a flush time for the device driver task. The device driver task has to flush its buffers periodically every <time> seconds. Please refer to the USER’S MANUAL of the Module to see if the device driver task is able to handle this service. The parameter <disk number> is only used to select a specific device driver task. Every disk which is connected to this task is flushed.

Example

? FLUSH

All modified buffers are flushed

? FLUSH 2 20

Flush time: 20 seconds

? _
VMEPROM BUILT-IN COMMANDS
# TABLE OF CONTENTS

## 1. GENERAL INFORMATION ....................................... 3-1

### 1.1 Command Line Syntax and Line Editing ........................................ 3-1

### 1.2 VMEPROM Built-in Commands ........................................... 3-6

#### 1.2.1 # - Symbolic Command Name ..................................... 3-7
#### 1.2.2 ARB - Set the Arbiter of the IBC Board ............................... 3-8
#### 1.2.3 AS - LINE ASSEMBLER .......................................... 3-9
#### 1.2.4 ASSIGN - Assign New Input or Output Ports ........................... 3-11
#### 1.2.5 BASE - SET/DISPLAY BASE REGISTER ................................ 3-12
#### 1.2.6 BENCH - Built-in Benchmarks .................................... 3-13
#### 1.2.7 BF - Block Fill ................................................ 3-14
#### 1.2.8 BM - Block Move ............................................. 3-15
#### 1.2.9 BP - BAUD PORT ............................................. 3-16
#### 1.2.10 BR - Set/Display/Delete Breakpoints ............................... 3-17
#### 1.2.11 BS - Block Search ............................................. 3-18
#### 1.2.12 BT - Block Test ............................................... 3-19
#### 1.2.13 BV - Block Verify ............................................. 3-20
#### 1.2.14 COLD - Cold Start VMEPROM ................................... 3-21
#### 1.2.15 CREATE TASK ............................................... 3-22
#### 1.2.16 DI - Disassembler ............................................. 3-24
#### 1.2.17 DR - Display Processor Registers .................................. 3-25
#### 1.2.18 DT - DATE AND TIME ......................................... 3-26
#### 1.2.19 DU - Dump S-record ........................................... 3-27
#### 1.2.20 EAGLE - Display All Information About Available EAGLE Modules .......... 3-28
#### 1.2.21 ER - LIST ERRORS ........................................... 3-29
#### 1.2.22 EV - SET/RESET EVENT ....................................... 3-30
#### 1.2.23 FGA - Change Boot Setup for Gate Array ............................ 3-32
#### 1.2.24 FM - FREE MEMORY .......................................... 3-33
#### 1.2.25 FMB - Force Message Broadcast .................................. 3-34
#### 1.2.26 FUNCTIONAL - Perform Functional Test ............................ 3-35
#### 1.2.27 GO - Start User Program ........................................ 3-36
#### 1.2.28 GD - Start User Program Without Breakpoints ........................ 3-37
#### 1.2.29 GM - GET MEMORY .......................................... 3-38
#### 1.2.30 GT - Start User Program with Temporary Breakpoint .................... 3-39
#### 1.2.31 HELP - HELP ................................................ 3-40
#### 1.2.32 HIST - Command history ........................................ 3-41
#### 1.2.33 ID - SET SYSTEM DATE/TIME ................................... 3-42
#### 1.2.34 INFO - Information about the CPU board ............................. 3-43
#### 1.2.35 KM - KILL MESSAGE .......................................... 3-44
#### 1.2.36 KT - KILL TASK ............................................... 3-45
#### 1.2.37 LO - Load S-record ............................................ 3-46
#### 1.2.38 LT - LIST TASKS ............................................. 3-47
#### 1.2.39 M - Modify Memory ............................................ 3-48
#### 1.2.40 MD - Display Memory .......................................... 3-50
#### 1.2.41 MEM - Set Data Bus Width of the VMEbus ................................ 3-51
| 1.2.42 | MS     | Set Memory to Constant or String | 3-52  |
| 1.2.43 | PROG   | Program FLASH EPROM               | 3-53  |
| 1.2.44 | PROMPT | CHANGE PROMPT SIGN                | 3-54  |
| 1.2.45 | RM     | Modify Processor Registers        | 3-55  |
| 1.2.46 | RR2    | EPROM Programming                 | 3-57  |
| 1.2.47 | SELFTEST | Perform On-board Selftest        | 3-62  |
| 1.2.48 | SM     | SEND MESSAGE                      | 3-63  |
| 1.2.49 | ST     | SET TASK TERMINAL TYPE            | 3-64  |
| 1.2.50 | T      | Trace Program Execution           | 3-67  |
| 1.2.51 | TC     | Set Trace Count                   | 3-70  |
| 1.2.52 | TIME   | Enable/Disable Program Run Time Display | 3-71 |
| 1.2.53 | TJ     | Trace on Change of Flow           | 3-72  |
| 1.2.54 | TM     | TRANSPARENT MODE                  | 3-73  |
| 1.2.55 | TP     | TASK PRIORITY                     | 3-74  |
| 1.2.56 | UN     | CONSOLE UNIT                      | 3-75  |
| 1.2.57 | ZM     | ZERO MEMORY                       | 3-76  |
This page was intentionally left blank
1. GENERAL INFORMATION

The VMEPROM command interpreter is a set of resident routines for program debugging and handling of the most common kernel functions. The command interpreter then searches for a given command. If a match is found, the command is executed.

The prompt of VMEPROM is a single question mark, followed by a space ("? ").

1.1 Command Line Syntax and Line Editing

Command Line Arguments

The VMEPROM command interpreter allows several options. In general the complete command line is divided into separate arguments. The arguments must be separated by one or more spaces or a comma. If a null-argument has to be entered, it must be represented by a comma only.

Example: ? PROG ARG1,,ARG3,

In this example, the arguments number 2 and 4 are null-arguments.

If any argument is using a comma, space, period or one of the I/O redirection arrows, it has to be put in brackets to suspend the command line interpretation.

Example: ? PROG1 (Hello, world.),(<....>),>2

Port 2 now shows the output of PROG1 which may be:

ARGUMENT 1 was: Hello, world.
ARGUMENT 2 was: <....>
ARGUMENT 3 was:
ARGUMENT 4 was:
ARGUMENT 5 was:
Input/Output Redirection

VMEPROM supports simple I/O redirection. The specifiers are the signs '<' for input and '>' for output and may appear at any location in the command line, but must be after the command name. Immediately after the redirection signs '<' and '>', a port number must be specified. The port number may be one of the ports available in the system. It is expected to be given in hexadecimal number system. The arguments specifying the I/O redirection are removed from the command line by the command interpreter and do not appear in the built-in command.

Example: ? PROG <2>3 ARG1,ARG2,ARG3,ARG4

In this example, the program PROG is started. It is getting all inputs from port 2 and all output is redirected to port 3.

Multiple Commands

VMEPROM allows command lines of up to 78 characters. This command line can contain several different commands. The parsing of the command line is terminated at the first period (".") and the remaining command line is saved to be used later.

Example: ? RM D0 12345678.SM 2,Hello
 ? SM 2,Hello
  ?

Command Line Editing

The PDOS get line (XGLM) primitive is used to get a command line of up to 78 characters into the command line buffer.

Input is normally in replace mode which means an incoming character replaces the character at the cursor. Various control characters can be used to edit the input line.

The following table summarizes the control characters:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ESC]</td>
<td>Cancel current line</td>
</tr>
<tr>
<td>[CTRL-C]</td>
<td>Cancel current line</td>
</tr>
<tr>
<td>[CTRL-I]</td>
<td>Enter insert mode</td>
</tr>
<tr>
<td>[CTRL-A]</td>
<td>Recall last entered line</td>
</tr>
<tr>
<td>[CTRL-L]</td>
<td>Move right 1 character</td>
</tr>
<tr>
<td>[CTRL-H]</td>
<td>Move left 1 character</td>
</tr>
<tr>
<td>[CTRL-D]</td>
<td>Delete character under cursor</td>
</tr>
<tr>
<td>[RUBOUT]</td>
<td>Delete 1 character to the left</td>
</tr>
</tbody>
</table>
A [CTRL-I] changes input from replace to insert mode. The mode returns to replace mode when any other editing control code is entered. Replace mode overwrites the character under the cursor. Insert mode inserts a character at the current cursor position.

In either mode, the cursor need not be at the end of the line when the [CR] is entered. The command line is passed as it appears on the screen.

When a line is accepted, it is copied to another buffer (MPB$) where it can be recalled by using the [CTRL-A] character. A [CR] and [LF] are output to the console followed by the recalled line. The cursor is positioned at the end of the line. This is a circular buffer and commands will rotate through it as they are recalled.

Numeric parameters are entered as signed decimal, unsigned hex, unsigned octal or unsigned binary numbers. All numbers are converted to two’s complement 32-bit or 16-bit integers depending on their function. Therefore it ranges from -2,147,483,648 to 2,147,483,647 (hex $80000000 to $7FFFFFFF) or -32,768 to 32,767 (hex $8000 to $7FFF). All built-in commands assume that numbers are entered in hex if not noted otherwise. To change from the expected number system, numbers must be preceded with a special sign. These are: a dollar sign ($) to enter into hexadecimal, an ampersand (&) to enter into decimal, an at/around sign (@) to enter into octal and a percent sign (%) to enter into the binary number system.

(Note: Numbers are not checked for overflow. Hence, $FFFFFFFF or 4,294,967,295 are equivalent to -1). A line beginning with an '*' is ignored. This is very useful to insert comment lines in command files.

In addition, every numeric parameter is passed through a Regular Expression Processor. This processor calculates numbers. For this calculation the following operations are allowed:

*: Multiplication
/: Division
+: Addition
-: Subtraction
^: Power of
(: Opening Bracket
): Closing Bracket
**Line Editing**

Some commands allow inputting data outside the command line. For this a line editor is used. There are some control characters to edit the line:

- `[ESC]` = Cancel current line and exit
- `[CTRL-C]` = Cancel current line and exit
- `[CTRL-I]` = Toggle between insert and replace mode.
  
  First the line editor is in insert mode.
- `[CTRL-A]` = Dependent on the called command.
- `[CTRL-L]` = Move right one character
- `[CTRL-E]` = Move to end of line
- `[CTRL-H]` = Move one character left
- `[CTRL-B]` = Move to begin of line
- `[CTRL-D]` = Delete character under cursor
- `[RUBOUT]` = Delete one character to the left
- `[CTRL-I]` = Delete character under cursor to end of line
- `[CTRL-O]` = Delete whole line

The cursor need not be at the end of the line when the [CR] is entered.
**Program or Command Abort**

There are two basic methods of aborting a running program or command. The first one is the ABORT switch on the CPU board. This switch causes a level 7 interrupt to the processor. If a VMEPROM command was under execution at this time, the message "Abort switch pressed" is displayed and control is transferred back to the command interpreter immediately. If a user program is running when the ABORT switch is pressed, the current contents of the processor registers are saved and a message along with the processor registers is displayed.

The second method is typing ^C twice on the keyboard. If that happens, VMEPROM will abort the current command and control is transferred to the command interpreter. The processor register is not saved by this action. They show the same status as they had before the program was started.

**NOTE:** Tasks with port 0 as input port will not be aborted.
1.2 VMEPROM Built-in Commands

The VMEPROM built-in commands are described in detail in this chapter.

The following notations are used throughout this document:

- Symbolic representation is put in arrows (i.e. `<address>` where an absolute address has to be inserted.

- Optional arguments are in square brackets (i.e. `[<option>]`). Those arguments must not be specified and have a default value.

- If one argument out of more can be selected, the arguments are separated by a `|` (i.e. `[B | W | L]` to select Byte, Word or Long Word size).

- If more than one out of many possibilities for an argument has to be selected, these are marked with a `&` sign (i.e. `[B|W|L&N&O|E]` to select B or W or L together with N and O or E).

Most of the VMEPROM commands assume that the parameters are given in hex (without a leading $ sign).

However, some values are assumed in decimal.

These are:

**Port**
VMEPROM port numbers are in the range 0-15 and have to be entered in decimal.

**Tasks**
The task numbers have to be entered in decimal

**Task Priorities**
The task priority has to be entered in decimal

**Error Numbers**
The error numbers are displayed in decimal and have to be entered in decimal
1.2.1 # - Symbolic Command Name

**Format:**

```
# <name>
# <name>,<command string>
```

The symbolic name command is used to display, delete or define a symbolic name for often used command lines. The first format displays all currently defined names, the second deletes a defined name from the list and the third one defines a new name with the command string. VMEPROM supports up to 5 symbolic names with command lines of up to 40 characters.

Symbolic names can reference other symbolic names.

**Example:**

```
? # ASM AS 8000 Define ASM for the command AS

? # DISP LT Define DISP for list tasks

? # D DISP Define D for DISP

? # Show defined symbolic names
ASM: AS 8000
DISP: LT
D: DISP

? DISP

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>256</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00047000</td>
<td>1/1/0/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>00FF800</td>
<td>0/0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
</tbody>
</table>

? D

? ASM

8000 : XEXT

? _
```
1.2.2 ARB - Set the Arbiter of the IBC Board

Format: ARB

The ARB command allows the user to set the arbitration mode of the IBC board for VMEbus. This command is also used to select the Standard Access Mode for the VMEbus. Additionally, the VMEbus interrupts can be enabled or disabled.

Example:

?ARB<cr>

Current arbiter mode: enabled, Mode = Prioritized ROUND ROBIN
  Set arbiter mode? (Y,y/-) : Y
  ROUND ROBIN mode ? (Y,y/-) : Y
  Prioritized ROUND ROBIN? (Y,y/-) : N
New arbiter mode = ROUND ROBIN
Set arbiter mode for VME-BUS:

  STATUS: ROR & RAT & RBCLR & FAIR
  SET: Release on bus clear (RBCLR) (Y/N) Y
  SET: Fair VME-BUS arbitration (FAIR) (Y/N) Y

Standard Access Mode (A24) for Slave Accesses currently disabled.
Enable A24 mode? (Y,y/-) : Y
  A31-A24 = 80
Change interrupt mask? (Y,y/-) : Y

Enable(1) / Disable(0) VMEbus interrupts by level:

  STATUS: Level: 7 6 5 4 3 2 1
          1 1 1 1 1 1 1
  SET: Enter new interrupt mask: 1 1 1 1 1 1 0

?_
1.2.3 AS - LINE ASSEMBLER

Format: AS <address>

The AS command invokes the line assembler/disassembler of VMEPROM. It can assemble and disassemble all 68020 instructions and all the PDOS system calls.

The AS command, when invoked, displays the current address offset and the address within the window. Then the current location is disassembled.

After the prompt on the next line, the user can enter one of the following:

1) A valid 680x0 mnemonic. Some 68020 addressing modes allow omission of arguments. These addressing modes can be entered by omitting the argument and typing the dividing character ','. 

   Examples: 
   - CLR.W ([$1,A0],D0.W,$2) 
   - CLR.W ([$1,A0],,$2) 
   - CLR.W ([,A0],,) 

2) A '#' sign followed by the new address within the window. This is an absolute address change.

3) An '=' to disassemble the same location again.

4) A '+' to disassemble the next location.

5) A '-' sign forces the disassembler to step back one possible opcode. If none is found the same location will be opened.

6) A '+' or '-' sign followed by the number of bytes the address has to be increased or decreased. This is a relative address change.

7) A '.' or [ESC] to exit the line assembler and return control to the command interpreter.

8) [CTRL-A] to edit the disassembled opcode.

All immediate values, addresses and offsets inside mnemonics are assumed to be entered in decimal. So hex values have to be proceeded with a dollar ($) sign. In addition, binary values may be used if proceeded by a percent sign ("%") and octal values if proceeded by an at/around sign ("@"). The disassemblers display all values in hex representation.

The line assembler accepts a pseudo opcode of the form DC.B, DC.W and DC.L to define constant data storage. An ASCII pattern can be stored by using DC.B with the format DC.B "ASCII. All characters after '"' will be written to memory. The disassembler displays all illegal opcodes as DC.W.
Both the line assembler and disassembler support the opcodes as described in Chapter 4 of the VMEPROM Manual.

Example:

```
? AS 8000
8000 : XEXT
     : MOVE.L #$123,D1  New opcode entered
8006 : ORI.B #0,D0  Step back one opcode
     : -
8000 : MOVE.L #$123,D1  Recall line
     : [CTRL-A] Line edited
     : MOVE.L #$1234,D1
8006 : ORI.B #0,D0  New opcode
     : XRDM
8008 : ORI.B #0,D0  Back 8 bytes
     : -8
8000 : MOVE.L #$1234,D1  Disassemble next instruction
     : +
8006 : XRDM  Disassemble next instruction
     : [CR]
8008 : ORI.B #0,D0  Go to absolute address $8010
     : #8010
8010 : ORI.B #0,D0  Back to the command interpreter
     : .
```

? _
1.2.4 ASSIGN - Assign New Input or Output Ports

Format:   ASSIGN <port>
          ASSIGN <port>,<output port>

The ASSIGN command has two functions, depending on the command line arguments. If the output port is omitted, ASSIGN sets a new input and output port for the current task. If the output port is specified, the default input/output ports are unchanged, but the alternate output ports of the task are changed. The output port specified must be in the range 1-4.

Example:

? ASSIGN 3       VMEPROM now uses port 3 for I/O
? ASSIGN 3,2     Use port 3 as unit 2 port
1.2.5 BASE - SET/DISPLAY BASE REGISTER

Format: 
BASE
BASE <address>

The BASE register in VMEPROM is used to offset all memory accesses into the tasks memory. So all debugging can be done relative to address 0, which is actually the begin address of your tasks memory. This saves a lot of typing and makes sure that no other tasks memory is destroyed by a typing error.

Example:

? base<cr> Display BASE register
Base = 00000000 : <cr> No changes

? base 8000<cr> Set BASE register to $8000

? base<cr> Display BASE register
Base = 00008000 : <cr>

? M 0<cr> Open address $0 +BASE register
8000+0000 A00E : <cr>
8000+0002 0000 : <cr>
8000+0004 0000 : .

?
1.2.6 BENCH - Built-in Benchmarks

Format: 

```
BENCH
BENCH <#,<address>
```

These functions can execute one of the built-in benchmarks. If only BENCH is entered, a short description of all benchmarks is displayed on the terminal. A benchmark is executed by entering the number of the benchmark (in decimal) and the address where it shall run in memory (in hex).

The following benchmarks are available:

- Bench 1: Decrement long word in memory, 10,000,000 times
- Bench 2: Pseudo DMA 1K bytes, 50,000 times
- Bench 3: Substring character search, 100,000 times, taken from EDN, 08/08/85
- Bench 4: Bit Test/Set/Reset, 100,000 times, taken from EDN, 08/08/85
- Bench 5: Bit Matrix Transposition, 100,000 times, taken from EDN, 08/08/85
- Bench 6: Cache test, executes 128K bytes program 1000 times
  CAUTION: This benchmark will destroy 128K bytes memory
- Bench 7: Floating Point - 1,000,000 Additions
- Bench 8: Floating Point - 1,000,000 Sines
- Bench 9: Floating Point - 1,000,000 Multiplications
- Bench 10: 100,000 Context switches
- Bench 11: 100,000 Set system event
- Bench 12: 100,000 Change task priority
- Bench 13: 100,000 Send and Receive task message
- Bench 14: 100,000 Read system time

Example:

```
? bench 1 8000 Execute benchmark #1 at address $8000
Bench 1: Decrement long word in memory, 10,000,000 times
Benchmark time = 0:07.23

? 
```
1.2.7 BF - Block Fill

Format:

- BF <begin>,<end>,<value>,[B | W | L]
- BF <begin>,<end>,<pattern>,P
- BF <begin>,<end>,<opcode>,O

This command fills the specified memory area with a constant. The type of the constant is defined by the option and may be a Byte, Word, Long word, Pattern or an Opcode. A pattern is an ASCII string which is to be put in inverted commas. The maximum length is only restricted by the length of the input line, which may not exceed 78 characters. An Opcode is each valid 680x0 mnemonic or an opcode as described in Chapter 4 of the VMEPROM Manual. If the pattern or the opcode contains argument separators, such as space, comma, or full stop, the data has to be put in brackets. If no option is specified, a default of Word is assumed.

Example:

```plaintext
? BF 8000 8100 NOP O

? MD 8000 10
00008000: 4E 71 4E 71 4E 71 4E 71 4E 71 4E 71 4E 71 4E 71 NqNqNqNqNqNqNq

? BF 8000 8100 ("Hello World") P

? MD 8000 10
00008000: 48 65 6C 6C 6F 20 57 6F 72 6C 64 48 65 6C 6C 6F Hello World Hello

? BF 8000 8100 12345678 L

? MD 8000 10
00008000: 12 34 56 78 12 34 56 78 12 34 56 78 12 34 56 78 .4Vx.4Vx.4Vx.4Vx

? BF 8000 8100 &255

? MD 8000 10
00008000: 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF .................
```

? _
1.2.8 BM - Block Move

Format: BM <begin>,<end>,<destination>

The BM command copies a memory from one area to another. The areas may be overlapped.

Example:

? MD 8000 20
00008000: 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF ................
00008010: 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF ................

? BM 8000 8020 9000

? MD 9000 20
00009000: 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF ................
00009010: 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF ................

? _
1.2.9 BP - BAUD PORT

Format:

```
BP
BP <port #>
BP {-}<port #>,<baud rate>
BP {-}<port #>,<baud rate>,<type>,<UART base addr>
```

The BAUD PORT command initializes a VMEPROM I/O port and binds a physical UART to a character buffer. The command sets the UART character format, receiver and transmitter baud rates, and enables receiver interrupts. The first parameter <port #> selects the console port in ranges from 1 to 15. This corresponds to character input buffers defined in the VMEPROM system RAM (SYRAM). If a minus (-) precedes the port number, then the associated port # is stored in the UNIT 2 (U2P$(A6)) variable. Receiver and transmitter baud rates are initialized to the same value according to the <baud rate> parameter. The <baud rate> parameter ranges from 0 to 8 or the corresponding baud rates of 19200, 9600, 4800, 2400, 1200, 600, 300, 110, or 38400. Either parameter type is acceptable.

**Baud Rates Allowed:**

```
0  =  19200 baud
1  =  9600 baud
2  =  4800 baud
3  =  2400 baud
4  =  1200 baud
5  =  600 baud
6  =  300 baud
7  =  110 baud
8  =  38400 baud
```

The <type> and <UART base addr> are optionally included when binding a logical port to a different UART. For <type> information, refer to the User's Manual of your CPU-board. The <port #> can also be used to set or reset the port flags. These are bit positions 8 through 15 of the resulting integer value and are defined to the right. It is recommended that hex format be used when setting these parameters.

```
$100 + port  = CtrlS CtrlQ protocol
$200 + port  = Pass control characters
$400 + port  = DTR protocol
$800 + port  = 8-bit character I/O
$1000 + port = receiver interrupts disable
$2000 + port = even parity enable
$4000 + port = clear flag bits
```

If the BP command has no arguments, a listing of all currently installed ports is sent to the console. ‘Task’ parameter indicates the currently assigned task to that port.

**Example:**

```
? BP
Port Type fwpi8dcs Base Baud task
# 1 1 00001100 FF800000 9600 1

? BP 2,1,1,$FF800200  Initialize the UART
?
```
1.2.10  BR - Set/Display/Delete Breakpoints

Format:

- BR
- BR *
- BR <number>
- BR <number>,<address>
- BR <number>,<address>,<command>
- BR <number>,<address>,[<command>],<count>

VMEPROM supports a maximum of 10 breakpoints in the range 0-9. The BR command is used to set, display or delete breakpoints.

The first format displays all currently defined breakpoints. The second one deletes all defined breakpoints. The third format is used to delete one single breakpoint. The other formats are used to define one breakpoint. If a breakpoint is already defined it will be overwritten. Two breakpoints looking for the same address are not possible.

If a count is specified, the program first stops at the breakpoint when this specification has been achieved. The default value is one.

The default action taken by a breakpoint is a display of the breakpoint number encountered and a display of all processor registers.

So there is a fourth option of the command line to change the default behavior at a breakpoint. The command, which can be specified is executed instead of the display described before. The command may not have any arguments and may have a length of up to 9 characters.

The command may be a symbolic name, one of the built-in commands of VMEPROM, an installed utility or a disk file (command file or program).

Example:

? BR 0 8020 Define breakpoint 0 at address $8020

? BR
Defined Breakpoints:
  B0  $8020  1

?
1.2.11 BS - Block Search

Format: BS <begin>,<end>,[/]<value>[,<option>]
        BS <begin>,<end>,[/]<pattern>,P
        BS <begin>,<end>,[/]<opcode>,O

This command searches the specified memory area for a constant. The type of the constant is defined by the option and may be a Byte, Word, Long word, Pattern, or an Opcode. A pattern is an ASCII string which is to be put in inverted commas. The maximum length is only restricted by the length of the input line, which may not exceed 78 characters. An Opcode is each valid 680x0 mnemonic or an opcode as described in Chapter 4 of the VMEPROM Manual. If the pattern or the opcode contains argument separators, such as space, comma, or full stop, the data has to be put in brackets. If no option is specified, a default of Word is assumed.

The data which has to be searched in memory may be preceded by a '/' to look only for locations not containing the value, pattern or opcode.

Example:

? BS 8000 8100 /1234 Search memory for "not" value
Search: 8020 = 5678 Found

? BS 8000 8100 5678 Search memory for value
Search: 8020 = 5678 Found

? BS 8000 8100 ("Hello World") P Search memory for pattern
None found

? BS 8000 8100 (ADDQ.L #1,D0) O Search memory for opcode
None found

? _
1.2.12 BT - Block Test

Format: BT <begin>,<end>

The Block Test command performs an in-depth memory test within the specified address limits. The following passes are performed:

1) Byte Pattern Test
2) Word Pattern Test
3) Long Pattern Test
4) Word Shift Test
5) Address Test

If any errors are found they are reported with the type of test which failed, the address and the differing values. In addition the error counter in the task control block (TCB) is incremented.

Example:

? bt 200000 300000 Test memory from $200000 to $300000
?

3-19
1.2.13 BV - Block Verify

Format: BV <begin>,<end>,<destination>

This command compares two blocks of memory. If the specified blocks are not equal, the different values and the memory location is displayed. In addition the error counter in the task control block (TCB) is incremented.

Example:

? bv 8000 8080 8080
Verify: 8021 = 70 80A1 = 71

?
1.2.14 COLD - Cold Start VMEPROM

Format: COLD

The COLD command is used to reinitialize all VMEPROM variables. It takes the same action as a reset, except that the kernel and all associated tasks are not affected.

Example:

? COLD
******************************************************************************
*                                                                   *
*  V M E P R O M                                                   *
*                                                                   *
*  SYS68K/IBC-20  Version a.bb  Date                               *
*                                                                   *
* (c) FORCE Computers and Eyring Research                          *
*                                                                   *
******************************************************************************

? _
1.2.15 CREATE TASK

Format:  
CT <command>,<size>,<time*256+prior>,<port>  
CT ,<size>,<time*256+prior>,<port>  
CT <address>,<size>,<time*256+prior>,<port>

The CREATE TASK command places a new task entry in the task queue and lists the realtime kernel of VMEPROM. Parameters for the new task include a command line, memory size, task priority/time slice, and an I/O port. The new task number is reported after task creation.

The <command> parameter is the command line for the new task. The string is passed to the new task via a message buffer and cannot exceed 64 characters in length.

Multiple commands and parameters are passed by using parentheses. If the first parameter is omitted, then the VMEPROM monitor is invoked.

If an address is specified instead of <command>, this address is interpreted as the start address of a program in memory. The address must be specified in hexadecimal and start with a number 0-9 not to conflict with a program name.

The amount of memory for the new task is given by <size> and is in 1 Kbyte increments (although rounded to the next 2 Kbyte boundary). The minimum amount of memory is 8 Kbyte. The system memory bit map is searched for a contiguous block of memory equal to <size>. If the search fails to find a large enough block, then memory is taken from the parent task and allocated to the new task.

The <priority> parameter specifies the new tasks priority. The range of task priority is from 1 to 255 where 255 is the highest priority. The highest priority, ready task always executes. Tasks on the same priority level are scheduled in a round robin fashion. The time a task is in running state is also given with the <priority> parameter. If no time is specified it will default to one time slice. Otherwise it is calculated to "time*256+priority".

The <port> parameter assigns an I/O port to the new task. Port 0 is the default and is called the phantom port. On the phantom port, all character outputs and conditional inputs are ignored while requests for character input result in the task aborting with error 86. More than one task may be assigned to an output port. The input port cannot be shared with another task. Input ports are allocated on a first come basis.

After a task is created, the spawned task number is reported. This number is used in killing the new task.

The values for size, priority and port have to be entered in decimal.
### Example:

```
? lt

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>256</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00047000</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
</tbody>
</table>

? ct ,10,64,2
    Sontask number = 3

? lt

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>246</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00044800</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
<tr>
<td>3/0</td>
<td>64</td>
<td>1</td>
<td>98</td>
<td>10</td>
<td>FFE06F50</td>
<td>00044800</td>
<td>00047000</td>
<td>2/2/0/0/0</td>
<td></td>
</tr>
</tbody>
</table>

? ct ,,256*10+40,3
    Sontask number = 4

? lt

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>238</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00042800</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
<tr>
<td>3/0</td>
<td>64</td>
<td>1</td>
<td>98</td>
<td>10</td>
<td>FFE06F50</td>
<td>00044800</td>
<td>00047000</td>
<td>2/2/0/0/0</td>
<td></td>
</tr>
<tr>
<td>4/0</td>
<td>40</td>
<td>10</td>
<td>99</td>
<td>8</td>
<td>FFE06F50</td>
<td>00042800</td>
<td>00044800</td>
<td>3/3/0/0/0</td>
<td></td>
</tr>
</tbody>
</table>
```

? _
1.2.16 DI - Disassembler

Format:  
DI <address>
DI <address>,<count>

The DI command causes the disassembler to be invoked and display the mnemonic, starting at the specified address. If count is specified, it is interpreted as the number of lines to display. If count is omitted, a full page is displayed on the terminal and the user is then prompted to continue disassembly (enter <cr>) or to return to the command interpreter (enter any other key).

The disassembler supports all 68020 mnemonics.

Example:

? DI 8000 5

8000  NOP
8002  NOP
8004  NOP
8006  NOP
8008  NOP

?
1.2.17 DR - Display Processor Registers

Format: DR [T]

The DR command displays processor registers. The displayed registers are not real current processor registers, but those kept in memory and loaded to the processor when a program is started. When program execution is terminated (XEXT instruction, trap or breakpoint or other exception) the processor registers are resaved and can be displayed by the DR command.

When choosing the option 'T', only the program counter, stack pointer, and address registers A5 and A6 will be displayed until 'T' is used a second time. Then all registers will once again be displayed. First VMEPROM is configured to display all registers.

Example:

```
?DR
D: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A: 00000000 00000000 00000000 00000000 00001000 00007000 0009AFFC
   VBR = 00000000  CAAR = 00000000  CACR = 00000001  SFC = 0  DFC = 0
   *USP = 0009AFFC  SSP = 00007BE6  MSP = 000078C4
   PC = 00008000  SR = 0000 .U..0........

?DRT
PC = 00008000  SP = 0009AFFC  A6 = 00007000  A5 = 00001000

?DR
PC = 00008000  SP = 0009AFFC  A6 = 00007000  A5 = 00001000

?DRT
D: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A: 00000000 00000000 00000000 00000000 00001000 00007000 0009AFFC
   VBR = 00000000  CAAR = 00000000  CACR = 00000001  SFC = 0  DFC = 0
   *USP = 0009AFFC  SSP = 00007BE6  MSP = 000078C4
   PC = 00008000  SR = 0000 .U..0........

?_ 
```
1.2.18 DT - DATE AND TIME

Format: DT

The DT command outputs the current date and time to the user console. These values can be changed by the ID command.

Example:

? DT
16-Mar-88
16:47:38

?
1.2.19 DU - Dump S-record

Format:       DU <begin>,<end>
        DU <begin>,<end>,<command line>

This command sends an S-Record to the standard output port. It may be redirected with the usual redirection method.

An optional command line may be specified which is sent via the output port before the S-record starts. This can be used to start a load command on the host system.

The following S-record types are supported:

S1       Start record
S2       Data record, this type is needed if the end address is smaller than $8000.
S3       Data record, this type is used if the end address is bigger than $800000.
S7       End-record for S3 records.
S8       End-record for S2 records.
S9       End-record for S1 records.

The address field of all End-records is 0.

Example:

? DU 8000 8020
S0030000FC
S2180080004E714E714E714E714E714E714E714E714E714E714E714F1
S2100080144E714E714E714E714E714E714E714E714E714E714E714E1
S804000000FB

? DU 8000 8020 >2
?

?
1.2.20 EAGLE - Display All Information About Available EAGLE Modules

Format: EAGLE

This command is intended to display all information about available EAGLE modules on the IBC-20 board. The information displayed by this command is contained in the ID EPROM of an EAGLE module and the content of such an ID EPROM has a fixed structure which has been specified by FORCE COMPUTERS.

However, the EAGLE command provides the following information about an EAGLE module.

- The EAGLE module identifier
- The name of the manufacturer who had developed the EAGLE module.
- The serial number of the board, the hardware and software revision number.
- Up to eight Ethernet addresses (these addresses are always displayed independent from the presence of a corresponding LAN controller).
- Detailed information about available software modules relating to different operating systems.
- Detailed information about the available devices on the EAGLE modules (base addresses, interrupt and DMA capabilities, logical device number, etc.).
1.2.21 ER - LIST ERRORS

Format:  
- ER [-c]  
- ER 0 [-c]  
- ER <error#>

The LIST ERROR command has three functions. The first one, with no argument, displays the number of errors found on one of the following commands:

1) Block Test  
2) Block Verify  
3) Block Search.

The second format, with the argument "0" resets the above error count to 0.

If the optional parameter [-c] is given when using the first two formats, an execution count will be displayed or reset to zero. The execution count will be incremented before it is displayed.

The third format requires a valid error number as an argument and displays the VMEPROM error message associated with <error#>.

Error numbers range as follows:

- VMEPROM errors 1-49  
- PDOS errors 50-99  
- Disk errors 100-299

Example:

? ER  
Current error count = 6

? er 0

? er 2  
Command line argument error

?er 0 -c

?er -c  
Current error count = 0  Execution count = 1
1.2.22 EV - SET/RESET EVENT

Format:

EV
EV {-|+}<event>
EV {-|+},<address>,<bit#>

VMEPROM events are set, reset, or listed with the EV command. Both logical and physical events can be accessed with EV. The delayed event queue can also be listed or cleared with the EV command.

If the first parameter is zero, the delay queue is cleared. For accessing a logical event, the event number <event> has to be entered. If <event> is proceeded by a plus (+) sign, the event is set and the old status is returned. If <event> is proceeded by a minus (-) sign, the specified event is cleared and its old status is displayed. For accessing a physical event, the second parameter must be the byte address followed by the bit number (0-7), where bit 7 is the most significant bit of the byte. Physical events are set (+), reset(-) and list( _) in the same way as logical events are accessed. If no special sign is specified, the current status of the event is displayed. If <event> is omitted, a status list of all events in the system and all pending delay events are displayed.

The event number has to be entered in decimal.

Current logical event definitions are as follows:

1-63 = Software events
64-80 = Software resetting events
81-95 = Output port events
96-111 = Input port events
112 = 1/5 second event
113 = 1 second event
114 = 10 second event
115 = 20 second event
116 = Reserved
117 = Reserved
118 = Reserved
119 = Reserved
120 = Level 2 lock
121 = Level 3 lock
122 = Batch event
123 = Spooler event
124 = Reserved
125 = Reserved
126 = Reserved
127 = Virtual ports
128 = Local event
Example:

? EV
  00000000 00000000 00000000 0000FE00
  EV 128 : TASK 0  SET DELAY = 43 TICS

? EV 10
  Is 0

? EV +10
  Was 0

? EV -10
  Was 1

? EV 10
  Is 0

? EV +,$10000,1
  Was 0

? EV, $10000,1
  Is 1
### 1.2.23 FGA - Change Boot Setup for Gate Array

**Format:** FGA

Some registers of the gate array are definable by the user. The contents of this register is stored in the on-board battery SRAM in a short form.

The boot software for the gate array will take these values after reset to initialize the gate array. The FGA command may be used to enter an interactive mode for changing this boot table in the battery SRAM.

The FGA command will show the actual value stored in the battery SRAM. To change any value, a new one has to be entered in binary form. If only a `<cr>` is entered, no change will be made. To step backward a minus has to be entered. If a `<>` or `<ESC>` is given, the FGA command returns to the shell.

**Example:**

```
? FGA

> > > Setup for FGA-002 BOOTER < < <

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>FGA Offset</th>
<th>Value in SRAM</th>
<th>Changed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>$0420</td>
<td>%00011110</td>
<td>%00011110</td>
</tr>
<tr>
<td>CTL_01</td>
<td>$0238</td>
<td>%00000100</td>
<td>%00000100</td>
</tr>
<tr>
<td>CTL_02</td>
<td>$023C</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>CTL_05</td>
<td>$0264</td>
<td>%00001110</td>
<td>%00001110</td>
</tr>
<tr>
<td>CTL_12</td>
<td>$032C</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>CTL_14</td>
<td>$0354</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>CTL_15</td>
<td>$0358</td>
<td>%01001100</td>
<td>%01001100</td>
</tr>
<tr>
<td>CTL_16</td>
<td>$035C</td>
<td>%00100000</td>
<td>%00100000</td>
</tr>
<tr>
<td>MBX_00</td>
<td>$0000</td>
<td>%00000000</td>
<td>%00000001</td>
</tr>
<tr>
<td>MBX_01</td>
<td>$0004</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>MBX_02</td>
<td>$0008</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>MBX_03</td>
<td>$000C</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>MBX_04</td>
<td>$0010</td>
<td>%00000000</td>
<td>%00000000</td>
</tr>
<tr>
<td>MBX_05</td>
<td>$0014</td>
<td>%00000000</td>
<td></td>
</tr>
<tr>
<td>MBX_06</td>
<td>$0018</td>
<td>%00000000</td>
<td></td>
</tr>
<tr>
<td>MBX_07</td>
<td>$001C</td>
<td>%00000000</td>
<td></td>
</tr>
</tbody>
</table>
```

---

3-32
1.2.24 FM - FREE MEMORY

Format:        FM
              FM -E
              FM {-}<size>

The FREE MEMORY command drops memory from your current task.

If no parameter is given all free memory contiguous to tasking memory is displayed.

If parameter '-E' is given all free memory is displayed. This includes memory which is not contiguous
to tasking memory but deallocated in the memory bit map.

If the <size> parameter is positive, then the memory is deallocated and made available to the system
for other task usage. If the <size> parameter is negative, then the memory is simply dropped from the
current task and is not recoverable. The size parameter must be entered in decimal.

Example:

? FM
No free memory contiguous to tasking memory

? FM -E
Free memory:  2 kbyte at $B6000

? FM 100
100 Kbytes free at address $9C800

? FM
Free memory:  100 Kbyte

? FM -10
10 Kbytes free at address $9A000

? FM
No free memory contiguous to tasking memory

? FM -E
Free memory:  100 kbyte at $9C800
Free memory:  2 kbyte at $B6000

? _
1.2.25 FMB - Force Message Broadcast

Format:  
FMB <slotlist>,<FMB channel>,<message>  
FMB [<FMB channel>]

The FMB command allows sending a byte message to individual slots in the backplane, broadcast to all the boards, and getting a pending message.

The first format is used to send a message. With this the first parameter is used to select the slots to which a message should be sent. Each slot number can be separated with a '/' sign; a '-' defines a range of slot numbers. Slot numbers can range from 0 to 21. A slot number of 0 sends the message to all slots. The second parameter defines which FMB channel should be used. It can be '0' or '1'. The message is the byte to be deposited into the FMB channel(s).

The second format is used to get messages. If no parameter is given, one message of each FMB channel is fetched and displayed. If a channel is specified only this channel is addressed and the message will be displayed.

Example:

? FMB
FMB channel 0 is empty
FMB channel 1 is empty

? FMB 1-21,0,$EF

? FMB 1-21,1,%10100001

? FMB
FMB channel 0 = $EF
FMB channel 1 = $A1

? FMB 1-21,1,$77

? FMB 1
FMB channel 1 = $77

? FMB 1/2/5/7-19/21,0,$1

? _
1.2.26 FUNCTIONAL - Perform Functional Test

Format:    FUNCTIONAL

NOTE:     This command is not designed for the user, but instead for internal purposes by FORCE COMPUTERS.
1.2.27 GO - Start User Program

Format:

\[ \text{G} \]
\[ \text{G } \text{<address>} \]
\[ \text{GO} \]
\[ \text{GO } \text{<address>} \]

A user program in memory is started with this command. The start address may be specified on the command line, or the value of the program counter, as displayed by the DR command, is taken if this field is omitted.

The following actions are taken by VMEPROM if this command is specified:

1) The processor registers are loaded with the user values.
2) The first instruction is executed.
3) If any breakpoints are defined, they are inserted in the user program.
4) The program is continued at the second instruction.

Example:

? G 8000
>>> This is a Test <<<

?
1.2.28 GD - Start User Program Without Breakpoints

**Format:**

```
GD
GD <address>
```

The GD command takes the same actions as the G or GO command, except that defined breakpoints are ignored and not inserted in the user program.

*Example:*

```
? GD 8000
>>> This is a Test <<<

?
```
1.2.29 GM - GET MEMORY

Format:          GM
                GM <size>

The GM command adds memory to the current task. The amount of memory is specified by
<size>. The <size> parameter has to be given in decimal. If no parameter follows GM, then all
of the available memory is added. No error is reported if the memory request cannot be met.

Example:

? FM
No free memory contiguous to tasking memory

? FM 20
20 Kbytes free at address $00071800

? GM
? FM
No free memory contiguous to tasking memory

?
1.2.30 GT - Start User Program with Temporary Breakpoint

Format:  

- GT <breakpoint>
- GT <breakpoint>,<address>
- GT <breakpoint>,<address>,<command>
- GT <breakpoint>,<address>,<command>,<count>

This is almost the same function as the G or GO command, except that an additional temporary breakpoint is inserted. This breakpoint is automatically removed if the program counter reaches this breakpoint.

If a command is given, it will be executed at the breakpoint. Otherwise all processor registers are displayed.

If a count is specified, the program first stops at the breakpoint when this specification has been achieved. The default value is one.

Example:

? GT 10020 10000
At temporary breakpoint

0 1  2  3 4  5  6  7
D: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

VBR = 00000000 CAAR = 00000000 CACR = 00000001 SFC = 0 DFC = 0
*USP = 00099FFC SSP = 0007BDE MSP = 000078C4
PC = 00010020 SR = 0000 ..U..0........

? GT 10020 10000 lt

? GT 10020,10000,,2
At temporary breakpoint

0 1  2  3 4  5  6  7
D: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

VBR = 00000000 CAAR = 00000000 CACR = 00000001 SFC = 0 DFC = 0
*USP = 00099FFC SSP = 0007BDE MSP = 000078C4
PC = 00010020 SR = 0000 ..U..0........
### 1.2.31 HELP - HELP

**Format:**

```
HELP
HELP <command>
```

The HELP command first displays a short description of all VMEPROM built-in commands on the terminal. Then a more detailed description of all commands is displayed.

After every screen full, the output stops. It may be continued by entering a `<cr>`. Control is transferred back to the command interpreter on any key other than `<cr>`.

If HELP is followed by a command name, a short description of this command is displayed.

If HELP is followed by one or more characters, but not a complete command name, a start description of all commands matching with the given character is displayed.

#### Example:

? HE M
M <address>[B|W|L&N&O|E|F#] Modify memory contents
MD <address>[,<count>] Display memory in Hex and ASCII
MEM [16|32] Set data bus width
MM <address>[B|W|L&N&O|E|F#] Alias for M command
MS <address>,<data|"string"> Preset memory with constant or string

? _
1.2.32 HIST - Command history

Format: HIST

The HIST command is used to show which commands can be recalled with [CTRL-A]. This is an easy way to check if a command is inside the alternate command line buffer. If it is, recalling the line is possible and it need not to be written a second time.

Example:

? HIST
BT 10000 20000
DR
BT 200000 300000

? [CTRL-A]
BT 10000 20000[CTRL-A]
DR[CTRL-A]
BT 200000 300000<cr>

? _
1.2.33 ID - SET SYSTEM DATE/TIME

Format: ID

The SET SYSTEM DATE/TIME command displays the VMEPROM header and prompts for the date and time. The header shows the version of VMEPROM and the used CPU-type as displayed after reset.

The date can be entered in either a day, ASCII month, year form or numeric month, day, year.

Any delimiter can be used to separate date and time parameters. Pressing [CR] leaves the old date and time.

Example:

? ID

******************************************************************************
* *
* V M E P R O M *
* *
* SYS68K/IBC-20 Version a.bb Date *
* *
* (c) FORCE Computers and Eyring Research *
* *
******************************************************************************

Date: 17-Aug-89 <cr>
Time: 18:45:21 <cr>

? _
1.2.34 INFO - Information about the CPU board

Format: INFO

The INFO command is used to obtain information about the board. The output is strongly dependent on the used board.

These outputs are given at all board types:

1) Board type.

2) VMEPROM Version and it’s start address.

3) EPROM base address.

4) I/O devices: Depending on the board type all I/O devices are listed including their base address.

5) RAM addresses SYRAM start address.
   Current tasks task control block start address.

Additional information may occur.

Example:

? INFO
FORCE IBC-20
VMEPROM Version a.bb at $FFE0484A

EPROM base addresses:
    System EPROM at $FFE00000; EEPROM at $FFC80000
    Boot EPROM at $FFE00000

I/O Devices:
    BIM at $FF803E00; RTC at $FF803000
    CIO at $FF800C00; SCC channel A at $FF802020

RAM addresses:
    Local RAM $0 to $001FFFFF; SRAM at $FFC00000
    SYRAM at $00001000; TCB at $00007000

? _
1.2.35 KM - KILL MESSAGE

Format:    KM
           KM <task #>

The KM command removes all task messages associated with <task #> from the message buffers.

If no task is specified, then all messages associated with the current task are deleted from the message buffers.

See also SEND MESSAGE.
1.2.36 KT - KILL TASK

Format:  
\[ \text{KT} \]  
\[ \text{KT} \{-\}<\text{task #}> \]

The KILL TASK command removes a task from the task list and returns the task's memory to the free pool for use by other tasks. Only your current task or a task spawned by your task can be killed. (Task 0 can kill any task except itself or a task that is kill protected.)

Each task is assigned a unique task number which is shown by the LIST TASK command. Only the current task (indicated by '*) or those spawned by the current task (indicated by current task number following a '/' character) may be killed. Task #0 is the system task and cannot be killed.

If a minus sign (-) precedes the task number, then the task's memory is not deallocated to the memory bit map. If the task number is zero, then the current task is killed without deallocating memory.

If no parameter is given, then the current task is killed and memory is deallocated.

All open files associated with the killed task are closed by the KT command.

Example:

```
? lt

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>238</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00042800</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
<tr>
<td>3/0</td>
<td>64</td>
<td>1</td>
<td>98</td>
<td>10</td>
<td>FFE06F50</td>
<td>00044800</td>
<td>00047000</td>
<td>2/2/0/0/0</td>
<td></td>
</tr>
<tr>
<td>4/0</td>
<td>40</td>
<td>10</td>
<td>99</td>
<td>8</td>
<td>FFE06F50</td>
<td>00042800</td>
<td>00044800</td>
<td>3/3/0/0/0</td>
<td></td>
</tr>
</tbody>
</table>

? kt 3

? lt

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>238</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00042800</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
<tr>
<td>4/0</td>
<td>40</td>
<td>10</td>
<td>99</td>
<td>8</td>
<td>FFE06F50</td>
<td>00042800</td>
<td>00044800</td>
<td>3/3/0/0/0</td>
<td></td>
</tr>
</tbody>
</table>

? _
```
1.2.37 LO - Load S-record

**Format:**

```
LO
LO <address>, <command line>,<-V|-T>
```

The LO command loads a S-record into memory from a standard input port. Normal I/O redirection may be used for input from other ports. The starting load address is optionally specified by `<address>`.

An optional command line may be specified which is sent to the host before S-record loading starts. It can be used to initiate a host system download without using the TM Command.

Two possible options exist which must be proceeded by a minus sign. If option V is given, the contents of the S-records will only be compared with contents of those memory locations which are to be loaded. The different values of the memory locations and the S-record data are displayed. If option T is given without an address parameter, the S-records are loaded immediately following the TCB. The following S-record types are supported by VMEPROM:

- **S0** Start record, ignored by VMEPROM and may be omitted.
- **S1** Data record with 16 bit address field
- **S2** Data record with 24 bit address field
- **S3** Data record with 32 bit address field
- **S7** End record with 32 bit address field
- **S8** End record with 24 bit address field
- **S9** End record with 16 bit address field

If the address for the LO command is specified on the command line, address fields in the data records are ignored and the S-record is loaded contiguously from the specified address upwards.

If the end record address field is equal, 0 control is transferred back to the VMEPROM command interpreter. If the address file holds an address, VMEPROM automatically executes a "G address" command after the S-record is loaded and an end record is found. Because of the "G" command all breakpoints which are defined are inserted in the program.

See also **DU - Dump S-records**

*Example:*

? lo <2 8800

?
1.2.38 LT - LIST TASKS

Format: LT

The LT command displays all tasks currently in the task list to the console. Task 0 is the system task and is created automatically during system initialization. This task cannot be killed.

Your current task is indicated by an '*' preceding the task number. Following the task number is a slash and the parent task number. Subsequent data provides the current status of each task and is defined as follows:

task {*=current} Task #/parent task #

pri Task priority (1-255)

tm Time slice (1-255)

ev1/ev2 Suspended event(s)

size Task size (k bytes)

pc Current program counter. If the task is in suspended state or ready state the program counter points to the first opcode this task will execute after the task is moved to run state.

tcb Task control block

eom End of memory

ports Task I/O ports in the following order: input port/output port/Unit 2 port/Unit 4 port/Unit 8 port

name The name of the command currently executing

Example:

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0/0</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>FFE1D4A</td>
<td>00007000</td>
<td>0004700</td>
<td>1/1/0/0/0</td>
<td>lt</td>
<td></td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
<td></td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000503AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
<td></td>
</tr>
</tbody>
</table>

? _
1.2.39 M - Modify Memory

Format: \texttt{M \langle address\rangle[,\langle option\rangle]} \linebreak \texttt{MM \langle address\rangle[,\langle option\rangle]}

Option is \texttt{B | W | L & N & O | E | Fx}

The Modify Memory command is used to inspect and change memory locations. Several options are allowed on the command line to specify the size of the memory and the access type. The following options are allowed:

\begin{itemize}
  \item \texttt{B} memory is byte sized (8 bits).
  \item \texttt{W} memory is word sized (16 bits). This is the default.
  \item \texttt{L} memory is long word sized (32 bits).
  \item \texttt{O} memory is byte sized and on odd addresses only.
  \item \texttt{E} memory is byte sized and on even addresses only.
  \item \texttt{N} memory is write only, the current contents is not displayed.
  \item \texttt{Fx} specifies the 68020 function code signals which should be driven from the 68020 to perform the read/write. The default value is 1. Possible values are:
    \begin{itemize}
      \item 1: User Data Space
      \item 2: User Program Space
      \item 5: Supervisor Data Space
      \item 6: Supervisor Program Space
      \item 7: CPU Space
    \end{itemize}
\end{itemize}

The Odd and Even options are overriding the B/W/L options. The N (no read) option has to be specified after the size qualifier and after the Odd/Even specification. All memory accesses check that the write access was successful by performing a read after the write unless N is specified. If the data written and the data read do not match, the command is terminated and an error message is displayed. The memory modify command supports a number of sub-commands, which can be entered instead of a new memory value. These sub-commands do not change the access option specified on the command line. The following sub-commands are supported:

\begin{itemize}
  \item \texttt{<cr>} open next location
  \item \texttt{=} open same location again
  \item \texttt{-} open previous location
  \item \texttt{-<count>} go back \texttt{<count>} bytes
  \item \texttt{+} open next location
  \item \texttt{+<count>} go forward \texttt{<count>} bytes
  \item \texttt{#<address>} open new absolute address
  \item \texttt{?<mnemonic>} insert 68000 opcode at current address
  \item \texttt{.} exit to the command interpreter
\end{itemize}
Example:

? M 8000
8000 4246 : <cr>
8002 1C2E : <cr>
8004 0441 : <cr>
8006 4247 : ?nop<cr>
8008 A05A : -2<cr>
8006 4E71 : -<cr>
8004 0441 : #8000<cr>
8000 4246 : <cr>
8002 1C2E : .
? M $10000, F5B

10000 48 : .

?_
1.2.40 MD - Display Memory

Format:  

MD <address>
MD <address>[,<count>]

The MD command displays the memory contents of the specified address. The data is displayed in hex and ASCII representation, 16 bytes on every line. If the hex value cannot be displayed in ASCII representation, a full stop (".") is displayed instead.

If no count is specified on the command line, the Display Memory command displays 16 lines, representing 256 bytes of data, and prompts the user to display more or to return to the command interpreter.

If a carriage return (<cr>) is entered, the next 256 bytes are displayed. Any other character returns control back to the command interpreter of VMEPROM.

If a count is specified on the command line, the value is interpreted as the number of bytes to be displayed. All values are assumed to be in hex.

If a base is specified with the BASE command this value is printed at the first line which is put out.

Example:

? MD 8000 30
00008000: 08 98 00 00 04 88 00 01 00 80 00 08 40 08 00 80 ............@...
00008010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008020: 40 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @........@...

? BASE 2000

? MD 8000 30
00002000: 08 98 00 00 04 88 00 01 00 80 00 08 40 08 00 80 ............@...
00008010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008020: 40 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @........@...

? _
1.2.41 **MEM - Set Data Bus Width of the VMEbus**

**Format:**

MEM
MEM 16
MEM 32

This command can display or set the data bus width of the CPU board on the VMEbus. If no argument is entered, the current data bus width is displayed. If an argument of '16' or '32' is given, the data bus width is set to 16 or 32 bits respectively.

*Example:*

? MEM<cr>
Data bus width is set to 32 bits

? MEM 16<cr>

? MEM<cr>
Data bus width is set to 16 bits

? MEM 32<cr>

? _
1.2.42 MS - Set Memory to Constant or String

Format: MS <address>,<data|"string">

This command writes the specified data pattern to memory. The data may consist of hex numbers and ASCII data in any combination. The ASCII data must be put in inverted commas.

Example:

? BF 8000 8100 8377 B

? MD 8000 20
00008000: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FFFF
1.2.43 PROG - Program FLASH EPROM

**Format:** \[\text{PROG} \ [\text{<source>}, \text{<destination>}, \text{<length>}, \text{<width>}]\]\\]

This command is used to program FLASH EPROMs. All parameters may be specified on the command line or may be entered interactively after the function has been invoked.

The first parameter \text{<source>} is the start address of the data which is to program into the FLASH EPROM.

The second parameter \text{<destination>} represents the base address of the FLASH EPROM.

The third parameter \text{<length>} specifies the length of the FLASH EPROM. If 0 is entered the length and width is automatically calculated.

The fourth parameter \text{<width>} selects the data width of the FLASH EPROMs. Three values are possible:

- '1': Byte width (8-bit)
- '2': Word width (16-bit)
- '4': Long width (32-bit)

Please note that the FLASH EPROM(s) must be completely programmed. Therefore programming only parts of a FLASH EPROM is not possible.

**Example:**

? PROG $100000 $FFC80000 0
programming......
FLASH EPROM successfully programmed

? PROG
Source base address = $40800000
FLASH EPROM base address = $FFC80000
Source length (0 for automatic select) = $20000
Width (1,2 or 4) = 1
programming......
FLASH EPROM successfully programmed

?_
1.2.44 PROMPT - CHANGE PROMPT SIGN

Format: PROMPT [<data|"string">]

The PROMPT command is used to change the prompt for the current task in the used specified pattern.

The data may consist of hex numbers and ASCII data in any combination. The ASCII data must be put in inverted commas.

If no parameter is given, the default VMEPROM prompt "?" will occur. The user defined prompt sign will be truncated to nine characters maximum.

Example:

? PROMPT "#"_
#_

#PROMPT ("HELLO> ")_
HELLO> _

HELLO> PROMPT_
?_
1.2.45 RM - Modify Processor Registers

Format:  
RM  
RM <register>  
RM <register>,<value>

The RM command modifies the processor registers or, if available, the data registers of the 68881 coprocessor. Three modes are allowed.

The first mode is an interactive mode, which scans all registers. For each register, the current value is displayed and the user is prompted to enter a new value. A <cr> leaves the register unchanged. After a new value or a <cr> has been entered, the same procedure will be started for the next register. If an <ESC> or <.> has been entered, control is transferred back to the command interpreter.

The second mode makes it possible to change only one specified register. The current value is then displayed and the user is prompted to enter a new value. A <cr> leaves the register unchanged. After a new value or a <cr> has been entered, control is transferred back to the command interpreter.

The third mode allows the specification of the new value for the given register on the command line and does not display the the old value.

The following registers may be modified by the user:

- **VBR**: Vector base register, only on 68010/68020/68030 systems
- **SFC/DFC**: Source and Destination function code register
- **CAAR**: CACHE address register, only for 68020/68030 systems
- **CACR**: CACHE control register, only for 68020/68030 systems
- **PC**: Program counter
- **SR**: Status register
- **USP**: User Stack pointer
- **SSP**: System Stack pointer
- **MSP**: Master Stack pointer, only on 68020/68030 systems
- **D0-D7**: Data registers D0-D7
- **A0-A7**: Address registers A0-A7, where A7 is the current stack pointer as defined by the status register

**Caution:** Be careful when modifying the Vector Base register (VBR) as VMEPROM is an interrupt driven system and any modifications to this register may crash the system.
Example:

? RM D0
D0 = 00000000 : 12345678<cr>

? RM D1 1000

? DR

<table>
<thead>
<tr>
<th>D: 12345678 00001000 00000000 00000000 00000000 00000000 00000000 00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 00000000 00000000 00000000 00000000 00000000 00001000 00007000 0009CFFC</td>
</tr>
</tbody>
</table>

VBR = 00000000 CAAR = 00000000 CACR = 00000001 SFC = 0 DFC = 0
*USP = 0009CFFC SSP = 00007BE6 MSP = 000078C4
PC = 00008000 SR = 0000 ..U..0........

? _
### 1.2.46 RR2 - EPROM Programming

**Format:**

RR2 [<f>,<file>],<board>,<mode>,<option>
RR2 [<m>,<addr>,<cnt>],<board>,<m>,<opt>

The RR2 command is used for programming EPROMS or EPROMS on a SYS68K/RR-2/RR-3 board. It can also be used to transfer files or memory contents into a SRAM area on the RR_2 or to load EPROM/EEPROM contents into the VMEPROM memory.

The following are examples on the usage of the RR2 command:

- ? RR2 F,FILENAME,RR_2_ADDRESS,MODE,OPTION
  - if the source is a disk file, or
- ? RR2 M,STRTADDR,BYTECNT,RR_2_ADDRESS,MODE,OPTION
  - if the source is in memory.

The following describes the parameters:

- **F,FILENAME** ............ source = disk file
  - F = source flag
  - FILENAME = the name of the source file
- **M,STRTADDR,BYTECNT**..... source = memory
  - M = source flag
  - STARTADDR = source start address
  - BYTECNT = source length in bytes
- **RR_2_ADDR** ............... the address of the RR_2 bank
- **MODE** .....................
  - 1 = 8 bit mode (single EPROM)
  - 2 = 16 bit mode (two EPROMS)
  - 4 = 32 bit mode (four EPROMS)
- **OPTION** ..................
  - P = program an EPROM (includes E and V and a bit test)
  - E = check if EPROM is empty
  - V = verify source and EPROM contents
  - L = load EPROM contents to memory

For further information on the hardware setup of the SYS68K/RR2 or SYS68K/RR3 board please refer to the user’s manual of the RR-2/3 board.
Example:

? RR2 M,$0,$8000,$800000,2,E

executes an empty check in word mode for EPROM type 27128 (16k x 8) at RR_2 address $800000. The M - source flag and the memory address are dummy.

? RR2 F,PROG/2,$800000,4,P

programs EPROMS at address $800000 in 32-bit mode with the source file PROG from disk 2.

? RR2 M,$10000,$2000,$800000,1,L

loads the contents of an 8k x 8 EPROM at address $800000 into the memory to address $10000.
SYS68K/RR-2/RR-3 board configuration:

This example contains the RR-2 board configuration and the program usage for 27128 EPROMs in the 16 bit mode.

**Jumper settings for 16k x 8 EPROMs on bank 2 (TOSHIBA 27128):**

- **B1b**
  - Read time selection on bank 2
  - 8
  - 5
  - 250 ns
  - 1 4

- **B2b**
  - Write time selection on bank 2
  - 3 15
  - 50 ms
  - 1 13

- **B4b**
  - Device type bank 2
  - 4
  - EPROM type 1
  - 1

- **B13b**
  - Device size bank 2
  - 10 6
  - 4 x 16k x 8
  - 1 5

- **B15**
  - Device pinning bank 2
  - 3 33
  - 1
B16  Enable VPP generator

2
  ○
  ○
  1

B17  Select VPP bank 2

3
  ○
  ○  21V
  ○
  1

B18  Select output enable on VPP bank 2

2
  ○
  ○
  1

B19  Select chip erase bank 2

3
  ○
  ○
  ○
  ○
  1

B11  Upper address bank 2

2   8
  ○   ○   ○   ○   $8
  ○   ○   ○   ○
  1   7

B12  Lower address bank 2

2   8
  ○   ○   ○   ○   $0
  ○   ○   ○   ○
  1   7
Program call for subsequent jobs:

a) EPROM empty check

? RR2 M,0,$8000,$800000,2,E

  option = empty check
  mode = word
  RR-2 base address
  byte count (2 EPROMs 16k x 8)
  memory address (don’t care)
  source = memory

b) program EPROMs

? RR2 F,MYFILE:PRG/4,$800000,2,P

  option = program
  mode = word
  RR-2 base address
  source file name
  source = file

c) load EPROMs into memory

? RR2 M,$10000,$8000,$800000,2,L

  option = load
  mode = word
  RR-2 base address
  byte count (2 EPROMs 16k x 8)
  memory address
  destination = memory
1.2.47 SELFTEST - Perform On-board Selftest

Format: SELFTEST

This command performs a test of the on-board functions of the IBC board. It may only be run if no other tasks are created. If there are any other tasks no selftest will be made and an error will be reported. The selftest tests the memory of the IBC board and all devices on the board.

The following tests are performed in this order:

1. **I/O test**

   This function tests the access to and the interrupts from the SCC. If the SCC cannot generate interrupts an error will be reported.

2. **Memory test on the memory of the current task.**

   The following procedures are performed:

   1) Byte Test
   2) Word Test
   3) Long Word Test

   All passes of the memory test perform pattern reading and writing as well as bit shift tests. If an error occurs while writing to or reading from memory it will be reported.

3. **Clock Test**

   If the CPU does not receive timer interrupts from the CIO an error will be displayed. This ensures that VMEPROM could initialize the CIO 68230 properly and the interrupts from the CIO are working.

   **CAUTION:** During this process, all memory is cleared.

   **Example:**

   ? SELFTEST

   VMEPROM Hardware Selftest

   I/O test . . . . . passed
   Memory test . . . . . passed
   Clock test . . . . . passed
   ? . . . . . passed
1.2.48 SM - SEND MESSAGE

Format: SM [<task #>,<message>]

The SEND MESSAGE command puts an ASCII text message in a message buffer. The destination is specified by <task#>. The message can be up to 63 characters in length.

If a message is sent to itself, i.e. the task which is sending the message, the complete message is interrupted as a command line and executed.

Note: No other commands can be appended to an 'SM' command with a period, since the <message> parameter takes everything up to the carriage return.

If no parameter is given, all pending messages are displayed.

See also: KM - KILL MESSAGE.

Example:

? SM 2,Hello
?_
1.2.49 ST - SET TASK TERMINAL TYPE

Format: ST
        ST <type>

The ST command sets the position cursor (PSC$) and clear screen (CSC$) variables in the task control block (TCB). This command makes it easy to use various types of terminals together with VMEPROM. Each task has its own characters for these two functions, which are initialized, when the task is started, to the parent task control set.

If a legal <type> is passed in the command line, then ST simply enters the corresponding sequences into the user status block.

Otherwise, the command prints the following table of options:

D = VT52
L = Lear Siegler ADM3a
V = VT100
T = TVI 950
U = User defined
Type = _

and prompts the user for an input. Enter the letter representing the type of terminal you are using.

The terminal type setup is only required for the VMEPROM screen editor. No other function uses the terminal dependant sequences.

The default setup of VMEPROM is the codes for a VT52 terminal.

In addition to the built in terminal types, the ST command allows to enter the values for position cursor, clear screen, clear to end of screen and clear to end of line interactively with the “C” option. So nearly every terminal can be used with VMEPROM.

? St U  to to enter a user defined terminal

Enter encoded position cursor value: $.

Now the position cursor code can be entered in hex. The hex value must be 16 bit.
The format of the leading characters for cursor positioning is as follows (note that each letter represents a bit):

B111 1111 0222 2222

B = 0 then $00 bias
  1 then $20 bias
O = 0 then row before column, 1 then column before row
1 = 7 bits for first ASCII lead in character
2 = 7 bits for second ASCII lead in character
A value of 0 will result in the code for a VT100 terminal.

Enter encoded clear screen value: $_$

The cursor home and clear screen can also be entered as a encoded 16 bit value. The format is (note that each letter represents a bit):

E111 1111 E222 2222

E = if 1 then precede with [ESC]
  1 = 7 bits for first ASCII character
  2 = 7 bits for second ASCII character
If all 16 bits are 0 then a VT100 is selected

Enter encoded clear to end of screen value: $.

This is the code to clear the access from the current cursor position to end of screen. The format is:

0111 1111 0222 2222

1 = 7 bit for first ASCII character
2 = 7 bit for second ASCII character

Enter encoded clear to end of line value: $_$

This is the code to clear from the cursor position to the end of the line. The format is:

0111 1111 0222 2222

1 = 7 bit for first ASCII character
2 = 7 bit for second ASCII character
Example:

? ST
  D = VT52
  L = Lear Siegler ADM3a
  V = VT100
  T = TVI 950
  U = User defined
Type = L

? ST D
?
?
1.2.50 T - Trace Program Execution

Format:  
T  
T <address>[,<begin>,<end>]  
T <R|S|?>  
TT  
TT <address>[,<begin>,<end>]  
TT <R|S|?>

The first format starts a user program in trace mode. The start address is the current value of the program counter (PC) as displayed by the DR command.

The second format is used to start a user program in trace mode at the specified address. Additionally two parameters (<begin> and <end>) are able to be given. These parameters specify an address range. Inside this range the program does not stop tracing.

The third format is used to display/set the trace mode. The parameter "S" toggles between enabling and disabling trace over subroutine. No stop inside a subroutine (i.e. started with BSR) will be done if trace over subroutine is enabled. The parameter "R" toggles between displaying the registers after each step and displaying only if trace count matches or the condition for trace over range is true. Displaying registers goes along with displaying the disassembled code of the next instruction which will be executed. The parameter "?" induces the displayal of the current settings.

If the program stops the user is prompted to continue the trace or to return to VMEPROM. Tracing can be continued by entering a space (" ") or a carriage return (<cr>).

See also: TC - Set Trace Count  
TJ - Trace on change of flow

Example:

? DI 8000 7  
8000    SUBA.L A5,A5  
8002    ADDQ.L #1,A5  
8004    BSR.B $800A  
8006    ADDQ.L #3,A5  
8008    XEXT  
800A    ADDQ.L #2,A5  
800C    RTS  

? DR T  
PC = 00008000 SP = 003B67FC A6 = 00007000 A5 = 00001000  

? T ?  
Display registers after each step  
Trace over subroutine is disabled
(Example cont’d)

? T 8000
Trace
PC = 00008002  SP = 003B67FC  A6 = 00007000  A5 = 00000000
8002 : ADDQ.L #1,A5<cr>
Trace
PC = 00008004  SP = 003B67FC  A6 = 00007000  A5 = 00000001
8004 : BSR.B $800A<cr>
Trace
PC = 0000800A  SP = 003B67F8  A6 = 00007000  A5 = 00000001
800A : ADDQ.L #2,A5<cr>
Trace
PC = 0000800C  SP = 003B67F8  A6 = 00007000  A5 = 00000003
800C : RTS<cr>
Trace
PC = 00008006  SP = 003B67FC  A6 = 00007000  A5 = 00000003
8006 : ADDQ.L #3,A5<cr>
Trace
PC = 00008008  SP = 003B67FC  A6 = 00007000  A5 = 00000006
8008 : XEXT<cr>
?

? T 8000 800A 9000
Trace
PC = 00008002  SP = 003B67FC  A6 = 00007000  A5 = 00000000
8002 : ADDQ.L #1,A5<cr>
Trace
PC = 00008004  SP = 003B67FC  A6 = 00007000  A5 = 00000001
8004 : BSR.B $800A<cr>
Trace
PC = 0000800A  SP = 003B67F8  A6 = 00007000  A5 = 00000001
800A : ADDQ.L #2,A5
NO STOP!
Trace
PC = 0000800C  SP = 003B67F8  A6 = 00007000  A5 = 00000003
800C : RTS
NO STOP!
Trace
PC = 00008006  SP = 003B67FC  A6 = 00007000  A5 = 00000003
8006 : ADDQ.L #3,A5<cr>
Trace
PC = 00008008  SP = 003B67FC  A6 = 00007000  A5 = 00000006
8008 : XEXT<cr>
?

? T R
Display registers only if stop condition reached

? T 8000 800A 9000
Trace
PC = 00008002  SP = 003B67FC  A6 = 00007000  A5 = 00000000
8002 : ADDQ.L #1,A5<cr>
Trace
PC = 00008004  SP = 003B67FC  A6 = 00007000  A5 = 00000001
8004 : BSR.B $800A<cr>
Trace
PC = 00008006  SP = 003B67FC  A6 = 00007000  A5 = 00000003
8006 : ADDQ.L #3,A5<cr>
Trace
PC = 00008008  SP = 003B67FC  A6 = 00007000  A5 = 00000006
8008 : XEXT<cr>

3-68
(Example cont’d)

? T S
Trace over subroutine is enabled

? T 8000
Trace
PC = 00008002  SP = 003B67FC  A6 = 00007000  A5 = 00000000
8002 : ADDQ.L #1,A5<cr>
Trace
PC = 00008004  SP = 003B67FC  A6 = 00007000  A5 = 00000001
8004 : BSR.B $800A<cr>
Trace
PC = 00008006  SP = 003B67FC  A6 = 00007000  A5 = 00000003
8006 : ADDQ.L #3,A5<cr>
Trace
PC = 00008008  SP = 003B67FC  A6 = 00007000  A5 = 00000006
8008 : XEXT<cr>

? _
1.2.51 TC - Set Trace Count

Format: **TC <count>**

The Set Trace Count command sets the number of instructions to be traced continuously. The default after reset is 1.

See also: *T - Trace program execution* and *TJ - Trace on change of flow*

*Example:*

```
? TC
Trace count = 0

? TC 100
? TC
Trace count = 100

?
```
1.2.52 TIME - Enable/Disable Program Run Time Display

Format: TIME
        TIME ON
        TIME OFF

VMEPROM has the ability to measure the run time of user programs or command execution of the built-in commands. This feature can be turned on and off with the TIME command. If only TIME is entered, the current status is displayed (i.e. On or OFF). VMEPROM displays the time in minutes, seconds, and tens and hundreds of seconds. If time measurement is enabled, a time stamp is taken whenever the command interpreter gets a complete input line. The timing stops when the function is executed and control is transferred back to the command interpreter.

Example:

? TIME
Time is off

? TIME ON
? BENCH 1 8000
Bench 1: Decrement long word in memory, 10,000,000 times
Benchmark time = 0:07.23
Programm execution time is 0:07.27

? TIME OFF
?
1.2.53 TJ - Trace on Change of Flow

Format: TJ
        TJ <address>

This command is only supported on 68020 versions. It traces a user program (like the Trace command), but only on instructions where a change of program flow occurs. Such instructions are for example: BRA, BSR, JMP, JSR, RTS etc.

See the Trace command for a complete description of program tracing.

See also T - Trace program execution

Note: This command is only available for 32 bit processors.
1.2.54 TM - TRANSPARENT MODE

Format:  
\[ \text{TM } <\text{port #}> \]
\[ \text{TM } <\text{port #}>,<\text{break}> \]

The TRANSPARENT MODE command directs your current input to \(<\text{port #}>\). Input received from \(<\text{port #}>\) is directed to your output. This command effectively allows you to access other systems as if you were a terminal.

This process continues until an [ESC] character is entered. This can be changed to another character by adding the \(<\text{break}>\) parameter.

Caution:  
Typing \(^C\) twice will abort every command currently in the state of execution. This is independent of the brake character.
1.2.55 TP - TASK PRIORITY

Format:  
TP  
TP <task #>  
TP <task #>,<[time * 256 +] priority>

The TASK PRIORITY command allows you to change task priority of different tasks. The task number is specified by <task #> and defaults to the current task if omitted. If no priority is given the tasks current priority is displayed. Otherwise the tasks priority is changed to the given value.

The range of <priority> is from 1 to 255 where 255 is the highest priority. The highest priority, ready task always executes. Tasks on the same priority level are scheduled in a round robin fashion. The time a task is in running state is also given with the <priority> parameter. If no time is specified the time slice will not be changed. Otherwise it is calculated to "time*256+priority".

Example:

? It
   task pri tm ev1/ev2 size pc tcb eom ports name
*0/0 1 1 238 FFE1D46A 00007000 00042800 1/0/0/0 lt
1/-1 1 1 1762 FFE1010C 00047000 001FF800 0/0/0/0 MEtask
2/-1 1 1 6 0004FA1C 0004E9EE 000503AE 0/0/0/0 DMAtask

? tp
Current tasks priority = 1, time slice = 1

? tp 1
Task #1 priority = 1, time slice = 1

? tp 1,256*10+1

? It
   task pri tm ev1/ev2 size pc tcb eom ports name
*0/0 1 1 238 FFE1D46A 00007000 00042800 1/0/0/0 lt
1/-1 1 10 1762 FFE1010C 00047000 001FF800 0/0/0/0 MEtask
2/-1 1 1 6 0004FA1C 0004E9EE 000503AE 0/0/0/0 DMAtask

? tp 1
Task #1 priority = 1, time slice = 10

? tp 1,256*1+1

? It
   task pri tm ev1/ev2 size pc tcb eom ports name
*0/0 1 1 238 FFE1D46A 00007000 00042800 1/0/0/0 lt
1/-1 1 1 1762 FFE1010C 00047000 001FF800 0/0/0/0 MEtask
2/-1 1 1 6 0004FA1C 0004E9EE 000503AE 0/0/0/0 DMAtask
1.2.56 UN - CONSOLE UNIT

Format:  

UN {
  \{-128\}
}<unit number>

The CONSOLE UNIT command displays/sets the console output unit number. Unit 1 is the system terminal. Unit 2 and 3 are auxiliary output ports. The unit 4 is used by VMEPROM for output redirection and shall not be used.

The first format is used to display the current output unit number.

The second format selects where the output is to be directed. If the parameter is negative no character echo to the input port will be done. Otherwise character echo to the input port is enabled. If the parameter is lower than -128 only the system prompt will be displayed at the input port. No character echo of the input port is done. The correct parameter for this command is calculated to "-128-unit number". This command is very helpful to recognize if a command line can be entered.

Example:

\(?\) UN
Unit mask = 1

\(?\) UN 3

\(?\) UN
Unit mask = 3

\(?\) UN -1
\{LT\} No echo

\{UN -129\} No echo

\(?\) [LT]

<table>
<thead>
<tr>
<th>task</th>
<th>pri</th>
<th>tm</th>
<th>ev1/ev2</th>
<th>size</th>
<th>pc</th>
<th>tcb</th>
<th>eom</th>
<th>ports</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>238</td>
<td>FFE1D46A</td>
<td>00007000</td>
<td>00042800</td>
<td>1/1/0/0/0</td>
<td>lt</td>
</tr>
<tr>
<td>1/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1762</td>
<td>FFE1010C</td>
<td>00047000</td>
<td>001FF800</td>
<td>0/0/0/0/0</td>
<td>MEtask</td>
</tr>
<tr>
<td>2/-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0004FA1C</td>
<td>0004E9EE</td>
<td>000509AE</td>
<td>0/0/0/0/0</td>
<td>DMAtask</td>
</tr>
</tbody>
</table>

\{UN 1\} No echo

\(?\) _
1.2.57  ZM - ZERO MEMORY

Format:  ZM

The ZERO MEMORY command clears the entire user work space to zeros. All flags and pointers are reset.

The memory is cleared from the end of the TCB up to the current user stack pointer. The values on the stack are not destroyed.

Example:

? ZM
?
?


VMEPROM SYSTEM CALLS
# TABLE OF CONTENTS

1. VMEPROM SYSTEM CALLS .................................................. 4-1

1.1 General Information .................................................. 4-1

1.2 Assembly Language Calls ........................................... 4-2

1.3 Description of Kernel Primitives .................................. 4-4

1.3.1 X881 - SAVE 68881 ENABLE ..................................... 4-5
1.3.2 XAPF - APPEND FILE ........................................... 4-6
1.3.3 XBCH - BAUD CONSOLE PORT .................................... 4-7
1.3.4 XCBC - CHECK FOR BREAK CHARACTER .......................... 4-8
1.3.5 XCBD - CONVERT BINARY TO DECIMAL ............................. 4-9
1.3.6 XCBH - CONVERT BINARY TO HEX ................................ 4-10
1.3.7 XCBM - CONVERT TO DECIMAL W/MESSAGE ....................... 4-11
1.3.8 XCBP - CHECK FOR BREAK OR PAUSE ........................... 4-12
1.3.9 XCBX - CONVERT TO DECIMAL IN BUFFER ....................... 4-13
1.3.10 XCBX - CONVERT TO DECIMAL IN BUFFER ..................... 4-14
1.3.11 XCF - CLOSE FILE W/ATTRIBUTE ................................ 4-15
1.3.12 XCHX - CONVERT BINARY TO HEX IN BUFFER .................. 4-16
1.3.13 XCLF - CLOSE FILE ........................................... 4-17
1.3.14 XCLS - CLEAR SCREEN ......................................... 4-18
1.3.15 XCPY - COPY FILE ........................................... 4-19
1.3.16 XCTB - CREATE TASK BLOCK ................................... 4-20
1.3.17 XDEV - DELAY SET/RESET EVENT ............................... 4-22
1.3.18 XDFL - DEFINE FILE .......................................... 4-23
1.3.19 XDLF - DELETE FILE ........................................... 4-24
1.3.20 XDMP - DUMP MEMORY FROM STACK ............................. 4-25
1.3.21 XDPE - DELAY PHYSICAL EVENT ................................ 4-26
1.3.22 XD TC - DEFINE TRAP VECTORS ............................... 4-27
1.3.23 XERR - RETURN ERROR D0 TO VMEPROM ....................... 4-29
1.3.24 XEXC - EXECUTE PDOS CALL D7.W ............................. 4-30
1.3.25 XEXT - EXIT TO VMEPROM .................................... 4-31
1.3.26 XFAC - FILE ALTERTED CHECK ................................ 4-32
1.3.27 XFBF - FLUSH BUFFERS ....................................... 4-33
1.3.28 XFFN - FIX FILE NAME ....................................... 4-34
1.3.29 XFTD - FIX TIME & DATE ..................................... 4-35
1.3.30 XFBF - FREE USER MEMORY .................................... 4-36
1.3.31 XGCB - CONDITIONAL GET CHARACTER ......................... 4-37
1.3.32 XGCC - GET CHARACTER CONDITIONAL ......................... 4-38
1.3.33 XGCP - GET PORT CHARACTER ................................ 4-39
1.3.34 XGCR - GET CHARACTER ....................................... 4-40
1.3.35 XGLB - GET LINE IN BUFFER .................................. 4-41
1.3.36 XGLM - GET LINE IN MONITOR BUFFER ......................... 4-42
1.3.37 XGLU - GET LINE IN USER BUFFER ............................. 4-43
1.3.38 XGML - GET MEMORY LIMITS .................................. 4-44
1.3.39 XGMP - GET MESSAGE POINTER ................................ 4-45
1.3.40 XGNP - GET NEXT PARAMETER ................................. 4-46
# TABLE OF CONTENTS

1.3.41 XGTM - GET TASK MESSAGE ........................................ 4-47
1.3.42 XGUM - GET USER MEMORY ........................................ 4-48
1.3.43 XISE - INITIALIZE SECTOR .......................................... 4-49
1.3.44 XKTB - KILL TASK ................................................. 4-50
1.3.45 XKTM - KILL TASK MESSAGE ........................................ 4-51
1.3.46 XLDF - LOAD FILE ................................................ 4-52
1.3.47 XLER - LOAD ERROR REGISTER ..................................... 4-53
1.3.48 XLFN - LOOK FOR NAME IN FILE SLOTS ............................... 4-54
1.3.49 XLFK - LOCK FILE ................................................. 4-55
1.3.50 XLKT - LOCK TASK ................................................ 4-56
1.3.51 XLSR - LOAD STATUS REGISTER .................................... 4-57
1.3.52 XNOP - OPEN SHARED RANDOM FILE ................................. 4-58
1.3.53 XPAD - PACK ASCII DATE .......................................... 4-59
1.3.54 XPBC - PUT BUFFER TO CONSOLE ................................... 4-60
1.3.55 XPCC - PUT CHARACTER(S) TO CONSOLE ............................. 4-61
1.3.56 XPCL - PUT CRLF TO CONSOLE ..................................... 4-62
1.3.57 XPCP - PLACE CHARACTER IN PORT BUFFER .......................... 4-63
1.3.58 XPCR - PUT CHARACTER RAW ....................................... 4-64
1.3.59 XPDC - PUT DATA TO CONSOLE ..................................... 4-65
1.3.60 XPEL - PUT ENCODED LINE TO CONSOLE ............................. 4-66
1.3.61 XPEM - PUT ENCODED MESSAGE TO CONSOLE ........................ 4-67
1.3.62 XPLC - PUT LINE TO CONSOLE ...................................... 4-68
1.3.63 XPMC - PUT MESSAGE TO CONSOLE ................................ 4-69
1.3.64 XPSC - POSITION CURSOR ......................................... 4-70
1.3.65 XPSCF - POSITION FILE .......................................... 4-71
1.3.66 XPSP - PUT SPACE TO CONSOLE .................................... 4-72
1.3.67 XRBF - READ BYTES FROM FILE ..................................... 4-73
1.3.68 XRCN - RESET CONSOLE INPUTS .................................... 4-74
1.3.69 XRCP - READ PORT CURSOR POSITION ............................... 4-75
1.3.70 XRDE - READ NEXT DIRECTORY ENTRY ............................... 4-76
1.3.71 XRDM - DUMP REGISTERS ......................................... 4-77
1.3.72 XRDN - READ DIRECTORY ENTRY BY NAME ............................ 4-78
1.3.73 XRDT - READ DATE ............................................... 4-79
1.3.74 XRFA - READ FILE ATTRIBUTES ..................................... 4-80
1.3.75 XRFP - READ FILE POSITION ....................................... 4-81
1.3.76 XRLF - READ LINE FROM FILE ..................................... 4-82
1.3.77 XRNF - RENAME FILE .............................................. 4-83
1.3.78 XROO - OPEN RANDOM READ ONLY FILE .............................. 4-84
1.3.79 XROP - OPEN RANDOM ............................................ 4-85
1.3.80 XRPS - READ PORT STATUS ........................................ 4-86
1.3.81 XRS - READ SECTOR ................................................ 4-87
1.3.82 XRSR - READ STATUS REGISTER .................................... 4-88
1.3.83 XRSR - READ STATUS REGISTER .................................... 4-88
1.3.84 XRSZ - READ SECTOR ZERO .......................................... 4-89
1.3.85 XRTE - RETURN FROM INTERRUPT ................................... 4-90
1.3.86 XRTM - READ TIME ............................................... 4-91
### TABLE OF CONTENTS

| 1.3.87  | XRTP - READ TIME PARAMETERS                  | 4-93 |
| 1.3.88  | XRTS - READ TASK STATUS                     | 4-94 |
| 1.3.89  | XRWF - REWIND FILE                          | 4-95 |
| 1.3.90  | XSEF - SET EVENT FLAG W/SWAP                | 4-96 |
| 1.3.91  | XSEV - SET EVENT FLAG                       | 4-97 |
| 1.3.92  | XSMP - SEND MESSAGE POINTER                 | 4-98 |
| 1.3.93  | XSOE - SUSPEND ON PHYSICAL EVENT            | 4-99 |
| 1.3.94  | XSOP - OPEN SEQUENTIAL FILE                 | 4-100|
| 1.3.95  | XSPF - SET PORT FLAG                        | 4-101|
| 1.3.96  | XSTM - SEND TASK MESSAGE                    | 4-102|
| 1.3.97  | XSTP - SET/READ TASK PRIORITY               | 4-103|
| 1.3.98  | XSUI - SUSPEND UNTIL INTERRUPT              | 4-104|
| 1.3.99  | XSUP - ENTER SUPERVISOR MODE                | 4-105|
| 1.3.100 | XSWP - SWAP TO NEXT TASK                    | 4-106|
| 1.3.101 | XSZF - GET DISK SIZE                        | 4-107|
| 1.3.102 | XTAB - TAB TO COLUMN                        | 4-108|
| 1.3.103 | XTEF - TEST EVENT FLAG                      | 4-109|
| 1.3.104 | XTLP - TRANSLATE LOGICAL TO PHYSICAL EVENT  | 4-110|
| 1.3.105 | XUAD - UNPACK ASCII DATE                    | 4-112|
| 1.3.106 | XUDT - UNPACK DATE                          | 4-113|
| 1.3.107 | XULF - UNLOCK FILE                          | 4-114|
| 1.3.108 | XULT - UNLOCK TASK                          | 4-115|
| 1.3.109 | XUSP - RETURN TO USER MODE                  | 4-116|
| 1.3.110 | XUTM - UNPACK TIME                          | 4-117|
| 1.3.111 | XVEC - SET/READ EXCEPTION VECTOR            | 4-118|
| 1.3.112 | XWBF - WRITE BYTES TO FILE                  | 4-119|
| 1.3.113 | XWDT - WRITE DATE                           | 4-120|
| 1.3.114 | XWFA - WRITE FILE ATTRIBUTES                | 4-121|
| 1.3.115 | XWFP - WRITE FILE PARAMETERS                | 4-122|
| 1.3.116 | XWLF - WRITE LINE TO FILE                   | 4-123|
| 1.3.117 | XWSE - WRITE SECTOR                         | 4-124|
| 1.3.118 | XWTM - WRITE TIME                           | 4-125|
| 1.3.119 | XZFL - ZERO FILE                            | 4-126|
1. VMEPROM SYSTEM CALLS

1.1 General Information

PDOS assembly primitives are assembly language system calls to PDOS. They consist of one word A-line instructions (words with the first four bits equal to hexadecimal 'A'). PDOS calls return results in the 68000 status register as well as regular user registers.

PDOS calls are divided into three categories:

1) System
2) Console I/O
3) File support primitives. Please note that these primitives are included, but all of them will return with an error. This is because of the loss of any file system in the VMEPROM of IBC.

The following primitives are available in a PDOS operating system environment, but not in VMEPROM:

**PDOS debugger:**
- XBUG

**PDOS monitor command:**
- XCHF
- XLST
- XBFL
- XAIM
- XGTP
- XEXZ

These primitives give reference to the PDOS Monitor/Debugger and are not included in VMEPROM. The monitor calls XGNP and XPCB of PDOS are emulated by VMEPROM and perform their expected functions.
1.2 Assembly Language Calls

PDOS assembly primitives are one word A-line instructions which use the exception vector at memory location $00000028. Most primitives use 68000 registers to pass parameters to and results from resident PDOS routines. Observe the following example for Trapping an error after a PDOS call:

```
CALLX LEA.L FILEN(PC),A1 ;GET FILE NAME
XSOP ;OPEN FILE, ERROR?
BNE.S ERROR ;Y
MOVE.W D1,SLTN(A4) ;N, SAVE SLOT #
```

PDOS primitives return error conditions in the processor status register. This provides error processing allowing a program to do long or short branches on different error conditions.

PDOS command primitives can be grouped into six levels according to their function and calling hierarchy. These levels are System Calls, System Support Calls, Console I/O Calls, File Support Calls, File Management Calls, and Disk Access Calls.

Level 1 PDOS primitives consist of system calls that deal with functions such as swapping, message passing, events, TRAP vector initialization, etc.

The PDOS system calls are as follows:

- **XGML** - Get memory limits
- **XGUM** - Get user memory
- **XFUM** - Free user memory
- **XRTS** - Read task status
- **XSTP** - Set/read task priority
- **XLKT** - Lock task
- **XULT** - Unlock task
- **XSWP** - Swap to next task
- **XCTB** - Create task block
- **XKTB** - Kill task
- **XSTM** - Send task message
- **XGTM** - Get task message
- **XKTM** - Kill task message
- **XGMP** - Get message pointer
- **XSMP** - Send message pointer
- **XSEV** - Set event flag
- **XSEF** - Set event flag w/swap
- **XTEF** - Test event flag
- **XGTF** - Get user memory
- **XDEV** - Delay set/reset event
- **XLSR** - Load status register
Level 2 consists of system support calls. Data conversion and data/time processing are their main functions. They are as follows:

- **XCBD** - Convert binary to decimal
- **XCBH** - Convert binary to hex
- **XCBM** - Convert to decimal w/message
- **XCDB** - Convert decimal to binary
- **XCBX** - Convert to decimal in buffer
- **XCHX** - Convert binary to hex in buffer
- **XRDT** - Read date
- **XRTM** - Read time
- **XGNP** - Get next parameter

- **XRTP** - Read time parameters
- **XFTD** - Fix time & date
- **XPAD** - Pack ASCII date
- **XUAD** - Unpack ASCII Date
- **XUAY** - Convert to decimal w/message
- **XUDB** - Convert binary to hex
- **XUDT** - Unpack date
- **XPDG** - Pack ASCII date
- **XNUA** - Unpack ASCII time
- **XVMM** - Write date
- **XWHT** - Write time

Level 3 primitives deal with console I/O. Included are commands for setting the baud rate and other characteristics of an I/O port, reading and writing characters or lines, clearing the screen, positioning the cursor, and monitoring port status.

- **XGCB** - Conditional get character
- **XGCR** - Get character
- **XGCP** - Get port character
- **XGCL** - Get line in buffer
- **XGMB** - Get line in monitor buffer
- **XGRC** - Get port character
- **XGMC** - Get line in monitor buffer
- **XGUL** - Get line in user buffer
- **XPBC** - Put buffer to console
- **XPCC** - Put character(s) to console
- **XPCL** - Put CRLF
- **XPDC** - Put data to console
- **XPEL** - Put encoded line to console
- **XPEM** - Put encoded message to console
- **XCLS** - Clear screen
- **XSTEM** - Position cursor
- **XSPC** - Position cursor
- **XTAB** - Tab to column
- **XRCP** - Read port cursor position
- **XBCP** - Baud console port
- **XSPF** - Set port flag
- **XRPS** - Read port status
- **XAPF** - Append file
- **XCPY** - Copy file
- **XLDF** - Load file
- **XRCN** - Reset console inputs
- **XRST** - Reset disk
- **XSZF** - Get disk size
Level 5 primitives are the file management calls of PDOS. They use the file lock (event 120) to prevent conflicts between multiple tasks. Functions such as defining, deleting, reading, writing, positioning, and locking are supported by the file manager.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDFL</td>
<td>Define file</td>
</tr>
<tr>
<td>XRNF</td>
<td>Rename file</td>
</tr>
<tr>
<td>XRFA</td>
<td>Read file attributes</td>
</tr>
<tr>
<td>XWFA</td>
<td>Write file attributes</td>
</tr>
<tr>
<td>XWFP</td>
<td>Write file parameters</td>
</tr>
<tr>
<td>XDLF</td>
<td>Delete file</td>
</tr>
<tr>
<td>XZFL</td>
<td>Zero file</td>
</tr>
<tr>
<td>XSOP</td>
<td>Open sequential</td>
</tr>
<tr>
<td>XROO</td>
<td>Open random read only</td>
</tr>
<tr>
<td>XROP</td>
<td>Open random</td>
</tr>
<tr>
<td>XNOP</td>
<td>Open non-exclusive random</td>
</tr>
<tr>
<td>XLKF</td>
<td>Lock file</td>
</tr>
<tr>
<td>XULF</td>
<td>Unlock file</td>
</tr>
<tr>
<td>XRFP</td>
<td>Read file position</td>
</tr>
<tr>
<td>XRWF</td>
<td>Rewind file</td>
</tr>
<tr>
<td>XPSF</td>
<td>Position file</td>
</tr>
<tr>
<td>XRWF</td>
<td>Read bytes from file</td>
</tr>
<tr>
<td>XRBF</td>
<td>Read bytes from file</td>
</tr>
<tr>
<td>XRLF</td>
<td>Read line from file</td>
</tr>
<tr>
<td>XWBF</td>
<td>Write bytes to file</td>
</tr>
<tr>
<td>XWLF</td>
<td>Write line to file</td>
</tr>
<tr>
<td>XFBF</td>
<td>Flush buffers</td>
</tr>
<tr>
<td>XFAC</td>
<td>File altered check</td>
</tr>
<tr>
<td>XCFA</td>
<td>Close file w/attribute</td>
</tr>
<tr>
<td>XCLF</td>
<td>Close file</td>
</tr>
</tbody>
</table>

The final level of primitives is for disk access via the read/write logical sector routines in the PDOS BIOS. A disk lock (event 121) is used to make these calls autonomous and prevent multiple commands from being sent to the disk controller.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XISE</td>
<td>Initialize sector</td>
</tr>
<tr>
<td>XWSE</td>
<td>Write sector</td>
</tr>
<tr>
<td>XRSE</td>
<td>Read sector</td>
</tr>
<tr>
<td>XRSZ</td>
<td>Read sector zero</td>
</tr>
</tbody>
</table>

### 1.3 Description of Kernel Primitives

The following chapters give a detailed description of all Kernel calls available in VMEPROM.
1.3.1 X881 - SAVE 68881 ENABLE

Mnemonic: X881  
Value: $A006  
Module: MPDOSK1  
Format: X881

Description: The SAVE 68881 ENABLE sets the BIOS save flag (SVF$(A6)) thus signaling the PDOS BIOS to save and restore 68881 registers and status during context switches. The save flag is again cleared by exiting to VMEPROM.

See also: None

Possible Errors: None
1.3.2 XAPF - APPEND FILE

Mnemonic: XAPF
Value: $A0AA
Module: MPDOSF
Format: XAPF

<status error return>

Registers: In (A1) = Source file name
(A2) = Destination file name

Note: A [CTRL-C] will terminate this primitive and return error -1 in data register D0.

Description: The APPEND FILE primitive is used to append two files together.

The source and destination file names are pointed to by address registers A1 and A2, respectively. The source file is appended to the end of the destination file. The source file is not altered.

See also: None

Possible Errors:

-1 = Break
50 = Invalid file name
53 = File not defined
60 = File space full
61 = File already open
68 = Not PDOS disk
69 = Not enough file slots
Disk errors
1.3.3 XBCP - BAUD CONSOLE PORT

Mnemonic: XBCP
Value: $A070
Module: MPDOSK2
Format: XBCP
<status error return>

Registers: In D2.W = f0PI 8DBS / <port #>
D3.W = Baud rate
D1.W = Port type
D5.L = Port base

Description: The BAUD CONSOLE PORT primitive initializes any one of the PDOS I/O ports and binds a physical UART to a character buffer. The primitive sets handshaking protocol, receiver and transmitter baud rates, and enables receiver interrupts.

Data register D2 selects the port number and sets (or clears) the corresponding flag bits. If D2.W is negative, then the absolute value is subsequently used and the port number is stored in U2P$(A6).

The right byte of data register D2 (bits 0-7) selects the console port.

The left byte of D2.W (bits 8-15) selects various flag options including ^S-^Q and/or DTR handshaking, receiver parity and interrupt enable, and 8-bit character I/O.

The receiver and transmitter baud rates are initialized to the same value according to register D3. Register D3 ranges from 0 to 7 or the corresponding baud rates of 19200, 9600, 4800, 2400, 1200, 600, 300, or 110.

If data register D4 is non-zero, then it selects the port type and register D5 selects the port base address. These parameters are system-defined and correspond to the UART module. If register D4 is zero, there is no change.

See also: XRPS - READ PORT STATUS
XSPF - SET PORT FLAG

Possible Errors: 66 = Invalid port or baud rate
1.3.4 XCBC - CHECK FOR BREAK CHARACTER

Mnemonic: XCBC
Value: $A072
Module: MPDOSK2
Format: XCBC
<status return>

Registers: Out SR =
           EQ....No break
           LO....[CTRL-C], Clear flag & buffer
           LT....[ESC], Clear flag
           MI....[CTRL-C] or [ESC]

Note: If the ignore control character bit ($02) of the port flag is set, then XCBC always returns .EQ. status.

Description: The CHECK FOR BREAK CHARACTER primitive checks the current user input port break flag (BRKF.(A5)) to see if a break character has been entered. The PDOS break characters are [CTRL-C] and the [ESC] key. A [CTRL-C] sets the port break flag to one, while an [ESC] character sets the flag to a minus one. The XCBC primitive samples and clears this flag. The condition of the break flag is returned in the status register. An 'LO' condition indicates a [CTRL-C] has been entered. The break flag and the input buffer are cleared. All subsequent characters entered after the [CTRL-C] and before the XCBC call are dropped.

All open procedure files are closed and any system frames are restored. Also, the last error number flag (LEN$) is set to -1 and a '^C' is output to the port. An 'LT' condition indicates an [ESC] character has been entered. Only the break flag is cleared and not the input buffer. Thus, the [ESC] character remains in the buffer.

The [CTRL-C] character is interpreted as a hard break and is used to terminate command operations. The [ESC] character is a soft break and remains in the input buffer, even though the break flag is cleared by the XCBC primitive. (This allows an editor to use the [ESC] key for special functions or command termination.)

See also: None

Possible Errors: None
1.3.5 XCBD - CONVERT BINARY TO DECIMAL

Mnemonic: XCBD
Value: $A050
Module: MPDOSK3
Format: XCBD

Registers: In D1.L = Number
Out (A1) = String

Description: CONVERT BINARY TO DECIMAL primitive converts a 32-bit, 2’s complement number to a character string. The number to be converted is passed to XCBD in data register D1. Address register A1 is returned with a pointer to the converted character string located in the monitor work buffer (MWB$).

Leading zeros are suppressed and a negative sign is the first character for negative numbers. The string is delimited by a null. The string has a maximum length of 11 characters and ranges from -2147483648 to 2147483647.

See also: XCBX - CONVERT TO DECIMAL IN BUFFER.

Possible Errors: None
1.3.6 XCBH - CONVERT BINARY TO HEX

Mnemonic: XCBH
Value: $A052
Module: MPDOSK3
Format: XCBH

Registers: In D1.L = Number
            Out (A1) = String

Description: CONVERT BINARY TO HEX primitive converts a 32-bit number to its hexadecimal (base 16) representation. The number is passed in data register D1 and a pointer to the ASCII string is returned in address register A1. The converted string is found in the monitor work buffer (MWB$) of the task control block and consists of eight hexadecimal characters followed by a null.

See also: XCHX - CONVERT BINARY TO HEX IN BUFFER.

Possible Errors: None
1.3.7 XCBM - CONVERT TO DECIMAL W/MESSAGE

Mnemonic: XCBM
Value: $A054
Module: MPDOSK3
Format: XCBM <message>

Registers: In D1.L = Number
Out (A1) = String

Description: CONVERT TO DECIMAL WITH MESSAGE primitive converts a 32-bit, signed number to a character string. The output string is preceded by the string whose PC relative address is in the operand field of the call.

The string can be up to 20 characters in length and is terminated by a null character. The number to be converted is passed to XCBM in data register D1. Address register A1 is returned with a pointer to the converted character string which is located in the monitor work buffer (MWB$) of the task control block.

Leading zeros are suppressed and the result ranges from -2147483648 to 2147483647.

The message address is a signed 16-bit PC relative address.

See also: None

Possible Errors: None
1.3.8 XCBP - CHECK FOR BREAK OR PAUSE

Mnemonic: XCBP
Value: $A074
Module: MPDOSK2
Format: XCBP
<status return>

Registers: Out SR = EQ...No character
            LT...[ESC]
            LO...[CTRL-C]
            NE...Pause

Note: If a 'BLT' instruction does not immediately follow the XCBP call, then the primitive exits to PDOS when an [ESC] character is entered.

If the ignore control character bit ($02) of the port flag is set, then XCBP always returns .EQ. status.

Description: CHECK FOR BREAK OR PAUSE primitive looks for a character from your PRT$(A6) port. Any non-control character will cause XCBP to output a pause message and wait for another character.

The pause message consists of:

[CR]
'Strike any key...'
[CR]

[CR].

A [CTRL-C] will abort any assigned console file and return the status 'LO'. If a 'BLT' instruction follows the XCBP primitive and an [ESC] character is entered, then the call returns with status 'LT'. Otherwise, an [ESC] will abort your program to VMEPROM.

An 'EQ' status indicates that no character was entered. An 'NE' status indicates a pause has occurred.

See also: None

Possible Errors: None
1.3.9 XCBX - CONVERT TO DECIMAL IN BUFFER

Mnemonic: XCBX
Value: $A06A
Module: MPDOSK3
Format: XCBX

Registers: In D1.L = Number
(A1) = Buffer

Description: CONVERT TO DECIMAL IN BUFFER primitive converts a 32-bit, 2's complement number to a character string. The number to be converted is passed to XCBX in data register D1. Address register A1 points to the buffer where the converted string is stored.

Leading zeros are suppressed and a negative sign is the first character for negative numbers. The string is delimited by a null. The string has a maximum length of 11 characters and ranges from -2147483648 to 2147483647.

See also: XCBD - CONVERT BINARY TO DECIMAL.

Possible Errors: None
1.3.10 XCDB - CONVERT ASCII TO BINARY

Mnemonic: XCDB
Value: $A056
Module: MPDOSK3
Format: XCDB
<status return>

Registers: In (A1) = String
Out D0.B = Delimiter
D1.L = Number
(A1) = Updated string
SR = LT....No number
   EQ....# w/o null delimiter
   GT....#

Note: XCDB does not check for overflow.

Description: CONVERT ASCII TO BINARY primitive converts an ASCII string of characters to a 32-bit, 2’s complement number. The result is returned in data register D1 while the status register reflects the conversion results.

XCDB converts signed decimal, hexadecimal, or binary numbers.

Hexadecimal numbers are preceded by "$" and binary numbers by "%". A "-" indicates a negative number. There can be no embedded blanks.

An 'LT' status indicates that no conversion was possible. Data register D0 is returned with the first character and address register A1 points immediately after it.

A 'GT' status indicates that a conversion was made with a null delimiter encountered. The result is returned in data register D1. Address register A1 is returned with an updated pointer and register D0 is set to zero.

An 'EQ' status indicates that a conversion was made but the ASCII string was not terminated with a null character.

The result is returned in register D1 and the non-numeric, non-null character is returned in register D0.

Address register A2 has the address of the next character.

See also: None

Possible Errors: None
1.3.11 XCFA - CLOSE FILE W/ATTRIBUTE

Mnemonic: XCFA
Value: $A0D0
Module: MPDOSF
Format: XCFA
<status error return>

Registers: In D1.W = File ID
D2.B = New attribute

Description: CLOSE FILE WITH ATTRIBUTES primitive closes the open file specified by data register D1. At the same time, the file attributes are updated according to the byte contents of data register D2.

D2.B = $80 AC or Procedure file
= $40 BN or Binary file
= $20 OB or 68000 object file
= $10 SY or 68000 memory image
= $08 BX or BASIC binary token file
= $04 EX or BASIC ASCII file
= $02 TX or Text file
= $01 DR or System I/O driver
= $00 Clear file attributes

If the file was opened for sequential access and the file has been updated, then the END-OF-FILE marker is set at the current file pointer. If the file was opened for random or shared access, then the END-OF-FILE marker is updated only if the file has been extended (data was written after the current END-OF-FILE marker). The LAST UPDATE is updated to the current date and time only if the file has been altered. All files must be closed when opened! Otherwise, directory information and possibly even the file itself will be lost.

Note: If the file is not altered, then XCFA will not alter the file attributes.

See also: XRFA - READ FILE ATTRIBUTES
XWFA - WRITE FILE ATTRIBUTES
XWFP - WRITE FILE PARAMETERS

Possible Errors:

52 = File not open
59 = Invalid file slot
75 = File locked
Disk errors
1.3.12 XCHX - CONVERT BINARY TO HEX IN BUFFER

Mnemonic: XCHX
Value: $A068
Module: MPDOSK3
Format: XCHX

Registers:
In  D1.L = Number
(A1) = Output buffer

Description: CONVERT BINARY TO HEX IN BUFFER primitive converts a 32-bit number to its hexadecimal (base 16) representation. The number is passed in data register D1 and a pointer to a buffer in address register A1. The converted string consists of eight hexadecimal characters followed by a null.

See also: XCBH - CONVERT BINARY TO HEX.

Possible Errors: None
1.3.13 XCLF - CLOSE FILE

Mnemonic: XCLF
Value: $A0D2
Module: MPDOSF
Format: XCLF
<status error return>

Registers: In D1.W = File ID

Description: CLOSE FILE primitive closes the open file as specified by the file ID in data register D1. If the file was opened for sequential access and the file was updated, then the END-OF-FILE marker is set at the current file pointer.

If the file was opened for random or shared access, then the END-OF-FILE marker is updated only if the file was extended (ie. data was written after the current END-OF-FILE marker).

If the file has been altered, the current date and time is stored in the LAST UPDATE variable of the file directory. All open files must be closed at or before the completion of a task (or before disks are removed from the system)! Otherwise, directory information is lost and possibly even the file itself.

See also: None

Possible Errors:

52 = File not open
59 = Invalid slot #
75 = File locked
Disk errors
1.3.14 XCLS - CLEAR SCREEN

Mnemonic: XCLS
Value: $A076
Module: MPDOSK2
Format: XCLS

Registers: None

Note: The clear screen characters are located in the user TCB variable CSC$(A6).

Description: CLEAR SCREEN primitive clears the console screen, homes the cursor, and clears the column counter. This function is adapted to the type of console terminals used in the PDOS system.

The character sequence to clear the screen is located in the task control block variable CSC$(A6). These characters are transferred from the parent task to the spawned task during creation. The initial characters come from the BIOS module.

If CSC$ is nonzero, then the CLEAR SCREEN primitive outputs up to four characters: one or two characters; an [ESC] followed by a character; or an [ESC], character, [ESC], and a final character. The one-word format allows for two characters. The parity bits cause the [ESC] character to precede each character.

If CSC$ is zero, then PDOS makes a call into the BIOS for custom clear screens. The entry point is B_CLS beyond the BIOS table.

The ST command maintains the CSC$ field, although it can be altered under program control.

See also: XRCP - READ PORT CURSOR POSITION

Possible Errors: None
**1.3.15 XCPY - COPY FILE**

**Mnemonic:** XCPY  
**Value:** $A0AE  
**Module:** MPDOSF  
**Format:** XCPY  

<status error return>

**Registers:**  
(A1) = Source file name  
(A2) = Destination file name

**Note:** A [CTRL-C] terminates this primitive and returns the error -1 in register D0.

**Description:** COPY FILE primitive copies the source file into the destination file. The source file is pointed to by address register A1 and the destination file is pointed to by register A2. A [CTRL-C] halts the copy, prints '^C' to the console, and returns with error -1. The file attributes of the source file are automatically transferred to the destination file.

**See also:** None

**Possible Errors:**

-1 = Break file transfer  
50 = Invalid file name  
53 = File not defined  
60 = File space full  
61 = File already open  
68 = Not PDOS disk  
69 = No more file slots  
70 = Position error  
Disk errors
1.3.16 XCTB - CREATE TASK BLOCK

Mnemonic: XCTB  
Value: $A026  
Module: MPDOSK1  
Format: XCTB  
<status error return>

Registers:
In  D0.W = Task size (1 Kbyte increments)  
D2.W = I/O port  
(A0) = Optional low memory pointer  
(A1) = Optional high memory pointer  
(A2) = Command line pointer or entry address  
Out  D0.L = Spawned task number

Note: If D0.W is positive, A0 and A1 are undefined.

If D0.W equals zero, A0 and A1 are the new task’s memory bounds and A2 contains the task’s entry address.

If D0.W is negative, A0 and A1 are the new task’s memory bounds and A2 points to the task’s command line.

Description: CREATE TASK primitive places a new task entry in the PDOS task list. Memory for the new task is either from the parent task or the system memory bit map. Data register D0 controls the creation mode of the new task as well as the task size. If register D0.W is positive, the first available contiguous memory block equal to D0.W (in 1 Kbyte) is allocated to the new task. If the block is not big enough, the upper memory of the parent task is allocated to the new task. The parent task’s memory is then reduced by D0.W x 1 Kbytes. Address register A2 points to the new task command line. If A2 is zero, VMEPROM is invoked. If register D0.W is zero, registers A0 and A1 specify the new task’s memory limits. Register A2 specifies the task’s starting PC. The task control block begins at (A0) and is immediately followed by an XEXT primitive. The task user stack pointer is set at (A1). Thus, the new program should allow $1000 bytes at the low end and enough user stack space at the upper end.

If data register D0.W is negative, registers A0 and A1 specify the new task’s memory limits. Register A2 points to the new task command line. (If A2=0, VMEPROM is invoked). The command line is transferred to the spawned program by a system message buffer. The maximum command line length is 64 characters. When the task is scheduled for the first time, message buffers are searched for a command. Messages with a source task equal to $FF are considered commands and moved to the task’s monitor buffer.
The task CLI then processes the line. If no command message is found, then the VMEPROM is called directly.

Data register D1.W specifies the new task’s priority. The range is from 1 to 255. The larger the number, the higher the priority.

Data register D2.W specifies the I/O port to be used by the new task.

If register D2.W is positive, then the port is available for both input and output. If register D2.W is negative, then the port is used only for output. If register D2.W is zero, then no port is assigned. Only one task may be assigned to any one input port while many tasks may be assigned to an output port. Hence, a port is allocated for input only if it is available. An invalid port assignment does not result in an error.

A call is made to D$INT in the debugger module. This initializes all addresses, registers, breaks, and offsets.

Finally, the spawned task’s number is returned in register D0.L to the parent task. This can be used later to test task status or to kill the task.

See also: None

Possible Errors:

- 72 = Too many tasks
- 73 = Not enough memory
1.3.17 XDEV - DELAY SET/RESET EVENT

Mnemonic: XDEV
Value: $A032
Module: MPDOSK1
Format: XDEV
<status error return>

Registers: In
D0.L = Time
D1.B = Event (+=Set, -=Reset)

Note: If D0.L=0, then the D1.B event is cleared.

Description: DELAY SET/RESET EVENT primitive places a timed event in a system stack controlled by the system clock. Data register D0.L specifies the time interval in clock tics. When it counts to zero, then the event D1.B is set if positive, or reset if negative.

If the event already exists in the stack, it is replaced by the new entry. If the time specified in D0 equals zero, then any pending timed event equal to D1.B is deleted from the stack.

If D1.B is positive, event D1.B is first cleared. If D1.B is negative, event D1.B is set before exiting the primitive.

See also: XSEF - SET EVENT FLAG W/SWAP
XSEV - SET EVENT FLAG
XSUI - SUSPEND UNTIL INTERRUPT
XTEF - TEST EVENT FLAG

Possible Errors: 83 = Delay event stack full
1.3.18 XDFL - DEFINE FILE

Mnemonic: XDFL
Value: $A0D4
Module: MPDOSF
Format: XDFL
<status error return>

Registers: In D0.W = # of contiguous sectors
(A1) = File name

Description: DEFINE FILE primitive creates a new file entry in a PDOS disk directory, specified by address register A1. A PDOS file name consists of an alphabetic character followed by up to 7 additional characters. An optional 3 character extension can be added if preceded by a colon. Likewise, the directory level and disk number are optionally specified by a semicolon and slash respectively. The file name is terminated with a null.

Data register D0 contains the number of sectors to be initially allocated at file definition. If register D0 is nonzero, then a contiguous file is created with D0 sectors. Otherwise, only one sector is allocated. Each sector of allocation corresponds to 252 bytes of data.

A contiguous file facilitates random access to file data since PDOS can directly position to any byte within the file without having to follow sector links. A contiguous file is automatically changed to a non-contiguous file if it is extended with non-contiguous sectors.

See also: None

Possible Errors:

50 = Invalid file name
51 = File already defined
55 = Fragmentation error
57 = File directory full
61 = File already open
68 = Not PDOS disk
Disk errors
1.3.19 XDLF - DELETE FILE

Mnemonic: XDLF
Value: $A0D6
Module: MPDOSF
Format: XDLF
<status error return>

Registers: In (A1) = File name

Description: DELETE FILE primitive removes the file whose name is pointed to by address register A1 from the disk directory and releases all sectors associated with that file for use by other files on that same disk. A file cannot be deleted if it is delete (*) or write (**) protected.

See also: None

Possible Errors:

50 = Invalid file name
53 = File not defined
58 = File delete or write protected
61 = File already open
68 = Not PDOS disk
Disk errors
1.3.20 XDMP - DUMP MEMORY FROM STACK

Mnemonic: XDMP
Value: $A04A
Module: MPDOSK3
Format: XDMP

Registers: In USP.L = <# of bytes>.W
<start address>.L
Out USP.L = USP.L + 6

Description: DUMP MEMORY FROM STACK primitive dumps a block of memory to the console as specified by two parameters on the user stack (USP). The left side of the output is a hexadecimal dump and the right side is a masked ($7F) ASCII dump.

To use this primitive, first push a 32-bit address and then a 16-bit number of the amount of memory to be dumped. The primitive will automatically clean up the user stack.

See also: None

Possible Errors: None
1.3.21 XDPE - DELAY PHYSICAL EVENT

Mnemonic: XDPE
Value: $A114
Module: MPDOSK1
Format: XDPE

Registers: In
A0 = Event address
D0.L = Time TICs for delay (0=clear entry)
D1.W = Event descriptor

Description: Causes the specified event to be set/cleared after the specified time has elapsed. Each event can have only one delayed action pending. Successive calls will supersede pending requests. Only the lower eight bits of the descriptor are used. To cancel pending actions, specify a delay time of 0.

The event descriptor is a 16-bit word that defines both the bit number at the specified A0 address and the action to take on the bit. The following bits are defined:

Bit number - 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
T x x x x x x x S x x x x B B B

T = Should the bit be toggled on scheduling?
1 = Yes (toggle), 0 = No (do not toggle)

S = Suspend on event bit clear or set
1 = Suspend on SET, 0 = Suspend on CLEAR

BBB = The 680 x 0 bit number to use as an event
x = Reserved, should be 0

Since the bit number is specified in the lower three bits of the descriptor, you may use the descriptor with the 680 x 0 BTST, BCLR, BSET instructions.

See also: XDEV - Delay Set/Clear Event
XSOE - Suspend on Physical Event
XTLP - Translate Logical to Physical Event
1.3.22 XDTV - DEFINE TRAP VECTORS

Mnemonic: XDTV  
Value: $A024  
Module: MPDOSK1  
Format: XDTV  

Registers: In  

D1.L = TVCZ FEDC BA98 7654 3210  
(A0) = Table base address  
(A1) = Vector table address  

Vector table: DC.L TRAP #0-<BASE ADR>  
....  
DC.L TRAP #15-<BASE ADR>  
DC.L ZDIV-<BASE ADR>  
DC.L CHK-<BASE ADR>  
DC.L TRAPV-<BASE ADR>  
DC.L TRACE-<BASE ADR>  

Note: The vector table size is variable and each entry corresponds to non-zero bits in the mask register (D1.L). Each entry is a long signed displacement from the base address register.  

D1.L = TVCZ FEDCBA9876543210  

TRAPs #0-#15  
Zero divide  
CHK  
TRAPV  
Trace exception  

Description: The DEFINE TRAP VECTORS primitive loads user routine addresses into the task control block exception vector variables. Each task has the option to process its own TRAP, zero divide, CHK, TRAPV, and/or trace exceptions.  

Data register D1 selects which vectors are to be loaded according to individual bits corresponding to vectors in the vector table pointed to by address register A1. Bits 0 through 19 (right to left) correspond to TRAPs 0 through 15, zero divide, CHK, TRAPV, and trace exceptions. A 1 bit moves a vector from the vector table (biased by base address A0) into the task control block.  

When an exception occurs, the task control block is checked for a corresponding non-zero exception vector. If found, then the return address is pushed on the user stack (USP) followed by the exception address and condition codes. PDOS next moves to user mode and executes a return with condition codes (RTR). This effectively acts like a jump subroutine with the return address on the user stack.
The trace processing is handled differently. If the processor is in supervisor mode when a trace exception occurs, the trace bit is cleared and the exception is dismissed. The processor remains in supervisor mode. If the processor is in user mode and there is a non-zero trace variable in the task control block, then the trace is again disabled, the trace processor address is pushed on the supervisor stack along with status, and a return from exception is executed (RTE).

See also:

Possible Errors: None
1.3.23 XERR - RETURN ERROR D0 TO VMEPROM

Mnemonic: XERR
Value: $A00C
Module: MPDOSK1
Format: XERR

Registers: In D0.W = Error code

Description: RETURN ERROR D0 TO VMEPROM primitive exits to VMEPROM and passes an error code in data register D0. PDOS prints 'PDOS ERR', followed by the decimal error number. The error call can be intercepted by changing the value of the ERR$ variable in the task TCB. This allows you to customize your own monitor.

See also: XEXT - EXIT TO VMEPROM

Possible Errors: None
1.3.24 XEXC - EXECUTE PDOS CALL D7.W

Mnemonic: XEXC
Value: $A030
Module: MPDOSK1
Format: XEXC

Registers: In D7.W = Aline PDOS CALL

Description: EXECUTE PDOS CALL D7.W primitive executes a variable PDOS primitive contained in data register D7. Any registers or error conditions apply to the corresponding PDOS call.

See also: Possible Errors: Call dependent
1.3.25 XEXT - EXIT TO VMEPROM

Mnemonic: XEXT
Value: $A00E
Module: MPDOSK1
Format: XEXT
(Always exits to VMEPROM)

Registers: None

Description: EXIT TO VMEPROM primitive exits a user program and returns to VMEPROM.

The exit can be intercepted by changing the value of the EXT$ variable in the task TCB.
This primitive allows you to customize your own monitor.

See also: XERR - RETURN ERROR D0 TO VMEPROM

Possible Errors: None
1.3.26  XFAC - FILE ALTERED CHECK

Mnemonic:  XFAC
Value:  $A0CE
Module:  MPDOSF
Format:  XFAC
<status error return>

Registers:  In  (A1) = FILE NAME
Out  CC = File not altered
     CS = File altered
     NE = Error

Description:  FILE ALTERED CHECK primitive looks at the altered bit (bit $80) of the file pointed to by address register A1. If the bit is zero (not altered), then the primitive returns with the carry status bit clear. If the alter bit is set (file altered), then it is cleared and the primitive returns with carry set. If either case, the bit is always cleared.

See also:  None

Possible Errors:  Disk errors
1.3.27 XFBF - FLUSH BUFFERS

Mnemonic: XFBF
Value: $A0F8
Module: MPDOSF
Format: XFBF
<status error return>

Registers: None

Description: FLUSH BUFFERS primitive forces all file slots with active channel buffers to write any updated data to the disk. It thus does a checkpoint of any open and altered file.

See also: None

Possible Errors: Disk errors
1.3.28 XFFN - FIX FILE NAME

Mnemonic: XFFN
Value: $A0A0
Module: MPDOSF
Format: XFFN
<status error return>

Registers: In (A1) = File name
Out D0.L = Disks(4th/3rd/2nd/1st)
(A1) = MWB$, Fixed file name

Description: FIX FILE NAME primitive parses a character string for file name, extension, directory level, and disk number. The results are returned in the 32-character monitor work buffer (MWB$(A6)). Data register D0 is also returned with the disk number. The error return is used for an invalid file name.

The monitor work buffer is cleared and the following assignments are made:

0(A1) = File name
8(A1) = File extension
11(A1) = File directory level

System defaults are used for the disk number and file directory level when they are not specified in the file name.

See also: XRDN - READ DIRECTORY ENTRY BY NAME

Possible Errors:

50 = Invalid file name
1.3.29 XFTD - FIX TIME & DATE

Mnemonic: XFTD
Value: $A058
Module: MPDOSK3
Format: XFTD

Registers: Out
D0.W = Hours * 256 + Minutes
D1.W = (Year * 16 + Month) * 32 + Day

Description: FIX TIME & DATE primitive returns a two-word encoded time and date generated from the system timers. The resultant codes include month, day, year, hours, and minutes. The ordinal codes can be sorted and used as inputs to the UNPACK DATE (XUDT) and UNPACK TIME (XUTM) primitives.

Data register D0.W contains the time and register D1.W contains the date. This format is used throughout PDOS for time stamping items.

See also: XPAD - PACK ASCII DATE
XRDT - READ DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE
XUTM - UNPACK TIME

Possible Errors: None
1.3.30 XFUM - FREE USER MEMORY

Mnemonic: XFUM
Value: $A040
Module: MPDOSK1
Format: XFUM
<status error return>

Registers: In
D0.W = Number of K bytes
(A0) = Beginning address

Description: FREE USER MEMORY primitive deallocates user memory to the system memory bit map. Data register D0.W specifies how much memory is to be deallocated while address register A0 points to the beginning of the data block.

Memory thus deallocated is available for any task use including new task creation.

Possible Errors:

79 = Memory error
1.3.31 XGCB - CONDITIONAL GET CHARACTER

Mnemonic: XGCB
Value: $A048
Module: MPDOSK2
Format: XGCB
<status return>

Registers: Out D0.L = Character in bits 0-7
SR = EQ....No character
LO....[CTRL-C]
LT....[ESC]
MI....[CTRL-C] or [ESC]

Note: If the ignore control character bit ($02) of the port flag is set, then XGCB ignores [CTRL-C] and [ESC].

Description: CONDITIONAL GET CHARACTER primitive checks for a character from first, the input message pointer (IMP$(A6)), second, the assigned input file (ACI$(A6)), and then finally, the interrupt driven input character buffer (PRT$(A6)). If a character is found, it is returned in the right byte of data register D0.L and the rest of the register is cleared.

If there is no input message, no assigned console port character, and the interrupt buffer is empty, the status is returned as 'EQ'.

The status is returned 'LO' and the break flag cleared if the returned character is a [CTRL-C]. The input buffer is also cleared. Thus, all characters entered after the [CTRL-C] and before the XGCB call are dropped.

The status is returned 'LT' and the break flag cleared if the returned character is the [ESC] character.

For all other characters, the status is returned 'HI' and 'GT'. The break flag is not affected.

Possible Errors: None
1.3.32 XGCC - GET CHARACTER CONDITIONAL

Mnemonic: XGCC
Value: $A078
Module: MPDOSK2
Format: XGCC
<status return>

Registers: Out D0.L = Character in bits 0-7
           SR = EQ....No character
           LO....[CTRL-C]
           LT....[ESC]
           MI....[CTRL-C] or [ESC]

Note: If the ignore control character bit ($02) of the port flag is set, then XGCC ignores [CTRL-C] and [ESC].

Description: GET CHARACTER CONDITIONAL primitive checks the interrupt driven input character buffer and returns the next character in the right byte of data register D0.L. The rest of the register is cleared. The input buffer is selected by the input port variable (PRT$) of the TCB.

If the buffer is empty, the 'EQ' status bit is set. If the character is a [CTRL-C], then the break flag and input buffer are cleared, and the status is returned 'LO'. If the character is the [ESC] character, then the break flag is cleared and the status is returned 'LT'.

If no special character is encountered, the character is returned in register D0 and the status set 'HI' and 'GT'.

If no port has been assigned for input (ie. port 0 or phantom port), then the routine always returns an 'EQ' status.

Possible Errors: None
1.3.33 XGCP - GET PORT CHARACTER

Mnemonic: XGCP
Value: $A09E
Module: MPDOSK2
Format: XGCP
<status return>

Registers: Out D0.L = Character in bits 0-7
SR = LO...[CTRL-C]
LT....[ESC]
MI....[CTRL-C] or [ESC]

Note: If the ignore control character bit ($02) of the port flag is set, then XGCP ignores [CTRL-C] and [ESC].

Description: GET PORT CHARACTER primitive checks for a character in the interrupt driven input character buffer. If a character is found, it is returned in the right byte of data register D0.L and the rest of the register is cleared. The input buffer is selected by the input port variable (PRT$) of the TCB.

If the interrupt buffer is empty, the task is suspended pending a character interrupt.

The status is returned 'LO' and the break flag cleared if the returned character is a [CTRL-C]. The input buffer is also cleared. Thus, all characters entered after the [CTRL-C] and before the XGCR call are dropped.

The status is returned 'LT' and the break flag cleared if the returned character is the [ESC] character.

For all other characters, the status is returned 'HI' and 'GT'. The break flag is not affected.

If no port has been assigned for input, (ie. port 0 or phantom port), then an error 86 occurs.

Possible Errors: None
1.3.34 XGCR - GET CHARACTER

Mnemonic: XGCR
Value: $A07A
Module: MPDOSK2
Format: XGCR

<status return>

Registers:

Out  D0.L = Character in bits 0-7
SR = LO....[CTRL-C]
LT....[ESC]
MI....[CTRL-C] or [ESC]

Note:
If the ignore control character bit ($02) of the port flag is set, then XGCR ignores
[CTRL-C] and [ESC].

Description: GET CHARACTER primitive checks for a character from first, the input message pointer
(IMP$(A6)); second, the assigned input file (ACI$(A6)); and then finally, the interrupt
driven input character buffer (PRT$(A6)). If a character is found, it is returned in the right
byte of data register D0.L and the rest of the register is cleared.

If there is no input message, no assigned console port character, and the interrupt buffer
is empty, the task is suspended pending a character interrupt.

The status is returned 'LO' and the break flag cleared if the returned character is a
[CTRL-C]. The input buffer is also cleared. Thus, all characters entered after the
[CTRL-C] and before the XGCR call are dropped.

The status is returned 'LT' and the break flag cleared if the returned character is the
[ESC] character.

For all other characters, the status is returned 'HI' and 'GT'. The break flag is not
affected.

If no port has been assigned for input, (ie. port 0 or phantom port), then an error 86
occurs.

Possible Errors: None
1.3.35 XGLB - GET LINE IN BUFFER

Mnemonic: XGLB
Value: $A07C
Module: MPDOSK2
Format: XGLB
{BLT.x ESCAPE} optional
<status return>

Registers:
In  (A1) = Buffer address
Out D1.L = Number of characters
      SR = EQ...[CR] only
      LT...[ESC]
      LO...[CTRL-C]

Note: If the ignore control character bit ($02) of the port flag is set, then XGLB ignores [CTRL-C] and [ESC].

Description: LINE IN BUFFER primitive gets a character line into the buffer pointed to by address register A1. The XGCR primitive is used by XGLB and hence characters can come from a memory message, a file, or the task console port.

The buffer must be at least 80 characters in length. The line is delimited by a carriage return. The status returns EQUAL if only a [CR] is entered.

If an [ESC] is entered, the task exits to VMEPROM unless a 'BLT' instruction immediately follows the XGLB call. If such is the case, then XGLB returns with status set at 'LT'.

If the assigned console flag (ACI$(A6)) is set, then the ‘&’ character is used for character substitutions. ‘&0’ is replaced with the last system error number. ‘&1’ is replaced with the first parameter of the command line, ‘&2’ with the second, and so forth up to ‘&9’.

The command line can be edited with various system defined control characters. A [BACKSPACE] ($08) moves the cursor one character to the left. A [CTRL-F] ($0C) moves the cursor one character to the right. A [RUB] ($7F) deletes one character to the left. A [CTRL-D] ($04) deletes the character under the cursor. The cursor need not be at the end of the line when the [CR] is entered.

See also: XGLU - GET LINE IN USER BUFFER

Possible Errors: None
1.3.36 XGLM - GET LINE IN MONITOR BUFFER

Mnemonic: XGLM
Value: $A07E
Module: MPDOSK2
Format: XGLM
   \{BLT.x ESCAPE\} optional
   \<status return>\n
Registers: Out \( (A1) = \) String
   D1.L = Number of characters
   SR = EQ...[CR] only
   LT...[ESC]
   LO...[CTRL-C]

Note: If the ignore control character bit ($02) of the port flag is set, then XGLM ignores [CTRL-C] and [ESC].

Description: The GET LINE IN MONITOR BUFFER primitive gets a character line into the monitor buffer located in the task control block. The XGCR primitive is used by XGLM and hence, characters can come from a memory message, a file, or the task console port.

The buffer has a maximum length of 80 characters and is delimited by a carriage return. The status returns EQUAL if only a [CR] is entered. If an [ESC] is entered, the task exits to VMEMPR unless a 'BLT' instruction immediately follows the XGLM call. If such is the case, then XGLM returns with status set at 'LT'.

If the assigned console flag (ACI$(A6)) is set, then the '&' character is used for character substitutions. '&0' is replaced with the last system error number. '&1' is replaced with the first parameter of the command line, '&2' with the second, and so forth up to '&9'.

The command line can be edited with various system-defined control characters. A [BACKSPACE] ($08) moves the cursor one character to the left. A [CTRL-L] ($0C) moves the cursor one character to the right. A [RUB] ($7F) deletes one character to the left. A [CTRL-D] ($04) deletes the character under the cursor. The cursor need not be at the end of the line when the [CR] is entered.

The last command line can be recalled to the buffer by entering a [CTRL-A] ($01). This line can then be edited using the above control characters.

Possible Errors: None
1.3.37 XGLU - GET LINE IN USER BUFFER

Mnemonic: XGLU
Value: $A080
Module: MPDOSK2
Format: XGLU
{BLT.x ESCAPE ;optional}
<status return>

Registers: Out (A1) = String
D1.L = Number of characters
SR = EQ...[CR] only
LT...[ESC]
LO...[CTRL-C]

Note: If the ignore control character bit ($02) of the port flag is set, then XGLU ignores [CTRL-C] and [ESC].

Description: The GET LINE IN USER BUFFER primitive gets a character line into the user buffer. Address register A6 normally points to the user buffer. The XGCR primitive is used by XGLU; hence, characters come from a memory message, a file, or the task console port. The line is delimited by a carriage return. The status returns EQUAL if only a [CR] is entered. Address register A1 is returned with a pointer to the first character.

The user buffer is located at the beginning of the task control block and is 256 characters in length. However, the XGLU routine limits the number of input characters to 78 plus two nulls.

If an [ESC] ($1B) is entered, the task exits to VMEPROM unless a 'BLT' instruction immediately follows the XGLU call. If such is the case, then XGLU returns with status set at 'LT'.

If the assigned console flag (ACI$(A6)) is set, then the '&' character is used for character substitutions. '&0' is replaced with the last system error number. '&1' is replaced with the first parameter of the command line, '&2' with the second, and so forth up to '&9'.

The command line can be edited with various system defined control characters. A [BACKSPACE] ($08) moves the cursor one character to the left. A [CTRL-L] ($0C) moves the cursor one character to the right. A [RUB] ($7F) deletes one character to the left. A [CTRL-D] ($04) deletes the character under the cursor. The cursor need not be at the end of the line when the [CR] is entered.

Possible Errors: None
1.3.38 XGML - GET MEMORY LIMITS

Mnemonic: XGML
Value: $A010
Module: MPDOSK1
Format: XGML

Registers: Out
(A0) = End TCB (TBE$)
(A1) = Upper memory limit (EUM$-USZ)
(A2) = Last loaded address (BUM$)
(A5) = System RAM (SYRAM)
(A6) = Task TCB

Description: GET MEMORY LIMITS subroutine returns the user task memory limits. These limits are defined as the first usable location after the task control block ($500 beyond address register A6) and the end of the user task memory. The task may use up to but not including the upper memory limit.

Address register A0 is returned pointing to the beginning of user storage (which is the end of the TCB). Register A1 points to the upper task memory limit less $100 hexadecimal bytes for the user stack pointer (USP). Register A2 is the last loaded memory address as provided by the PDOS loader. Address registers A5 and A6 are returned with the pointers to system RAM (SYRAM) and the task control block (TCB).

Possible Errors: None
1.3.39 XGMP - GET MESSAGE POINTER

Mnemonic: XGMP
Value: $A004
Module: MPDOSK1
Format: XGMP
<status return>

Registers: In  D0.L = Message slot number (0..15)
Out  D0.L = Source task # (-1 = no message)
      SR = EQ....Message (Event[64+Message slot#]=0)
      NE....No message
      D0.L = Error number 83 if no message
      (A1) = Message

Description: GET MESSAGE POINTER primitive looks for a task message pointer. If no message is
ready, then data register D0 returns with a minus one (-1) and status is set to 'Not
Equal'.

If a message is waiting, then data register D0 returns with the source task number,
address register A1 returns with the message pointer, event (64 + message slot #) is set
to zero indicating message received, and status is returned equal.

See also: XGTM - GET TASK MESSAGE
          XKTM - KILL TASK MESSAGE
          XSMP - SEND MESSAGE POINTER
          XSTM - SEND TASK MESSAGE

Possible Errors:

    83 = Message slot empty
1.3.40 XGNP - GET NEXT PARAMETER

Mnemonic: XGNP
Value: $A05A
Module: Emulated by VMEPROM
Format: XGNP
<status return>

Registers: Out SR = LO....No parameter
[(A1)=0]
EQ....Null Parameter
[(A1)=0]
HI....Parameter
[(A1)=PARAMETER]

Description: GET NEXT PARAMETER primitive parses the VMEPROM command buffer for the next command parameter. The XGNP primitive clears all leading spaces of a parameter. A parameter is a character string delimited by a space, comma, period, or null. If a parameter begins with a left parenthesis, then all parsing stops until a matching right parenthesis or null is found. Hence, spaces, commas, and periods are passed in a parameter when enclosed in parentheses. Parentheses may be nested to any depth.

A 'LO' status is returned if the last parameter delimiter is a null or period. XGNP does not parse past a period. In this case, address register A1 is returned pointing to a null string.

An 'EQ' status is returned if the last parameter delimiter is a comma and no parameter follows. Address register A1 is returned pointing to a null string.

A 'HI' status is returned if a valid parameter is found. Address register A1 then points to the parameter.

Possible Errors: None
1.3.41 XGTM - GET TASK MESSAGE

Mnemonic: XGTM
Value: $A01E
Module: MPDOSK1
Format: XGTM
<status return>

Registers:  
In (A1) = Buffer address
Out D0.L = Source task #
(-1 = no message)
SR = EQ....message found
     NE....no message

Description: GET TASK MESSAGE primitive searches the PDOS message buffers for a message with a destination equal to the current task number. If a message is found, it is moved to the buffer pointed to by address register A1. The message buffer is then released, and the status is set EQUAL. If no message is found, status is returned NE.

The buffer must be at least 64 bytes in length. (This is a configuration parameter.) The message buffers are serviced on a first in, first out basis (FIFO). Messages are data independent and pass any type of binary data.

See also: XGMP - GET MESSAGE POINTER
XKTM - KILL TASK MESSAGE
XSMP - SEND MESSAGE POINTER
XSTM - SEND TASK MESSAGE

Possible Errors: None
**1.3.42 XGUM - GET USER MEMORY**

Mnemonic: XGUM  
Value: $A03E  
Module: MPDOSK1  
Format: XGUM  
<status error return>

Registers:  
In  
D0.W = Number of K bytes  
Out  
(A0) = Beginning memory address  
(A1) = End memory address

Description: GET USER MEMORY primitive searches the system memory bit map for a contiguous block of memory equal to D0.W Kbytes. If found, the 'EQ' status is set, address registers A0 and A1 are returned the start and end memory address, and the memory block is marked as allocated in the bit map.

See also: XFUM - FREE USER MEMORY

Possible Errors:  
73 = Not enough memory
1.3.43 XISE - INITIALIZE SECTOR

Mnemonic: XISE
Value: $A0C0
Module: MPDOSF
Format: XISE
<status error return>

Registers: In
D0.B = Disk number
D1.W = Logical sector number
(A2) = Buffer address

Description: INIT SECTOR primitive is a system-defined, hardware-dependent program which writes 256 bytes of data from a buffer (A2) to a logical sector number (D1) on disk (D0). This routine is meant to be used only for disk initialization and is equivalent to the WRITE SECTOR (XWSE) primitive for all sectors except 0. Sector 0 is not checked for the PDOS ID code.

See also: XRSE - READ SECTOR
XRSZ - READ SECTOR ZERO
XWSE - WRITE SECTOR

Possible Errors:
Disk errors
1.3.44 XKTB - KILL TASK

Mnemonic: XKTB
Value: $A0FA
Module: MPDOSK1
Format: XKTB
<status error return>

Registers: In D0.B = Task number

Note: If D0.B equals zero, then kill current task. If D0.B is negative, then kill task without allocating task memory to system bit map.

Description: KILL TASK primitive removes a task from the PDOS task list and optionally returns the task’s memory to the system memory bit map. Only the current task or a task spawned by the current task can be killed. Task 0 cannot be killed.

The kill process includes releasing the input port assigned to the task and closing all files associated with the task.

The task number is specified in data register D0.B. If register D0.B equals zero, then the current task is killed and its memory deallocated in the system memory bit map.

If D0.B is positive, then the selected task is killed and its memory deallocated. If D0.B is negative, then task number ABS(D0.B) is killed, but its memory is not deallocated in the memory bit map.

See also: XCTB - CREATE TASK BLOCK

Possible Errors:

74 = No such task
76 = Task locked
1.3.45 XKTM - KILL TASK MESSAGE

Mnemonic: XKTM
Value: $A028
Module: MPDOSK1
Format: XKTM
<status return>

Registers:
In D0.B = Task #
(A1) = Buffer address
Out D0.L = Source task #
(-1 = no message)
SR = EQ....message found
     NE....no message

Description: KILL TASK MESSAGE primitive allows you to read (and thus clear) any task’s messages from the system message buffers.

See also: XGMP - GET MESSAGE POINTER
XGTM - GET TASK MESSAGE
XSMP - SEND MESSAGE POINTER
XSTM - SEND TASK MESSAGE

Possible Errors: None
1.3.46 XLDF - LOAD FILE

Mnemonic: XLDF  
Value: $A0B0  
Module: MPDOSF  
Format: XLDF
<status error return>

Registers:  
In  
D1.B = Execution flag  
(A0) = Start of load memory  
(A1) = End of load memory  
(A3) = File name  
  
Out  
(A0) = EAD$ - Lowest loaded address  
(A1) = BUM$ - Last loaded address

Note: If D1.B=0, then XLDF returns to your calling program. If D1.B<>0, then the program is immediately executed.

Description: LOAD FILE primitive reads and loads 68000 object code into user memory. The file name pointer is passed in address register A3. Registers A0 and A1 specify the memory bounds for the relocatable load. The file must be typed 'OB' or 'SY'. If data register D1.B is zero, then XLDF returns to the calling program. Otherwise, the loaded program is immediately executed.

The 68000 object should be position-independent section 0 code without any external references or definitions.

A 'SY' file is generated from an 'OB' file by the MSYFL utility. The condensed object is a direct memory image and must be position-independent code.

The XLDF primitive uses long word moves and may move up to three bytes more than contained in an 'SY' file. As such, you must allow for extra space for data moves to an existing program.

Possible Errors:

63 = Illegal object tag  
64 = Illegal section  
65 = File not loadable  
71 = Exceeds task size  
73 = Not enough memory  
Disk errors
1.3.47 XLER - LOAD ERROR REGISTER

Mnemonic: XLER
Value: $A03A
Module: MPDOSK1
Format: XLER

Registers: In D0.W = Error number

Description: LOAD ERROR REGISTER primitive stores data register D0.W in the task control block variable LEN$(A6). This variable will replace the parameter substitution variable '&0' during a procedure file.

User programs should execute this call when an error occurs.

The enable echo flag (ECF$(A6)) is cleared by this call.

Possible Errors: None
1.3.48 **XLFN - LOOK FOR NAME IN FILE SLOTS**

Mnemonic: XLFN  
Value: $A0A2  
Module: MPDOSF  
Format: XLFN

<status return>

Registers:  
In  
D0.B = Disk number  
(A1) = Fixed file name  
Out  
D3.W = File ID (Disk #/Index)  
(A3) = Slot entry address  
SR = NE...File name not found  
EQ...File name found

Note: If D3.W=0, then no slots are available.

Description: LOOK FOR NAME IN FILE SLOTS primitive searches through the file slot table for the file name as specified by registers D0.B and A1. If the name is not found, register D3.W returns with a -1 or 0. The latter indicates the file was not found and there are no more slots available. Otherwise, register D3.W returns the associated file ID and register A3 returns the address of the file slot.

A file slot is a 38-byte buffer where the status of an open file is maintained. There are 32 file slots available. The file ID consists of the disk # and the file slot index.

File slots assigned to read-only files are skipped and not considered for file match.

Possible Errors: None
1.3.49 XLKF - LOCK FILE

Mnemonic: XLKF
Value: $A0D8
Module: MPDOSF
Format: XLKF
<status error return>

Registers: In D1.W = File ID

Description: LOCK FILE primitive locks an opened file so that no other task can gain access until an UNLOCK FILE (XULF) primitive is executed. Only the locking task has access to the locked file.

A locked file is indicated by a -1 ($FF) in the left byte of the lock file parameter (LF) of the file slot usage (FS) command. The locking task number is stored in the left byte of the task number parameter (TN).

See also: XULF - UNLOCK FILE

Possible Errors:

52 = File not open
59 = Invalid slot #
75 = File locked
Disk errors
1.3.50 XLKT - LOCK TASK

Mnemonic:  XLKT
Value:     $A014
Module:    MPDOSK1
Format:    XLKT
           <status return>

Registers: Out  SR =  EQ...Not locked
            NE...Locked

Description: LOCK TASK primitive locks the requesting task in the run state by setting the swap lock variable in system RAM to nonzero. The task remains locked until an UNLOCK TASK (XULT) is executed. The status of the lock variable BEFORE the call is returned in the status register.

XLKT waits until all locks (Level 2 and Level 3 locks) are cleared before the task is locked.

See also:  XULT - UNLOCK TASK

Possible Errors:  None
1.3.51 XLSR - LOAD STATUS REGISTER

Mnemonic: XLSR  
Value: $A02E  
Module: MPDOSK1  
Format: XLSR  

Registers: In D1.W = 68000 status register

Description: LOAD STATUS REGISTER primitive allows you to directly load the 68000 status register. Of course, only appropriate bits (i.e. the interrupt mask too high, supervisor mode, trace mode, etc.) are to be set so that the system is not crashed.

See also: XSUP - ENTER SUPERVISOR MODE

Possible Errors: None
1.3.52 XNOP - OPEN SHARED RANDOM FILE

Mnemonic: XNOP
Value: $A0DA
Module: MPDOSF
Format: XNOP
<status error return>

Registers: In (A1) = File name
Out D0.W = File attribute
D1.W = File ID

Notes: Uses multiple directory file search. You MUST lock and position file before each multitask access.

Description: OPEN SHARED RANDOM FILE primitive opens a file for shared random access by assigning the file to an area of system memory called a file slot. The file ID and file attribute are returned to the calling program in registers D1 and D0, respectively. Thereafter, the file is referenced by the file ID and not by the file name. A new entry in the file slot table is made only if the file is not already opened for shared access.

The file ID (returned in register D1) is a 2-byte number. The left byte is the disk number and the right byte is the file slot index. The file attributes are returned in register D0.

The END-OF-FILE marker on a shared file is changed only when the file has been extended. All data transfers are buffered through a channel buffer; data movement to and from the disk is by full sectors.

An "opened count" is incremented each time the file is shared-opened and is decremented by each close operation. The file is only closed by PDOS when the count is zero. This count is saved in the right byte of the locked file parameter (LF) and is listed by the file slot usage command (FS).

Possible Errors:

50 = Invalid file name
53 = File not defined
60 = File space full
61 = File already open
68 = Not PDOS disk
69 = Not enough file slots
Disk errors
1.3.53 XPAD - PACK ASCII DATE

Mnemonic: XPAD
Value: $A00A
Module: MPDOSK3
Format: XPAD

Registers:
In (A1) = 'DY-MON-YR'
Out D1.W = (Year*16+month)*32+day
(YYYY YYYM MMMD DDDD)
(A1) = Updated
SR = .EQ. - Conversion ok
     .NE. - Error

Description: PACK ASCII DATE primitive converts an ASCII date string to an encoded binary number in data register D1. The result is compatible with other PDOS date primitives such as XUAD.

See Also: XFTD - FIX TIME & DATE
XRDT - READ DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE

Possible Errors: Status errors.
1.3.54 XPBC - PUT BUFFER TO CONSOLE

Mnemonic: XPBC
Value: $A084
Module: MPDOSK2
Format: XPBC

Registers: None

Description: PUT USER BUFFER TO CONSOLE primitive outputs the ASCII contents of the user buffer to the user console and/or SPOOL file. The output string is delimited by the null character. The user buffer is the first 256 bytes of the task control block and is pointed to by address register A6. With the exception of control characters and characters with the parity bit on, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields. If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XGLB - GET LINE IN BUFFER

Possible Errors: None
### 1.3.55 XPCC - PUT CHARACTER(S) TO CONSOLE

**Mnemonic:** XPCC  
**Value:** $A086  
**Module:** MPDOSK2  
**Format:** XPCC  

**Registers:** In D0.W = Character(s)

**Description:** PUT CHARACTER TO CONSOLE primitive outputs one or two ASCII characters in data register D0 to the user console and/or SPOOL file. The right byte (bits 0 through 7) is first and is followed by the left byte (bits 8 through 15) if non-zero. If the right byte or both bytes are zero, nothing is output to the console.

With the exception of control characters and characters with the parity bit on, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

**See also:** XPCR - PUT CHARACTER RAW  
XPDC - PUT DATA TO CONSOLE

**Possible Errors:** None
1.3.56  XPCL - PUT CRLF TO CONSOLE

Mnemonic:   XPCL
Value:      $A088
Module:     MPDOSK2
Format:     XPCL

Registers:  None

Description: PUT CRLF TO CONSOLE primitive outputs the ASCII characters carriage return <$0A>
and line feed <$0D> to the user console and/or SPOOL file. The column counter is cleared.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of
the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6))
and are not sent to the corresponding output ports. If a disk error occurs in the spool file,
then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

Possible Errors: None
1.3.57 XPCP - PLACE CHARACTER IN PORT BUFFER

Mnemonic: XPCP
Value: $AOBC
Module: MPDOSK2
Format: XPCP

Registers:

In
D0.B = Character to insert
D1.W = Input port number (1 to 15)

Out
SR = .EQ. = High water (character is inserted)
     .NE. = Character is inserted

Description: XPCP allows a character to be placed into the input buffer of any VMEPROM port from a task or program.

Note: Once the status returns EQ (high water), subsequent XPCP calls will return a status of NE as if everything were normal, but the data is discarded. Once the status of EQ is detected, the transmitting task should monitor the status of the port with the XRPS (read port status) call until bit 56 is cleared.

The port specified in the XPCP call is independent of window g - it refers to the physical port, not the logical port.
1.3.58 XPCR - PUT CHARACTER RAW

Mnemonic: XPCR
Value: $A0BA
Module: MPDOSK2
Format: XPCR

Registers: In D0.B = CHARACTER

Description: The PUT CHARACTER RAW primitive outputs the character in the lower byte of data register D0 to the user console. No attempt is made by PDOS to interpret control characters.

See also: XPCC - PUT CHARACTER(S) TO CONSOLE
XPDC - PUT DATA TO CONSOLE

Possible Errors: None
1.3.59 XPDC - PUT DATA TO CONSOLE

Mnemonic: XPDC
Value: $A096
Module: MPDOSK2
Format: XPDC

Registers: In D7.W = LENGTH
(A1) = DATA STRING

Description: PUT DATA TO CONSOLE primitive outputs data-independent bytes to the console. Address register A1 points to the string while data register D7 has the string length.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPCC - PUT CHARACTER(S) TO CONSOLE
XPCR - PUT CHARACTER RAW

Possible Errors: None
1.3.60 XPEL - PUT ENCODED LINE TO CONSOLE

Mnemonic: XPEL
Value: $A06E
Module: MPDOSK2
Format: XPEL

Registers: In (A1) = Message

Description: PUT ENCODED LINE TO CONSOLE primitive outputs to the user console the message pointed to by address register A1. An encoded message is similar to any other string with the exception that the parity bit is used to output blanks and the character $80 outputs a carriage return/line feed.

If the parity bit is set and the masked character ($7F) is less than or equal to a blank, then the numeric value of the negated character is used as the number of blanks to be inserted in the output stream. If the mask character is greater than a blank, then that character is output followed by one blank.

With the exception of control characters, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPEM - PUT ENCODED MESSAGE TO CONSOLE
XPLC - PUT LINE TO CONSOLE
XPMC - PUT MESSAGE TO CONSOLE

Possible Errors: None
1.3.61 XPEM - PUT ENCODED MESSAGE TO CONSOLE

Mnemonic: XPEM
Value: $A09C
Module: MPDOSK2
Format: XPEM <message>

Registers: None

Description: PUT ENCODED MESSAGE TO CONSOLE primitive outputs to the user console the PC relative message contained in the word following the call. An encoded message is similar to any other string with the exception that the parity bit is used to output blanks and the character $80 outputs a carriage return/line feed.

If the parity bit is set and the masked character ($7F) is less than or equal to a blank, then the numeric value of the negated character is used as the number of blanks to be inserted in the output stream. If the mask character is greater than a blank, then that character is output followed by one blank.

With the exception of control characters, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPEL - PUT ENCODED LINE TO CONSOLE
XPLC - PUT LINE TO CONSOLE
XPMC - PUT MESSAGE TO CONSOLE

Possible Errors: None
1.3.62 XPLC - PUT LINE TO CONSOLE

Mnemonic: XPLC
Value: $A08A
Module: MPDOSK2
Format: XPLC

Registers: In (A1) = ASCII string

Description: PUT LINE TO CONSOLE primitive outputs the ASCII character string pointed to by address register A1 to the user console and/or SPOOL file. The string is delimited by the null character.

With the exception of control characters and characters with the parity bit on, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPEL - PUT ENCODED LINE TO CONSOLE
XPEM - PUT ENCODED MESSAGE TO CONSOLE
XPMC - PUT MESSAGE TO CONSOLE

Possible Errors: None
1.3.63 XPMC - PUT MESSAGE TO CONSOLE

Mnemonic: XPMC
Value: $A08C
Module: MPDOSK2
Format: XPMC <message>

Registers: None

Description: PUT MESSAGE TO CONSOLE primitive outputs the ASCII character string pointed to by the message address word immediately following the PDOS call to the user console and/or SPOOL file. The address is a PC relative 16-bit displacement to the message. The output string is delimited by the null character.

With the exception of control characters and characters with the parity bit on, each character increments the column counter by one. A [BACKSPACE] ($08) decrements the counter while a [CR] ($0D) clears the counter. [TAB]s ($09) are expanded with blanks to MOD 8 character zone fields.

If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPEL - PUT ENCODED LINE TO CONSOLE
XPEM - PUT ENCODED MESSAGE TO CONSOLE
XPLC - PUT LINE TO CONSOLE

Possible Errors: None
1.3.64  XPSC - POSITION CURSOR

Mnemonic: XPSC  
Value: $A08E  
Module: MPDOSK2  
Format: XPSC  

Registers: In  
D1.B = Row  
D2.B = Column  

Note: Uses PSC$(A6) as lead characters.  

Description: POSITION CURSOR primitive positions the cursor on the console terminal according to the row and column values in data registers D1 and D2. Register D1 specifies the row on the terminal and generally ranges from 0 to 23, with 0 being the top row. Register D2 specifies the column of the terminal and ranges from 0 to 79, with 0 being the left-hand column. Register D2 is also loaded into the column counter reflecting the true column of the cursor.

The XPSC primitive outputs either one or two leading characters followed by the row and column. The leading characters output by XPSC are located in PSC$(A6) of the task control block. These characters are transferred from the parent task to the spawned task during creation. The initial characters come from the BIOS module.

The row and column characters are biased by $20 if the parity bit of the first character is set. Likewise, if the second character’s parity bit is set, then row/column order is reversed. This accommodates most terminal requirements for positioning the cursor.

If PSC$ is zero, then PDOS makes a call into the BIOS for custom position cursor. The entry point is B_PSC beyond the BIOS table.

The ST command of the user interface can be used to change the position cursor codes.

See also: XCLS - CLEAR SCREEN  
XRCP - READ PORT CURSOR POSITION  

Possible Errors: None
1.3.65 XPSF - POSITION FILE

Mnemonic: XPSF
Value: $A0DC
Module: MPDOSF
Format: XPSF
<status error return>

Registers: In
D1.W = File ID
D2.L = Byte position

Note: A byte position equal to -1 positions to the end of the file.

Description: POSITION FILE primitive moves the file byte pointer to any byte position within a file. The file ID is given in register D1 and the long word byte position is specified in register D2.

An error occurs if the byte position is greater than the current end-of-file marker.

A contiguous file greatly enhances the speed of the position primitive since the desired sector is directly computed. However, the position primitive does work with non-contiguous files, as PDOS follows the sector links to the desired byte position.

A contiguous file is extended by positioning to the end-of-file marker and writing data. However, PDOS will alter the file type to non-contiguous if a contiguous sector is not available. This would result in random access being much slower.

See also: XRFP - READ FILE POSITION
XRWF - REWIND FILE

Possible Errors:

52 = File not open
59 = Invalid slot #
70 = Position error
Disk errors
1.3.66 XPSP - PUT SPACE TO CONSOLE

Mnemonic: XPSP
Value: $A098
Module: MPDOSK2
Format: XPSP

Registers: None

Description: PUT SPACE TO CONSOLE outputs a [SP] ($20) character to the user console. There are no registers or status involved. If there are coinciding bits in the unit (UNT$(A6)) and spool unit (SPU$(A6)) variables of the TCB, then the processed characters are written to the spool unit file slot (SPI$(A6)) and are not sent to the corresponding output ports. If a disk error occurs in the spool file, then all subsequent output characters echo as a bell until the error is corrected by selecting a different UNIT or resetting the SPOOL UNIT.

See also: XPCC - PUT CHARACTER(S) TO CONSOLE

Possible Errors: None
1.3.67 XRBF - READ BYTES FROM FILE

Mnemonic: XRBF
Value: $A0DE
Module: MPDOSF
Format: XRBF

<status error return>

Registers:
In   D0.L = Number of bytes
     D1.W = File ID
     (A2) = R/W buffer address
Out  D3.L = Number of bytes read
       (On EOF only.)

Description: READ BYTES FROM FILE primitive reads the number of bytes specified in register D0 from the file specified by the file ID in register D1 into a memory buffer pointed to by address register A2. If the channel buffer has been rolled to disk, the least-used buffer is freed and the desired buffer is restored to memory. The file slot ID is placed on the top of the last-access queue.

If an error occurs during the read operation, the error return is taken with the error number in register D0 and the number of bytes actually read in register D3.

The read is independent of the data content. The buffer pointer in register A2 is on any byte boundary. The buffer is not terminated with a null.

A byte count of zero in register D0 results in one byte being read from the file. This facilitates single byte data acquisition.

See also: XRLF - READ LINE FROM FILE
          XWBF - WRITE BYTES TO FILE
          XWLF - WRITE LINE TO FILE

Possible Errors:

52 = File not open
56 = End of file
59 = Invalid slot #
Disk errors
1.3.68  XRCN - RESET CONSOLE INPUTS

Mnemonic:  XRCN
Value:  $A0B2
Module:  MPDOSF
Format:  XRCN

Registers:  None

Description:  RESET CONSOLE INPUTS closes the current procedure file. If there are other procedure files pending (nested), then they become active again.

See also:  XCBC - CHECK FOR BREAK CHARACTER

Possible Errors:  None
1.3.69 XRCP - READ PORT CURSOR POSITION

Mnemonic: XRCP
Value: $A092
Module: MPDOSK2
Format: XRCP

Registers:
In   D0.W = Port #
Out  D1.L = Row
     D2.L = Column

Note: If D0.W=0, then the current port (PRT$(A6)) is used.

Description: READ PORT CURSOR POSITION primitive reads the current cursor position for the port designated by data register D0.B. The PDOS system maintains a column count (0-79) and a row count (0-23) for each port. When the cursor reaches row 23, the count is not incremented, acting like a screen scroll.

See also: XCLS - CLEAR SCREEN
          XPSC - POSITION CURSOR

Possible Errors: None
1.3.70 XRDE - READ NEXT DIRECTORY ENTRY

Mnemonic: XRDE
Value: $A0A6
Module: MPDOSF
Format: XRDE
<status error return>

Registers: 
In  
D0.B = Disk number
D1.B = Read flag (0=1st)
(A2) = Last 32 byte directory entry
TW1$ = Sector number
TW2$ = number of directory entries

Out  
D1.W = Sector number
(A2) = Next entry

Description: READ NEXT DIRECTORY ENTRY primitive reads sequentially through a disk directory. If register D1.B is zero, then the routine begins with the first directory entry. If register D1.B is nonzero, then based on the last directory entry (pointed to by register A2), the next entry is read.

The calling routine must maintain registers D0.B and A2, the user I/O buffer, and temporary variables TW1$ and TW2$ of the task control block between calls to XRDE.

Possible Errors:

53 = File not defined (End of directory)
68 = Not PDOS disk
Disk errors
1.3.71 XRDM - DUMP REGISTERS

Mnemonic: XRDM
Value: $A02A
Module: MPDOSK1
Format: XRDM

Registers: In All

Description: The DUMP REGISTERS primitive formats and outputs all the current register values of the 68000 to the user console along with the program counter, status register, and the supervisor stack.

The registers and status are not affected by this primitive.

See also: XDMP - DUMP MEMORY FROM STACK

Possible Errors: None
1.3.72 XRDN - READ DIRECTORY ENTRY BY NAME

Mnemonic: XRDN  
Value: $A0A8  
Module: MPDOSF  
Format: XRDN  
<status error return>

Registers:  
In D0.B = Disk number  
MWB$ = File name  
Out D1.W = Sector number in memory  
(A2) = Directory entry  
TW2$ = Entry count

Description: READ DIRECTORY ENTRY BY NAME primitive reads directory entries by file name. Register D0.B specifies the disk number. The file name is located in the Monitor Work Buffer (MWB$) in a fixed format. Several other parameters are returned in the monitor TEMP storage of the user task control block. These variables assist in the housekeeping operations on the disk directory.

See also: XFFN - FIX FILE NAME

Possible Errors:

53 = File not defined  
68 = Not PDOS disk  
Disk errors
1.3.73 XRDT - READ DATE

Mnemonic: XRDT
Value: $A05C
Module: MPDOSK3
Format: XRDT

_registers: Out (A1) = 'MN/DY/YR'<null>

_description: READ DATE primitive returns the current system date as a nine character string. The format is 'MN/DY/YR' followed by a null. Address register A1 points to the string in the monitor work buffer.

_see also: XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE
XUTM - UNPACK TIME

_possible_errors: None
1.3.74 XRFA - READ FILE ATTRIBUTES

Mnemonic: XRFA
Value: $A0E0
Module: MPDOSF
Format: XRFA

<status error return>

Registers:
In  (A1) = File name
Out (A2) = Directory entry
D0.L = Disk number
D1.L = File size (in bytes)
D2.L = Level/attributes

Note: Uses multiple directory file search.

Description: READ FILE ATTRIBUTES primitive returns the disk number of where the file was found in data register D0.L. Data register D1.L is returned with the size of the file in bytes. The file directory level is returned in the upper word of register D2.L and the file attributes are returned in register D2.W. The file name is pointed to by address register A1.

File attributes are defined as follows:

$80xx AC - Procedure file
$40xx BN - Binary file
$20xx OB - 68000 object file
$10xx SY - 68000 memory image
$08xx BX - BASIC binary token file
$04xx EX - BASIC ASCII file
$02xx TX - Text file
$01xx DR - System I/O driver
$xx04 C - Contiguous file
$xx02 * - Delete protect
$xx01 ** - Delete and write protect

See also: XCFA - CLOSE FILE W/ATTRIBUTE
XWFA - WRITE FILE ATTRIBUTES
XWFP - WRITE FILE PARAMETERS

Possible Errors:

50 = Invalid file name
53 = File not defined
60 = File space full
Disk errors
1.3.75 XRFP - READ FILE POSITION

Mnemonic: XRFP
Value: $A0FE
Module: MPDOSF
Format: XRFP
<status error return>

Registers: In D1.W = File ID
Out (A3) = File slot address
D2.L = Byte position
D3.L = EOF byte position

Description: READ FILE POSITION primitive returns the current file position, end-of-file position, and file slot address. The open file is selected by the file ID in data register D1.W.

Address register A3 is returned pointing to the open file slot. Data registers D2.L and D3.L are returned with the current file byte position and the end-of-file position respectively.

See also: XPSF - POSITION FILE
XRWF - REWIND FILE

Possible Errors:

52 = File not open
59 = Invalid slot #
Disk errors
1.3.76  XRLF - READ LINE FROM FILE

Mnemonic: XRLF
Value: $A0E2
Module: MPDOSF
Format: XRLF
<status error return>

Registers:
In
  D1.W = File ID
  (A2) = R/W buffer address
Out
  D3.L = # of bytes read
  (On EOF only.)

Description: READ LINE primitive reads one line, delimited by a carriage return [CR], from the file specified by the file ID in register D1. If a [CR] is not encountered after 132 characters, then the line and primitive are terminated. Address register A2 points to the buffer in user memory where the line is to be stored. If the channel buffer has been rolled to disk, the least-used buffer is freed and the buffer is restored to memory. The file slot ID is placed on the top of the last-access queue. If an error occurs during the read operation, the error return is taken with the error number in register D0 and the number of bytes actually read in register D3.

The line read is dependent upon the data content. All line feeds ([LF]) are dropped from the data stream and the [CR] is replaced with a null. The buffer pointer in register A2 may be on any byte boundary. The buffer is not terminated with a null on an error return.

See also:
  XRBF - READ BYTES FROM FILE
  XWBF - WRITE BYTES TO FILE
  XWLF - WRITE LINE TO FILE

Possible Errors:

  52 = File not open
  56 = End of file
  59 = Invalid slot #
  Disk errors
1.3.77 XRNF - RENAME FILE

Mnemonic: XRNF
Value: $A0E4
Module: MPDOSF
Format: XRNF
<status error return>

Registers: In 
(A1) = Old file name
(A2) = New file name

Description: RENAME FILE primitive renames a file in a PDOS disk directory. The old file name is pointed to by address register A1. The new file name is pointed to by address register A2.

The XRNF primitive is used to change the directory level for any file by letting the new file name be a numeric string equivalent to the new directory level. XRNF first attempts a conversion on the second parameter before renaming the file. If the string converts to a number without error, then only the level of the file is changed.

See also: XDFL - DEFINE FILE
XDLF - DELETE FILE

Possible Errors:

50 = Invalid file name
51 = File already defined
Disk errors
1.3.78 XROO - OPEN RANDOM READ ONLY FILE

Mnemonic: XROO
Value: $A0E6
Module: MPDOSF
Format: XROO

<status error return>

Registers: In (A1) = File name
Out D0.W = File attribute
D1.W = File ID

Note: Uses multiple directory file search.

Description: OPEN RANDOM READ ONLY FILE primitive opens a file for random access by assigning
the file to an area of system memory called a file slot, and returning a file ID and file
attribute to the calling program. Thereafter, the file is referenced by the file ID and not
by the file name. This type of file open provides read only access.

The file ID (returned in register R1) is a 2-byte number. The left byte is the disk number
and the right byte is the channel buffer index. The file attribute is returned in register D0.

Since the file cannot be altered, it cannot be extended nor is the LAST UPDATE
parameter changed when it is closed. All data transfers are buffered through a channel
buffer and data movement to and from the disk is by full sectors.

A new file slot is allocated for each XROO call even if the file is already open. The file
slot is allocated beginning with slot 1 to 32.

Possible Errors:

50 = Invalid file name
53 = File not defined
61 = File already open
68 = Not PDOS disk
69 = Not enough file slots

Disk errors
1.3.79 XROP - OPEN RANDOM

Mnemonic: XROP
Value: $A0E8
Module: MPDOSF
Format: XROP

<status error return>

Registers: In (A1) = File name
Out D0.W = File attribute
D1.W = File ID

Note: Uses multiple directory file search.

Description: The OPEN RANDOM FILE primitive opens a file for random access by assigning the file to an area of system memory called a file slot, and returning a file ID and file attribute to the calling program. Thereafter, the file is referenced by the file ID and not by the file name.

The file ID (returned in register D1) is a 2-byte number. The left byte is the disk number and the right byte is the channel buffer index. The file attribute is returned in register D0.

The END-OF-FILE marker on a random file is changed only when the file has been extended. All data transfers are buffered through a channel buffer and data movement to and from the disk is by full sectors.

The file slot is allocated beginning with slot 32 to slot 1. If the file is already open, then the file slot is shared.

Possible Errors:

50 = Invalid file name
53 = File not defined
61 = File already open
68 = Not PDOS disk
69 = Not enough file slots
Disk errors
1.3.80 XRPS - READ PORT STATUS

Mnemonic: XRPS
Value: $A094
Module: MPDOSK2
Format: XRPS
<status error return>

Registers: In  D0.W = Port number

Note: If D0.W=0, then the current port (PRT$(A6)) is used.

Description: The READ PORT STATUS primitive reads the current status of the port specified by data register D0.W. The high order word of data register D1.L is returned zero if no procedure file is open. Otherwise, it is returned with ACI$.

            The low order word is returned with the port flag bits and the status as returned for the port UART routine. The flag bits indicate if eight bit I/O is occurring, if DTR or ^S ^Q protocol is in effect, and other flags.

See also:  XBCP - BAUD CONSOLE PORT
           XSPF - SET PORT FLAG

Possible Errors:

            66 = Invalid port or baud rate
1.3.81 XRSE - READ SECTOR

Mnemonic: XRSE
Value: $A0C2
Module: MPDOSF
Format: XRSE
<status error return>

Registers: In
D0.B = Disk number
D1.W = Sector number
(A2) = Buffer pointer

Description: READ SECTOR primitive calls a system-defined, hardware-dependent program which reads 256 bytes of data into a memory buffer pointed to by address register A2. The disk is selected by data register D0. Register D1 specifies the logical sector number to be read.

See also: XISE - INITIALIZE SECTOR
XRSZ - READ SECTOR ZERO
XWSE - WRITE SECTOR

Possible Errors:

Disk errors
1.3.82 XRSR - READ STATUS REGISTER

Mnemonic: XRSR  
Value: $A042  
Module: MPDOSK1  
Format: XRSR  

Registers: Out D0.W = 68000 status register

Description: READ STATUS REGISTER primitive allows you to read the 68000 status register. Of course, this is equivalent to the 'MOVE.W SR,Dx' instruction on the 68000. However, this instruction is privileged on the 68010 and 68020. Hence, it is advisable to use the XRSR primitive to read the status register to make software upward compatible.

Possible Errors: None
1.3.83 XRST - RESET DISK

Mnemonic: XRST
Value: $A0B4
Module: MPDOSF
Format: XRST

Registers: In
D1.W = -1.... Reset by task
>=0... Reset by disk

Description: RESET DISK primitive closes all open files either by task or disk number. The primitive also clears the assigned input file ID. If register D1 equals -1, then all files associated with the current task are closed. Otherwise, register D1 specifies a disk and all files opened on that disk are closed.

XRST has no error return and as such, closes all files even though errors occur in the close process. This is necessary to allow for recovery from previous errors.

See also: XCFA - CLOSE FILE W/ATTRIBUTE
XCLF - CLOSE FILE

Possible Errors: None
1.3.84 XRSZ - READ SECTOR ZERO

Mnemonic: XRSZ
Value: $A0C4
Module: MPDOSF
Format: XRSZ
<status error return>

Registers: In  D0.B = Disk number
Out  D1.L = 0
(A2) = User buffer pointer (A6)

Description: READ SECTOR ZERO primitive is a system-defined, hardware-dependent program which reads 256 bytes of data into the user memory buffer (usually pointed to by address register A6). The disk is selected by data register D0.W. Register D1.L is cleared and logical sector zero is read.

See also: XISE - INITIALIZE SECTOR
XRSE - READ SECTOR
XWSE - WRITE SECTOR

Possible Errors:

Disk errors
1.3.85 XRTE - RETURN FROM INTERRUPT

Mnemonic: XRTE
Value: $A044
Module: MPDOSK1
Format: XRTE

Registers: In  
SSP = Status register.W
Program counter.L

Description: RETURN FROM INTERRUPT primitive is used to return from an interrupt process routine with a context switch. This allows an immediate rescheduling of the highest priority ready task which may be suspended pending the occurrence of an event set by the interrupt routine.

If the interrupted system is locked when the XRTE primitive is executed, then the reschedule flag (RFLG.(A5)) is cleared and a return from exception instruction (RTE) is executed. When the system clears the task lock, RFLG. is tested and set (TAS) and a rescheduling occurs at that time.

Possible Errors: None
1.3.86  XRTM - READ TIME

Mnemonic:  XRTM
Value:  $A05E
Module:  MPDOSK3
Format:  XRTM

Registers:  Out  (A1) = 'HR:MN:SC'<null>
10(A1).W = Tics/second (B.TPS)
12(A1).L = Tics (TICS.)

Description:  READ TIME primitive returns the current time as a nine-character string. The format is 'HR:MN:SC' followed by a null. Address register A1 points to the string in the monitor work buffer.

See also:  XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRDT - READ DATE
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE
XUTM - UNPACK TIME

Possible Errors:  None
1.3.87 XRTP - READ TIME PARAMETERS

Mnemonic: XRTP
Value: $A034
Module: MPDOSK1
Format: XRTP

Registers:
Out
D0.L = TICS.
D1.L = MONTH/DAY/YEAR/0
D2.L = HOURS/MINUTES/SECONDS/0
D3.L = B.TPS

Description:
READ TIME PARAMETERS primitive returns the current time parameters. Data register D0 returns with the current tic count (TICS.(A5)). Register D1.L returns with the current date and register D2.L the current time. Both are three bytes that are left-justified. Finally, data register D3.L returns with the number of clock tics per second.

See also:
XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRDT - READ DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE
XUTM - UNPACK TIME

Possible Errors: None
1.3.88 XRTS - READ TASK STATUS

Mnemonic: XRTS
Value: $A012
Module: MPDOSK1
Format: XRTS
<status return>

Registers:  
In  D0.W = Task number
Out  D1.L  = 0 - Not executing
       = +N - Time slice
       = -N - (Event #1/Event #2)
       A0.L = TLST entry (IF -D0: A0=TLST.)
       SR = Status of D1.L

Note: If D0.W=-1, then the current task number is returned in D1.L.

Description: READ TASK STATUS primitive returns in register D1 and the status register returns the
time parameter of the task specified by register D0. The time reflects the execution
mode of the task. If D1 returns zero, then the task is not in the task list. If D1 returns a
value greater than zero, then the task is in the run state (executing). If D1 returns a
negative value, then the task is suspended pending event -(D1).

The task number is returned from the CREATE TASK BLOCK (XCTB) primitive. It can
also be obtained by setting data register D0 equal to a minus one. In this case, register
D1.L is returned with the current task number.

See also: XSTP - SET/READ TASK PRIORITY

Possible Errors: None
1.3.89 XRWF - REWIND FILE

Mnemonic: XRWF
Value: $A0EA
Module: MPDOSF
Format: XRWF
<status error return>

Registers: In D1.W = File ID

Description: REWIND FILE primitive positions the file specified by the file ID in register D1, to byte position zero.

See also: XPSF - POSITION FILE
XRFP - READ FILE POSITION

Possible Errors:

52 = File not open
59 = Invalid slot #
70 = Position error
Disk errors
1.3.90 XSEF - SET EVENT FLAG W/SWAP

Mnemonic: XSEF
Value: $A018
Module: MPDOSK1
Format: XSEF
<status return>

 Registers: In D1.B = Event (+=Set, -=Reset)
Out SR = NE....Set
        EQ....Reset

Note: An XSWP is automatically executed after the event is set or reset. Event 128 is local to each task.

If D1.B is positive, then the event is set.
If D1.B is negative, then the event is reset.

Description: SET EVENT FLAG WITH SWAP primitive sets or resets an event flag bit. The event number is specified in data register D1.B and is module 128. If the content of register D1.B is positive, then the event bit is set to 1. Otherwise, the bit is reset to 0. Event 128 can only be set. (It is cleared by the task scheduler.)

The status of the event bit prior to changing the event is returned in the status register. If the event was 0, then the 'EQ' status is returned. Also, an immediate context switch occurs thus scheduling any higher priority task pending on that event.

Events are summarized as follows:

1-63 = Software events
64-80 = Software resetting events
81-95 = Output port events
96-111 = Input port events
112 = 1/5 second event
113 = 1 second event
114 = 10 second event
115 = 20 second event
116 = TTA active
117 = LPT active

See also: XDEV - DELAY SET/RESET EVENT
XSEV - SET EVENT FLAG
XSUI - SUSPEND UNTIL INTERRUPT
XTEF - TEST EVENT FLAG

Possible Errors: None
SECTION 4
VMEPROM SYSTEM CALLS

1.3.91 XSEV - SET EVENT FLAG

Mnemonic: XSEV
Value: $A046
Module: MPDOSK1
Format: XSEV
<status return>

Registers:
In D1.B = Event (+=Set, -=Reset)
Out SR = NE....Set
EQ....Reset

Note: Event 128 is local to each task.

If D1.B is positive, then the event is set.
If D1.B is negative, then the event is reset.

Description: SET EVENT FLAG primitive sets or resets an event flag bit. The event number is specified in data register D1.B and is module 128. If the content of register D1.B is positive, then the event bit is set to 1. Otherwise, the bit is reset to 0. Event 128 can only be set. (It is cleared by the task scheduler.)

The status of the event bit prior to changing the event is returned in the status register. If the event was 0, then the 'EQ' status is returned. A context switch DOES NOT occur with this call making it useful for interrupt routines outside the PDOS system.

Events are summarized as follows:

1-63 = Software events
64-80 = Software resetting events
81-95 = Output port events
96-111 = Input port events
112 = 1/5 second event
113 = 1 second event
114 = 10 second event
115 = 20 second event
116 = TTA active
117 = LPT active

See also: XDEV - DELAY SET/RESET EVENT
XSEV - SET EVENT FLAG
XSUI - SUSPEND UNTIL INTERRUPT
XTEF - TEST EVENT FLAG

Possible Errors: None
1.3.92 XSMP - SEND MESSAGE POINTER

Mnemonic: XSMP
Value: $A002
Module: MPDOSK1
Format: XSMP
<status return>

Registers:
In  D0.B = Message slot number (0..15)
(A1) = Message
Out  SR = EQ....Message sent (Event[64+slot #]=1)
      NE....No message sent

Description: SEND MESSAGE POINTER primitive sends a 32-bit message to the message slot specified by data register D0.B. Address register A1 contains the message. If there is still a message pending, then the primitive immediately returns with status set 'Not Equal' and D0.L equal to 83. Otherwise, the message is taken by PDOS event (64 + message slot number) is set to one indicating a message is ready, and status is returned 'Equal'.

The primitive XSMP is only valid for message slots 0 through 15. (This is because of current event limitations.)

See also: XGMP - GET MESSAGE POINTER
           XGTM - GET TASK MESSAGE
           XKTM - KILL TASK MESSAGE
           XSTM - SEND TASK MESSAGE

Possible Errors:

83 = Message buffer pending
1.3.93 XSOE - SUSPEND ON PHYSICAL EVENT

Mnemonic: XSOE
Value: $A112
Module: MPDOSK1
Format: XSOE

Registers:
In  D1.L  = Event 1 Descriptor.w, Event 0 Descriptor.w
    A0  = Event 0 address (0=no event 0 to suspend on)
    A1  = Event 1 address (0=no event 1 to suspend on)
Out  D0 = -1 if awaken on event 0; 1 if awaken on event 1

Note: This call is the same as XSUI but with physical events.

Description: Allows a task to suspend on one or two events within the system. Tasks that suspend on physical events are listed as suspended on events -1/1. If event 0 is the scheduling event, a -1 is returned; otherwise, a 1 is returned.

The event descriptor is a 16 bit word that defines both the bit number at the specified A0,A1 address and the action to take on the bit. The following bits are defined:

Bit number: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
T  x  x  x  x  x  x  S  x  x  x  B  B  B

T = Should the bit be toggled on scheduling?
1 = Yes (toggle), 0 = No (do not toggle)

S = Suspend on event bit clear or set
1 = Suspend on SET, 0 = Suspend on CLEAR

BBB = The 680 x 0 bit number to use as an event
x = Reserved, should be 0

Since the bit number is specified in the lower three bits of the descriptor, you may use the descriptor with the 680x0 BTST,BCLR,BSET instructions.

See also: XDPE - Delay On Physical Event
           XTLP - Translate Logical To Physical Event
1.3.94  XSOP - OPEN SEQUENTIAL FILE

Mnemonic:  XSOP  
Value:    $A0EC  
Module:  MPDOSF  
Format:  XSOP  
<status error return>

Registers:  In (A1) = File name  
            Out D0.W = File attribute  
                           D1.W = File ID

Note:  Uses multiple directory file search.

Description:  OPEN SEQUENTIAL FILE primitive opens a file for sequential access by assigning the file to an area of system memory called a file slot and returning a file ID and file type to the calling program. Thereafter, the file is referenced by the file ID and not by the file name.

The file ID (returned in register D1) is a 2-byte number. The left byte is the disk number and the right byte is the file slot index. The file attribute is returned in D0.

The END-OF-FILE marker on a sequential file is changed whenever data is written to the file. All data transfers are buffered through a channel buffer; data movement to and from the disk is by full sectors.

The file slots are allocated beginning with slot 32 down to slot 1.

Possible Errors:

50 = Invalid file name  
53 = File not defined  
61 = File already open  
68 = Not PDOS disk  
69 = Not enough file slots  
Disk errors
1.3.95 XSPF - SET PORT FLAG

Mnemonic: XSPF  
Value: $A09A  
Module: MPDOSK2  
Format: XSPF

<status error return>

Registers:  
In  
D0.W = Port number  
D1.B = Port flag (fwpi8dcs)  
Out  
D1.B = Old port flag

Note: If D0.W=0, then the current port (PRT$(A6)) is used.

Description: SET PORT FLAG primitive stores the port flag passed in data register D1.B in the port flag register as specified by register D0.W. If flag bits 'p', 'i', or '8' change, the BIOS baud port routine is called.

See also: XBCP - BAUD CONSOLE PORT  
XRPS - READ PORT STATUS

Possible Errors:

66 = Invalid port or baud rate
1.3.96 XSTM - SEND TASK MESSAGE

Mnemonic: XSTM
Value: $A020
Module: MPDOSK1
Format: XSTM
<status error return>

Registers: In
D0.B = TASK NUMBER
(A1) = MESSAGE

Description: SEND TASK MESSAGE primitive places a 64-character message into a PDOS system message buffer. The message is data-independent and is pointed to by address register A1.

Data register D0 specifies the destination of the message. If register D0 is negative, and there is no input port (phantom port), then the message is sent to the parent task. If there is a port, then the message is sent to itself and will appear at the next command line. Otherwise, register D0 specifies the destination task.

The ability to direct a message to a parent task is very useful in background tasking. An assembler need not know from which task it was spawned and can merely direct any diagnostics to the parent task.

If the destination task number equals -1, the task message is moved to the monitor input buffer and parsed as a command line. This feature is used by the CREATE TASK BLOCK primitive to spawn a new task.

See also: XGMP - GET MESSAGE POINTER
XGTM - GET TASK MESSAGE
XKTM - KILL TASK MESSAGE
XSMP - SEND MESSAGE POINTER
XSTM - SEND TASK MESSAGE

Possible Errors:

78 = Message buffer full
1.3.97 XSTP - SET/READ TASK PRIORITY

Mnemonic: XSTP
Value: $A03C
Module: MPDOSK1
Format: XSTP
<status error return>

Registers: In  D0.B = Task #
            D1.W = Task time/Task priority
Out   D1.B = Task priority (If D1.B was 0)

Note: If D0.B=-1, then select current task. If D1.B=0, then read task priority into D1.B.

Description: SET/READ TASK PRIORITY primitive either sets or reads the task priority selected by data register D0.B. If D1.B is nonzero, then the priority is set. Otherwise, it is read and returned in D1.B. If the upper byte of D1.W is nonzero, then the corresponding task time slice is also set.

See also: XRTS - READ TASK STATUS

Possible Errors:

74 = No such task
1.3.98 XSUI - SUSPEND UNTIL INTERRUPT

Mnemonic: XSUI
Value: $A01C
Module: MPDOSK1
Format: XSUI

Registers:
In D1.W = EV1/EV2
Out D0.L = Event

Description: SUSPEND UNTIL INTERRUPT primitive suspends the user task until one of the events specified in data register D1 occurs. A task can suspend until an event sets (positive event) or until it resets (negative event). A task can suspend pending two different events. This is useful when combined with timeout counters to prevent system lockups. Data register D0.L is returned with the event which caused the task to be scheduled.

A suspended task does not receive any CPU cycles until one of the event conditions is met. When the event bit is set (or reset), the task begins executing at the next instruction after the XSUI call. The task is scheduled during the normal swapping functions of PDOS according to its priority. Register D0.L is used to determined which event scheduled the task.

A suspended task is indicated in the LIST TASK (LT) command under the ‘Event’ parameter. Multiple events are separated by a slash.

Events 64 through 128 toggle when they cause a task to move from the suspended state to the ready state. All others must be reset by the event routine.

If a locked task attempts to suspend itself, the call polls the events until a successful return condition is met.

See also: XDEV - DELAY SET/RESET EVENT
XSEF - SET EVENT FLAG W/SWAP
XSEV - SET EVENT FLAG
XTEF - TEST EVENT FLAG

Possible Errors: None
1.3.99 XSUP - ENTER SUPERVISOR MODE

Mnemonic: XSUP
Value: $A02C
Module: MPDOSK1
Format: XSUP

Registers: None

Description: ENTER SUPERVISOR MODE primitive moves your current task from user mode to supervisor mode. Care should be taken not to crash the system since you would then be executing off the supervisor stack! This primitive enables programs to access I/O addresses and use privileged instructions.

Exit to user mode by executing a ‘ANDI.W #$DFFF,SR’ instruction or the XUSP primitive.

See also: XLSR - LOAD STATUS REGISTER
XUSP - RETURN TO USER MODE

Possible Errors: None
1.3.100 XSWP - SWAP TO NEXT TASK

Mnemonic: XSWP
Value: $A000
Module: MPDOSK1
Format: XSWP

Registers: None

Description: SWAP TO NEXT TASK primitive relinquishes control to the PDOS task scheduler. The next ready task with the highest priority begins executing. (This may be to the same task if there is only one task or the task is the highest priority ready task.)

Possible Errors: None
1.3.101 XSZF - GET DISK SIZE

Mnemonic: XSZF
Value: $A0B6
Module: MPDOSF
Format: XSZF
<status error return>

Registers: In D0.B = Disk number
Out D5.L = Directory size/# of files
D6.L = Allotted/Used
D7.L = Largest/Free

Description: GET DISK SIZE primitive returns disk size parameters in data registers D5 through D7. Data register D5 returns the number of currently defined files in the low word along with the maximum number of files available in the directory in the high word. The low order 16 bits of data register D6 (0-15) returns the total number of sectors used by all files. The high order 16 bits of D6 (16-31) returns the number of sectors allocated for file storage.

The low order 16 bits of data register D7 (0-15) is calculated from the disk sector bit map and reflects the number of sectors available for file allocation. The high order 16 bits of D7 (16-31) is returned with the size of the largest block of contiguous sectors. This is useful in defining large files.

Possible Errors:

68 = Not PDOS disk
Disk errors
1.3.102 XTAB - TAB TO COLUMN

Mnemonic: XTAB  
Value: $A090  
Module: MPDOSK2  
Format: XTAB <column>

Registers: None

Description: TAB TO COLUMN primitive positions the cursor to the column specified by the number following the call. Spaces are output until the column counter is greater than or equal to the parameter.

The first print column is zero. At least one space character will always be output.

Possible Errors: None
**1.3.103 XTEF - TEST EVENT FLAG**

Mnemonic: XTEF  
Value: $A01A  
Module: MPDOSK1  
Format: XTEF  
<status return>

Registers:  
In D1.B = Event number (+=0-127, -=128)  
Out SR = NE....Event set (1)  
EQ....Event clear (0)

Description: TEST EVENT FLAG primitive sets the 68000 status word EQUAL or NOT-EQUAL depending upon the zero or nonzero state of the specified event flag. The flag is not altered by this primitive.

The event number is specified in data register D1 and is module 128. Event 128 is local to each task.

See also: XDEV - DELAY SET/RESET EVENT  
XSEF - SET EVENT FLAG W/SWAP  
XSEV - SET EVENT FLAG  
XSUI - SUSPEND UNTIL INTERRUPT

Possible Errors: None
1.3.104 XTLP - TRANSLATE LOGICAL TO PHYSICAL EVENT

Mnemonic: XTLP
Value: $A110
Module: MPDOSK1
Format: XTLP

Out A0 = Event 0 address (0=no event 0 to suspend on)
A1 = Event 1 address (0=no event 1 to suspend on)
D1 = Event 1 Descriptor.w,Event 0 Descriptor.w

Description: XTLP takes a VMEPROM logical event number and translates the event into a physical event. This call is used when a program needs to suspend on both a logical and a physical event. The logical event is first translated; then the XSOE call is used to suspend it.

Events are summarized as follows:

1 - 63 = Software events
64 - 80 = Software self clearing events
81 - 95 = Output port events
96 -111 = Input port events
112 -115 = Timer events
116 -127 = System control events
128 = Local

The event descriptor is a 16-bit word that defines both the bit number at the specified A0,A1 address and the action to take on the bit. The following bits are defined:

Bit number:  15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
            T  x  x  x  x  x  x  x  S  x  x  x  B  B  B

T = Should the bit be toggled on scheduling?
1 = Yes (toggle), 0 = No (do not toggle)

S = Suspend on event bit clear or set
1 = Suspend on SET, 0 = Suspend on CLEAR

BBB = The 680 x 0 bit number to use as an event
x = Reserved, should be 0
Since the bit number is specified in the lower three bits of the descriptor, you may use the descriptor with the 680 x 0 BTST, BCLR, BSET instructions. You may also use the following physical manipulation calls which are macros for single assembly instructions. They are optimal as long as the values have already been placed in the correct registers. Physical events may need synchronization via the XTAS macro to avoid corruption. The macros are defined in the file PESMACS:SR.

XTST - Test Physical Event (replaces BTST D1, A0))
XSET - Test and Set Physical Event (replaces BSET D1,(A0))
XCLR - Test and Clear Physical Event (replaces BCLR D1,(A0))

Input: D1.W - Event descriptor
       A0 - Event address
Output: None
       Status: EQ - the bit was clear (0)
              NE - the bit was set (1)

The bottom three bits are evaluated as a bit number. The bit at the address is set and the previous value is returned in the Z bit of the status register.

XTAS - Test and Set Physical Event (Bit 7 atomic)

This macro replaces TAS (A0). The seventh bit at the address is set and the previous value is returned in the N bit of the status register.

Input: A0 - Event address
Output: None
Status: EQ - the bit was clear (0)
       NE - the bit was set (1)

See also: XDPE - Delay On Physical Event
          XSOE - Suspend On Physical Event
1.3.105 XUAD - UNPACK ASCII DATE

Mnemonic: XUAD
Value: $A036
Module: MPDOSK3
Format: XUAD

Registers:
In: \( D1.W = (\text{Year} \times 16 + \text{Month}) \times 32 + \text{Day} \)
\[\text{(YYYY YYYM MMMD DDDD)}\]
Out: \( (A1) = \text{DY-MON-YR}<\text{null}> \)
\[\text{(Outputs ??? for invalid months)}\]

Description: UNPACK ASCII DATE primitive returns a pointer in address register A1 to an ASCII date string. Data register D1.W contains the binary date \( (\text{Year} \times 16 + \text{Month}) \times 32 + \text{Day} \). The format of the string is more exact than simple numbers separated by slashed.

Note: XUAD does not check for a valid date and hence, funny looking strings could result. Invalid months are replaced by ‘???.’

See also: XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRDT - READ DATE
XRTM - READ TIME
XUDT - UNPACK DATE
XUTM - UNPACK TIME

Possible Errors: None
1.3.106 XUDT - UNPACK DATE

Mnemonic: XUDT
Value: $A060
Module: MPDOSK3
Format: XUDT

Registers: 
In D1.W = (Year * 16 + Month) * 32 + Day
Out (A1) = 'MN/DY/YR'<null>

Description: UNPACK DATE primitive converts a one-word encoded date into an eight-character string terminated by a null (nine characters total). Data register D1 contains the encoded date and returns with a pointer to the formatted string in address register A1. The output of the FIX TIME & DATE (XFTD) primitive is valid input to this primitive.

See also: XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRDT - READ DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUTM - UNPACK TIME

Possible Errors: None
1.3.107 XULF - UNLOCK FILE

Mnemonic: XULF
Value: $A0EE
Module: MPDOSF
Format: XULF
<status error return>

Registers: In D1.W = File ID

Description: UNLOCK FILE primitive unlocks a locked file for access by any other task. The file is specified by the file ID in data register D1.

See also: XLKF - LOCK FILE

Possible Errors:

52 = File not open
59 = Invalid slot #
Disk errors
1.3.108 XULT - UNLOCK TASK

Mnemonic: XULT
Value: $A016
Module: MPDOSK1
Format: XULT

Registers: None

Description: UNLOCK TASK primitive unlocks the current task by clearing the swap lock variable in system RAM. This allows other tasks to be scheduled and receive CPU time.

See also: XLKT - LOCK TASK

Possible Errors: None
1.3.109 XUSP - RETURN TO USER MODE

Mnemonic: XUSP
Value: $A008
Module: MPDOSK1
Format: XUSP

Registers: None

Description: RETURN TO USER MODE primitive moves your current task from supervisor mode to user mode. Executing an 'ANDI.W #$DFFF,SR' instruction also returns you to user mode, but must be executed in supervisor mode. The XUSP primitive can be executed in either mode.

See also: XLSR - LOAD STATUS REGISTER
XSUP - ENTER SUPERVISOR MODE

Possible errors: None
1.3.110 XUTM - UNPACK TIME

Mnemonic: XUTM
Value: $A062
Module: MPDOSK3
Format: XUTM

Registers: In D1.W = HOUR*256+MINUTE
(HHHH HHHH MMMM MMMM)
Out (A1) = HR:MN<null>

Description: UNPACK TIME primitive converts a one word encoded date into a five character string terminated by a null (six characters total). Data register D1 contains the encoded time and returns a pointer to the formatted string in address register A1. The output of the FIX TIME & DATE (XFTD) primitive is valid input to this primitive.

See also: XFTD - FIX TIME & DATE
XPAD - PACK ASCII DATE
XRDT - READ DATE
XRTM - READ TIME
XUAD - UNPACK ASCII DATE
XUDT - UNPACK DATE

Possible Errors: None
1.3.111 XVEC - SET/READ EXCEPTION VECTOR

Mnemonic: XVEC
Value: $A116
Module: MPDOSK1
Format: XVEC

Registers:
In  D0.W = Exception number (#2-255)
     (A0) = New exception service routine (0=read only)
Out  (A0) = Old service routine

Description: Sets and/or reads the execution vector for the system. The old service routine address is returned so that you may change a routine and then restore the former routine under program control.

See also: XDTV - Define Trap Vectors

Possible Errors: None
### 1.3.112 XWBF - WRITE BYTES TO FILE

**Mnemonic:** XWBF  
**Value:** $A0F0  
**Module:** MPDOSF  
**Format:** XWBF  
<status error return>

**Registers:**  
- In:  
  - D0.L = Byte count - must be positive  
  - D1.W = File ID  
  - (A2) = Buffer address

**Description:** WRITE BYTES TO FILE primitive writes from a memory buffer, pointed to by address register A2, to a disk file specified by the file ID in register D1. Register D0 specifies the number of bytes to be written. If the channel buffer has been rolled to disk, the least-used buffer is freed and the buffer is restored to memory. The file slot ID is placed on the top of the last-access queue.

The write is independent of the data content. The buffer pointer in register A2 may be on any byte boundary. The write operation is not terminated with a null character.

A byte count of zero in register D0 results in no data being written to the file.

If it is necessary for the file to be extended, PDOS first uses sectors already linked to the file. If a null or end link is found, a new sector obtained from the disk sector bit map is linked to the end of the file. If this makes the file non-contiguous, it is retyped as a non-contiguous file.

**See also:**  
- XRBF - READ BYTES FROM FILE  
- XRLF - READ LINE FROM FILE  
- XWLF - WRITE LINE TO FILE

**Possible Errors:**

- 52 = File not open  
- 58 = File delete or write protected  
- 59 = Invalid slot #  
- 60 = File space full  
- Disk errors
1.3.113 XWDT - WRITE DATE

Mnemonic: XWDT  
Value: $A064  
Module: MPDOSK3  
Format: XWDT

Registers:  
D0.B = Month (1-12)  
D1.B = Day (1-31)  
D2.B = Year (0-99)

Description: WRITE DATE primitive sets the system date counters. Register D0 specifies the month and ranges from 1 to 12. Register D1 specifies the day of month and ranges from 1 to 31. Register D2 is the last 2 digits of the year.

No check is made for a valid date.

Possible Errors: None
### 1.3.114 XWFA - WRITE FILE ATTRIBUTES

**Mnemonic:** XWFA  
**Value:** $A0F2  
**Module:** MPDOSF  
**Format:** XWFA  
<status error return>

**Registers:**  
- In (A1) = File name  
- (A2) = ASCII file attributes

**Note:** (A2)=0 clears all attributes.

**Description:** WRITE FILE ATTRIBUTES primitive sets the attributes of the file specified by the file name pointed to by register A1. Register A2 points to an ASCII string containing the new file attributes followed by a null character. The format is:

(A2) = {file type}{protection}

{file type} =  
- AC - Procedure file  
- BN - Binary file  
- OB - 68000 object file  
- SY - 68000 memory image  
- BX - BASIC binary token file  
- EX - BASIC ASCII file  
- TX - Text file  
- DR - System I/O driver

{protection} =  
- * - Delete protect  
- ** - Delete and Write protect

If register A2 points to a zero byte, then all flags, with the exception of the contiguous flag, are cleared.

**See also:**  
- XCFA - CLOSE FILE W/ATTRIBUTE  
- XRFA - READ FILE ATTRIBUTES  
- XWFP - WRITE FILE PARAMETERS

**Possible Errors:**  
- 50 = Invalid file name  
- 53 = File not defined  
- 54 = Invalid file type  
- Disk errors
1.3.115 XWFP - WRITE FILE PARAMETERS

Mnemonic: XWFP
Value: $A0FC
Module: MPDOSF
Format: XWFP
<status error return>

Registers: In (A1) = File name
D0.L = Sector index of EOF/Bytes in last sector
D1.L = Time/Date created
D2.L = Time/Date last accessed
D3.W = OR'd status (less contiguous bit)

Description: WRITE FILE PARAMETERS primitive updates the end-of-file and date parameters of the file specified by the name pointed to by address register A1 in the disk directory.

See also: XCFA - CLOSE FILE W/ATTRIBUTE
XRFA - READ FILE ATTRIBUTES
XWFA - WRITE FILE ATTRIBUTES

Possible Errors:
50 = Invalid file name
53 = File not defined
Disk errors
1.3.116 XWLF - WRITE LINE TO FILE

Mnemonic: XWLF
Value: $A0F4
Module: MPDOSF
Format: XWLF
<status error return>

Registers: In D1.W = File ID
(A2) = Buffer address

Description: WRITE LINE TO FILE primitive writes a line delimited by a null character to the disk file specified by the file ID in register D1. Address register A2 points to the string to be written. If the channel buffer has been rolled to disk, the least-used buffer is freed and the buffer is restored to memory. The file slot ID is placed on the top of the last-access queue.

The write line primitive is independent of the data content, with the exception that a null character terminates the string. The buffer pointer in register A2 may be on any byte boundary. A single write operation continues until a null character is found.

If it is necessary for the file to be extended, PDOS first uses sectors already linked to the file. If a null link is found, a new sector obtained from the disk sector bit map is linked to the end of the file. If this makes the file non-contiguous, it is retyped as a non-contiguous file.

See also: XRBF - READ BYTES FROM FILE
XRLF - READ LINE FROM FILE
XWBF - WRITE BYTES TO FILE

Possible Errors:

52 = File not open
58 = File delete or write protected
59 = Invalid slot #
60 = File space full
Disk errors
1.3.117 XWSE - WRITE SECTOR

Mnemonic: XWSE
Value: $A0C6
Module: MPDOSF
Format: XWSE
<status error return>

Registers: In
D0.B = Disk number
D1.W = Sector number
(A2) = Buffer address

Description: WRITE SECTOR primitive is a system-defined, hardware-dependent program which writes
256 bytes of data from a buffer, pointed to by address register A2, to the logical sector
and disk device specified by data registers D1 and D0 respectively.

See also: CHAPTER 8 BIOS
XISE - INITIALIZE SECTOR
XRSE - READ SECTOR
XRSZ - READ SECTOR ZERO

Possible Errors:

Disk errors
1.3.118 XWTM - WRITE TIME

Mnemonic: XWTM
Value: $A066
Module: MPDOSK3
Format: XWTM

Registers:

In
D0.B = Hours (0-23)
D1.B = Minutes (0-59)
D2.B = Seconds (0-60)

Description:
WRITE TIME primitive sets the system clock time. Register D0 specifies the hour and ranges from 0 to 23. Register D1 specifies the minutes and register D2, the seconds. The latter two range from 0 to 59.

There is no check made for a valid time.

Possible Errors: None
1.3.119 XZFL - ZERO FILE

Mnemonic: XZFL
Value: $A0F6
Module: MPDOSF
Format: XZFL
<status error return>

Registers: In (A1) = File name

Description: ZERO FILE primitive clears a file of any data. If the file is defined, then the end-of-file marker is placed at the beginning of the file. If the file is not defined, it is defined with no data.

See also: XDFL - DEFINE FILE
XDLF - DELETE FILE

Possible errors:

50 = Invalid file name
61 = File already open
68 = Not PDOS disk
Disk errors
APPENDIX
A. VMEbus Board Setup .......................................................... A
   A1. VMEbus Memory .......................................................... A-1
   A2. SYS68K/SIO-1/SIO-2 ................................................. A-2
   A3. SYS68K/ISIO-1/2 .................................................. A-3
   A4. SYS68K/ISCSI-1 Disk Controller .................................. A-4
   A5. Boards providing the Application Command Interface (ACI) ......... A-5
      A5.1 UART Driver ................................................... A-5
      A5.1.1 Onboard EAGLE Module .................................... A-5
      A5.1.2 Offboard EAGLE Modules .................................. A-6
   A6.2 Disk Driver .................................................. A-7

B. S-Record Formats .................................................. B
   B1. S-Record Types .................................................. B-1
   B2. S-Record Example ................................................ B-2

C. System RAM Definitions ........................................... C

D. Task Control Block Definitions ................................... D

E. Interrupt Vector Table of VMEPROM .................................. E

F. Benchmark Source Code ............................................ F

G. Special Locations .................................................. G
   G1. VMEPROM Configuration Table .................................. G-1

H. Minimum Demands for Device Driver Tasks in Order to Run with VMEPROM H
   H1. Device Driver Tasks for Serial Devices .............................. H-1
   H2. Device Driver Tasks for Block Devices .............................. H-4
      H2.1 Floppy Devices .............................................. H-4
      H2.2 SCSI Devices .................................................. H-9
APPENDIX A

A. VMEbus Board Setup

This appendix summarizes the changes to be made to the default setup of additional VMEbus boards so that they are VMEPROM compatible. Appendices A.2 through A.6 are available in EPROM, but are not installed. All drivers may be installed with the INSTALL command. When INSTALL followed by a question mark is entered, the following will appear:¹

? INSTALL ?

THE FOLLOWING UARTS AND DISK DRIVERS ARE ALREADY IN EPROM:

| DISK DRIVER | FORCE ISCSI-1     | ADDR: $FF007300 |
| DISK DRIVER | FORCE IBC/ME      | ADDR: $FF004CC0 |
| DISK DRIVER | FORCE EAGLE/ME    | ADDR: $FF004E30 |
| UART DRIVER | FORCE CPU-39/DUSCC | ADDR: $FF004500 |
| UART DRIVER | FORCE SIO-1/2     | ADDR: $FF004800 |
| UART DRIVER | FORCE ISIO-1/2    | ADDR: $FF004C00 |
| UART DRIVER | FORCE IBC/ME      | ADDR: $FF008410 |
| UART DRIVER | FORCE EAGLE/ME    | ADDR: $FF008610 |
| UART DRIVER | FORCE UNIX MAIL   | ADDR: $FF005100 |
| UART DRIVER | FORCE IBC RAM port | ADDR: $FF00EE7C |

By typing in: INSTALL <file>,<address><cr>, a specific driver may be loaded in the system. The addressed file should be located in EPROM.

A1. VMEbus Memory

In general, every FORCE memory board can be used together with VMEPROM. The base address must be set correctly in order to use the board within the tasking memory of VMEPROM. The board base addresses of any additional memory boards must be set to be contiguous to the on-board memory. It is strongly recommended that only 32 bit memory boards are used because of speed purposes.

¹ Please note that the printed UART and Disk Driver addresses are only examples. They may change according to software versions.
A2. SYS68K/SIO-1/SIO-2

These two serial I/O boards are set to the base address $B00000 by default. VMEPROM expects the first SIO-1/SIO-2 boards at $FCB00000. This is in the standard VME address range (A24, D16, D8) with the address $B00000. The address modifier decoder (AM-Decoder) of the SIO-1/2 boards must be set to:

- Standard Privileged Data Access
- Standard Nonprivileged Data Access

Please refer to the SIO User’s Manual for setup. If a second SIO-1/2 board will be used, the base address must be set to FCB00200. The AM-decoder setup described above must again be used. Please refer to the User’s Manual of your SIO board for the address setup of the second SIO board. Before using the driver for the SIO-1/2 board, the driver must be installed by using the INSTALL command. The following must be entered:

? INSTALL U2,$FF004800

In order to install one of the ports of the SIO boards in VMEPROM, the BP command can be used. The SIO-1/2 boards use the driver type 2. To install the first port of a SIO board with a 9600 baud rate, the following command line can be used:

? BP 4, 9600, 2, $FCB00000

The port can then be used as port number 4. Please note that the hardware configuration must be detected before a port can be installed. This can be done with the CONFIG command. Please refer to the command description in the VMEPROM User’s Manual for a detailed description of the CONFIG and BP commands. The base addresses of all ports of a SIO-1/2 board which must be specified with the BP command is as follows:

<table>
<thead>
<tr>
<th>SIO port #</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (first SIO board)</td>
<td>$FCB00000</td>
</tr>
<tr>
<td>2</td>
<td>$FCB00040</td>
</tr>
<tr>
<td>3</td>
<td>$FCB00080</td>
</tr>
<tr>
<td>4</td>
<td>$FCB000C0</td>
</tr>
<tr>
<td>5</td>
<td>$FCB00100</td>
</tr>
<tr>
<td>6</td>
<td>$FCB00140</td>
</tr>
<tr>
<td>1 (second SIO board)</td>
<td>$FCB00200</td>
</tr>
<tr>
<td>2</td>
<td>$FCB00240</td>
</tr>
<tr>
<td>3</td>
<td>$FCB00280</td>
</tr>
<tr>
<td>4</td>
<td>$FCB002C0</td>
</tr>
<tr>
<td>5</td>
<td>$FCB00300</td>
</tr>
<tr>
<td>6</td>
<td>$FCB00340</td>
</tr>
</tbody>
</table>
VMEPROM supports up to two serial I/O boards. These can be either the SIO-1/2 board, the ISIO-1/2 board, or a mixture of both. Please note that the first board of every type must be set to the first base address. In using one SIO-1 board and one ISIO-1 board, the base address of the boards must to be set to:

\[
\begin{array}{ll}
\text{SIO-1} & \$FCB00000 \\
\text{ISIO-1} & \$FC960000 \\
\end{array}
\]

**A3. SYS68K/ISIO-1/2**

These serial I/O boards are set to the address \$960000 in the standard VME address range by default. VMEPROM awaits this board at this address; no changes need to be made to the default setup. An optional second board may be used. When used, the address must be set to \$980000. Read the SYS68K/ISIO-1/2 User’s Manual for a description of the base address setup. Before using the driver for the ISIO-1/2 board, the driver must be installed by using the INSTALL command. The following must be entered:

\> INSTALL U3,$FF004C00

In order to install one of the ports of an ISIO board in VMEPROM, the BP command can be used. The ISIO-1/2 boards are driver type 3. In order to install the first port of an ISIO board with a 9600 baud rate, the following command line can be used:

\> BP 4, 9600, 3, $FC968000

The port number is four. The hardware configuration must be detected before a port can be installed. This is done with the CONFIG command. Read the command description in the VMEPROM User’s Manual for a description of the CONFIG and BP commands. The base address of all ISIO-1/2 ports, specified by the BP command, is as follows:

<table>
<thead>
<tr>
<th>ISIO port #</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (first ISIO board)</td>
<td>$FC968000</td>
</tr>
<tr>
<td>2</td>
<td>$FC968020</td>
</tr>
<tr>
<td>3</td>
<td>$FC968040</td>
</tr>
<tr>
<td>4</td>
<td>$FC968060</td>
</tr>
<tr>
<td>5</td>
<td>$FC968080</td>
</tr>
<tr>
<td>6</td>
<td>$FC9680A0</td>
</tr>
<tr>
<td>7</td>
<td>$FC9680C0</td>
</tr>
<tr>
<td>8</td>
<td>$FC9680E0</td>
</tr>
<tr>
<td>1 (second ISIO board)</td>
<td>$FC988000</td>
</tr>
<tr>
<td>2</td>
<td>$FC988020</td>
</tr>
<tr>
<td>3</td>
<td>$FC988040</td>
</tr>
<tr>
<td>4</td>
<td>$FC988060</td>
</tr>
<tr>
<td>5</td>
<td>$FC988080</td>
</tr>
<tr>
<td>6</td>
<td>$FC9880A0</td>
</tr>
<tr>
<td>7</td>
<td>$FC9880C0</td>
</tr>
<tr>
<td>8</td>
<td>$FC9880E0</td>
</tr>
</tbody>
</table>
VMEPROM supports two serial I/O boards. These can be the SIO-1/2 or ISIO-1/2 board or mixture of both. The first board of each type must be set to the first base address. When using one SIO-1 and one ISIO-1 board, the base address of the boards must be set to:

- SIO-1 $FCB00000
- ISIO-1 $FC960000

A4. SYS68K/ISCSI-1 Disk Controller

VMEPROM supports up to two floppy disk drives and three Winchester disk drives together with the ISCSI-1 disk controller. The floppy drives must be jumpered to drive select 3 and 4 and can be accessed as disk number 0 and 1 out of VMEPROM. The floppy drives are installed automatically when a ISCSI-1 controller is detected by the CONFIG command. Usable floppy drives must support 80 tracks/side, and must be double sided/double density. The step rate used is 3 ms. The Winchester drives are not installed automatically. The VMEPROM FRMT command must be used for defining the following factors:

- The physical structure of the drive (i.e. number of heads, number of cylinders, drive select number, etc.)
- The bad block of the Winchester drive
- The partitions to be used

If this setup is done once for a particular drive, the data is stored in the first sector of the Winchester and is loaded automatically when the disk controller is installed in VMEPROM. The driver for the ISCSI-1 may be installed by using the INSTALL command. The following must be entered:

? INSTALL W,$FF007300

The default base address of the ISCSI-1 controller is $A00000 in the standard VME address range. This is the address $FCA00000 for the CPU board and no changes have to be made to this setup. The ISCSI-1 driver uses interrupts by default. This cannot be disabled. Please make sure that the interrupt daisy chain is closed so that the controller can work properly.
A5. Boards providing the Application Command Interface (ACI)

Four drivers are included in VMEPROM which manage the communication through the ACI, two disk drivers and two UART drivers. Two of each type are necessary because one controls the onboard EAGLE module(s) and the other controls offboard modules. The driver for offboard modules searches for every board in the system (except itself) and installs as many devices as the driver can handle. To ensure that the driver can find all IBC boards in system, their base addresses must be set according to the following table.

<table>
<thead>
<tr>
<th>Slot #</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$80000000</td>
</tr>
<tr>
<td>2</td>
<td>$84000000</td>
</tr>
<tr>
<td>3</td>
<td>$88000000</td>
</tr>
<tr>
<td>4</td>
<td>$8C000000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>21</td>
<td>$D0000000</td>
</tr>
</tbody>
</table>

A5.1 UART Driver

A5.1.1 Onboard EAGLE Module

To install the UART driver, type:

? INSTALL U7,$FF008610

The UART driver can handle up to 64 serial ports. However, the kernel only allows up to 15 ports. To select a specific port use the BP command. The BP command expects a UART base address. This address is a logical address starting with $1 for the first serial device. The second serial device gets a logical address $2 and so on. For example, when an EAGLE module has 3 serial channels their logical addresses are $1, $2 and $3. To inform the kernel about the second channel, type:

? BP $1905,1,7,$2

Now port 5 is connected to the second serial device on the EAGLE module. The baud rate is set to 9600 baud. The handshake is set to XON/XOFF.
A5.1.2 Offboard EAGLE Modules

To install the UART driver, type:

? INSTALL U8,$FF008410

Now the driver searches for up to 21 boards in the system if they provide the ACI. Every serial device is installed. Additionally, the RAM port of every board with an ACI is installed.

The UART driver can handle up to 64 serial ports. However, the kernel only allows up to 15 ports. To select a specific port use the BP command. The BP command expects a UART base address. This address is a logical address, $1 for the first physical serial device, $2 for the second and so on. The logical address of the RAM port is always the base address of the currently installed board.

The following is an example where a system contains 3 IBC-20 cards.

The first IBC-20 has an EAGLE with 3 serial channels; the IBC-20 base address is $84000000. The second has no serial device; the IBC-20 base address is $B4000000. The third has two EAGLE modules with 6 serial channels; the IBC-20 base address is $B8000000.
After the INSTALL command the driver knows 12 serial channels.

<table>
<thead>
<tr>
<th>Logical Address</th>
<th>UART</th>
</tr>
</thead>
<tbody>
<tr>
<td>$84000000</td>
<td>RAM port of the first IBC-20</td>
</tr>
<tr>
<td>$00000001</td>
<td>The first serial channel of the first IBC</td>
</tr>
<tr>
<td>$00000002</td>
<td>The second serial channel of the first IBC-20</td>
</tr>
<tr>
<td>$00000003</td>
<td>The third serial channel of the first IBC-20</td>
</tr>
<tr>
<td>$B4000000</td>
<td>RAM port of the second IBC-20</td>
</tr>
<tr>
<td>$B8000000</td>
<td>RAM port of the third IBC-20</td>
</tr>
<tr>
<td>$00000004</td>
<td>The first serial channel of the third IBC-20</td>
</tr>
<tr>
<td>$00000005</td>
<td>The second serial channel of the third IBC-20</td>
</tr>
<tr>
<td>$00000006</td>
<td>The third serial channel of the third IBC-20</td>
</tr>
<tr>
<td>$00000007</td>
<td>The fourth serial channel of the third IBC-20</td>
</tr>
<tr>
<td>$00000008</td>
<td>The fifth serial channel of the third IBC-20</td>
</tr>
<tr>
<td>$00000009</td>
<td>The sixth serial channel of the third IBC-20</td>
</tr>
</tbody>
</table>

To inform the kernel about the second channel of the third IBC-20, type:

? BP $1905,1,8,$5

Now port 5 is connected to the second serial device on the EAGLE module. The baud rate is set to 9600 baud. The handshake is set to XON/XOFF.

### A6.2 Disk Driver

VMEPROM supports up to two floppy disk drives and up to four hard disk drives per driver. The first floppy controller and every hard disk controller which is found on the EAGLE Module(s) are installed (up to the limit of four hard disk drives). Hard disks must have a valid partition table on the first physical block. If none is found a default partition table is used. The VMEPROM FRMT command must be used to define the partitions.

Depending on the device driver task the disk access can be cached. Therefore, not every data which is written to the disk from VMEPROM must be written to the hard disk. The FLUSH command of VMEPROM is used to be sure that all data is written to every hard disk.

The driver for onboard EAGLE Modules automatically is installed after power up, while the offboard driver must be installed with the command:

? INSTALL W,$FF004CC0
APPENDIX B

B. S-Record Formats

B1. S-Record Types

Eight types of S-records have been defined to accommodate the needs of the encoding, transportation and decoding functions. VMEPROM supports S0, S1, S2, S3, S7, S8 and S9 records (S7 and S8 on load only).

An S-record format module may contain S-records of the following types:

**S0** The header record for each block of S-records.

**S1** A record containing code/data and the 2-byte address at which the code/data is to reside.

**S2** A record containing code/data and the 3-byte address at which the code/data is to reside.

**S3** A record containing code/data and the 4-byte address at which the code/data is to reside.

**S5** A record containing the number of S1, S2 and S3 records transmitted in a particular block. The count appears in the address field. There is no code/data field. Not supported by VMEPROM.

**S7** A termination record for a block of S3 records. The address field may optionally contain the 4-byte address of the instruction to which control is to be passed. There is no code/data field.

**S8** A termination record for a block of S2 records. The address field may optionally contain the 3-byte address of the instruction to which control is to be passed. There is no code/data field.

**S9** A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed.

Only one termination record is used for each block of S-records. S7 and S8 records are usually used only when control is to be passed to a 3 or 4 byte address. Normally, only one header record is used, although it is possible for multiple header records to occur.
B2. S-Record Example

S214020000000004440002014660000CB241F8044CB1
S214020010203C0000020E428110C1538066FA487AE4
S214020020001021DF0008487A001221DF000C4E750E
S21402003021FC42553200030600821FC41444452C2

<table>
<thead>
<tr>
<th>XX.- Check-sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0200XX 24 bit Address</td>
</tr>
<tr>
<td>14 Byte Count</td>
</tr>
<tr>
<td>S2 Record Type</td>
</tr>
</tbody>
</table>

S9030000FC

<table>
<thead>
<tr>
<th>FC Check-sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 Data</td>
</tr>
<tr>
<td>03 Byte Count</td>
</tr>
<tr>
<td>S9 Record Type</td>
</tr>
</tbody>
</table>
C. System RAM Definitions

/* SYRAM:H -- DEFINITION OF SYRAM BLOCK OF MEMORY
  05-Jan-88 Revised to correspond to PDOS 3.3
  BRIAN C. COOPER, EYRING RESEARCH INSTITUTE, INC.
  Copyright 1985-1988
*/

#define NT  64 /* number of tasks */
#define NM ((NT+3)&0xFC) /* number of task messages */
#define NP  16 /* number of task message pointers */
#define ND ((NT+3)&0xFC) /* number of delay events */
#define NC  8 /* number of active channel buffers */
#define NF  64 /* number of file slots */
#define NU  15 /* number of I/O UART ports */
#define IZ  6 /* input buffer size (2^p2p. */
#define MZ 0x4000000 /* maximum memory size */
#define TZ  64 /* task message size */
#define T2  64 /* task message size */

#define NTB NT
#define NTM NM
#define NFP NP
#define NCB NC
#define NFS NF
#define NEV ND
#define NIE (ND/2)
#define NPS (NU+1)
#define P2P IZ
#define MMB MZ
#define TMB TZ

#define IMK (0xFF>>(8-P2P)) /* input buffer wrap around mask */
#define NCP ((1<<P2P)+2) /* (# characters/port) + 2 */
#define MPZ 2048 /* memory page size */
#define MMZ MZ
#define TMZ TZ
#define NMB (MBZ/8) /* number of map bytes */
#define MBZ (MMZ/MPZ) /* memory bitmap size */
#define NMD ((NT+3)&0xFC) /* number of delay events */
#define NTP NP
#define NPS NF
#define NCB NC
#define NFS NF
#define NEV ND
#define NIE (ND/2)
#define NPS (NU+1)
#define P2P IZ
#define MMB MZ
#define TMB TZ

#define IMK (0xFF>>(8-P2P)) /* input buffer wrap around mask */
#define NCP ((1<<P2P)+2) /* (# characters/port) + 2 */
#define MPZ 2048 /* memory page size */
#define MMZ MZ
#define TMZ TZ
#define NMB (MBZ/8) /* number of map bytes */
#define MBZ (MMZ/MPZ) /* memory bitmap size */
#define NMD ((NT+3)&0xFC) /* number of delay events */
#define NTP NP
#define NPS NF
#define NCB NC
#define NFS NF
#define NEV ND
#define NIE (ND/2)
#define NPS (NU+1)
#define P2P IZ
#define MMB MZ
#define TMB TZ

struct SYRAM{
  char _bios; /* address of bios rom */
  char _mail; /* mail array address */
  unsigned int _rdkn; /* ram disk # */
  unsigned int _rdks; /* ram disk size */
  char _rdka; /* basic present flag */
  char _dflg; /* directory flag */
  int _f681; /* 68000/68010 flag */
  char _sram; /* run module B$SRAM */
  int spare1; /* reserved for expansion */
  int _fcnt; /* fine counter */
  long _tics; /* 32 bit counter */
  unsigned char _smn; /* month */
  unsigned char _sday; /* day */
  unsigned char _syr; /* year */
  unsigned char _shr; /* hours */
  unsigned char _smin; /* minutes */
  unsigned char _ssec; /* seconds */
  char _patb[16]; /* input port allocation table */
  char _brkf[16]; /* input break flags */
  char _f8bt[16]; /* port flag bits */
  char _utyp[16]; /* port uart type */
  char _urat[16]; /* port rate table */
C. System RAM Definitions (cont’d)

```c
/*078*/ char _evtb[10]; /* 0-79 event table */
/*082*/ char _evto[2]; /* 80-95 output events */
/*084*/ char _evti[2]; /* 96-111 input events */
/*086*/ char _evts[2]; /* 112-127 system events */
/*088*/ char _evt128[16]; /* task 128 events */
/*090*/ long _evtm[4]; /* events 112-115 timers */
/*098*/ char _tc; /* clock adjust constant */
/*100*/ char *tclp; /* task list pointer */
/*104*/ int _tsmp; /* user tcb ptr */
/*108*/ int _tsmp; /* supervisor interrupt mask */
/*112*/ int _utim; /* user task time */
/*116*/ char _tspsn; /* spawn task no. (** must be even **)*/
/*119*/ char _utim; /* current task number */
/*120*/ char _spare2; /* reserved */
/*121*/ char _tuxp; /* task queue offset flag/no */
/*122*/ char _tlock[2]; /* task lock/reschedule flags */
/*126*/ char _el122; /* batch task # */
/*128*/ char _el123; /* spooler task # */
/*129*/ char _el124; /* */
/*130*/ char _el125; /* */
/*131*/ long _ckms; /* system checksum */
/*133*/ int _pntd; /* pnet node # */
/*135*/ char _bsgr[6]; /* bus error vector */
/*139*/ char _mler[6]; /* illegal vector */
/*141*/ char _mcmn; /* control C count */
/*143*/ char _w_rnd; /* window id’s */
/*145*/ char _w_rnd; /* window addresses */
/*147*/ char _w_rnd; /* output stream */
/*149*/ char _w_rnd; /* output stream */
/*151*/ char _w_rnd; /* i/o redirect */
/*153*/ char _w_rnd; /* file expand count */
/*155*/ char _w_rnd; /* processor ident byte */
/*157*/ long _w_bgn; /* abs addr of $K$BEGN table */
/*160*/ int _w_rnd[14]; /* port row/col 1..15 */
/*162*/ int _w_rnd[15]; /* output port pointers 1..15 */
/*164*/ char _w_rnd[16]; /* uart base addresses 1..15 */
/*166*/ long _w_rnd; /* memory map bias */
/* */
/* the following change with different configurations:/
*/
/* configuration for VMEPROM is defined to:*/
*/
/* NT = 64, NF = 64, MZ = $400000*/
/**/
/* NOTE: the offset on top of each line is calculated only for this */
/* configuration */
/* */
/*019C*/ char _maps[NMB]; /* system memory bitmap */
/*119C*/ char _cmpt[NPS-1]*NCP/* character input buffers */
/*157A*/ char _cmpt[NPS-1]*NCP/* character output buffers */
/*1958*/ char _rdtb[16]; /* redirect table */
/*1968*/ int _pm[NPS]; /* task queue */
/*198A*/ char _ptst[NTP]+1; /* task list */
/*1A0A*/ char _ptse[NTB*32]; /* task schedule event table */
/*256A*/ long _tmsf[NTM]; /* task message buffers */
/*266A*/ char _tmsa[NTB*6]; /* task message pointers */
/*268A*/ char _tmsi[2+8+NIE*10]; /* delay event insert queue */
/*289A*/ char _tdev[2+NEV*10]; /* delay events */
/*361A*/ int _bcnt[32]; /* basic screen command table*/
/*3856*/ int _xchb[NCB]; /* channel buffer queue */
/*3866*/ char _xchb[NCB]+BPS/* channel buffers */
/*4366*/ char _xfs1[NFS*FSS]; /* file slots */
/*4CEA*/ char _xlat[12]; /* level 2 lock (file prims, evnt 120)*/
/*4CEA*/ char _xlat[13]; /* level 3 lock (disk prims, evnt 121)*/
/*4CEA*/ long _drvl; /* driver link list entry point */
/*4CEA*/ long _utll; /* utility link list entry point */
/*4CPF*/ int _rdkl[NRD*4 + 1]; /* RAM disk list */
*/
```
APPENDIX D

D. Task Control Block Definitions

```c
#define MAXARG 10 /* max argument count of the cmd line */
#define MAXBP 10 /* max 10 breakpoints */
#define MAXNAME 5 /* max 5 names in name buffer */
#define TMAX 64 /* max number of tasks */
#define ARGLEN 20 /* maximum argument length */

/* special system flags for VMEPROM */
#define SOMEREG 0x0001 /* display only PC,A7,A6,A5 */
#define T_DISP 0x0002 /* no register display during Trace(TC>1) */
#define T_SUB 0x0004 /* trace over subroutine set */
#define T_ASUB 0x0008 /* trace over subroutine active */
#define T_RANG 0x0010 /* trace over range set */
#define REG_INI 0x0020 /* no register initialization if set */
#define RE_DIR 0x0040 /* output redirection into file and */

/* the registers are stored in the following order: */
#define VBR 0
#define SFC 1
#define DFC 2
#define CACR 4
#define PC 5
#define SR 6
#define USTACK 7
#define SSTACK 8
#define MSTACK 9
#define D0 10 /* 10-17 = D0-D7 */
#define A0 18 /* 18-24 = A0-A6 */

#define N_REGS 25
#define BYTE unsigned char
#define WORD unsigned int
#define LWORD unsigned long

struct TCB{
    /*000*/ char _ubuf[256]; /* 256 byte user buffer */
    /*100*/ char _clb[80]; /* 80 byte monitor command line buffer */
    /*150*/ char _mwb[32]; /* 32 byte monitor parameter buffer */
    /*170*/ char _mpb[60]; /* monitor parameter buffer */
    /*1AC*/ char _cob[8]; /* character out buffer */
    /*1B4*/ char _swb[508]; /* system work buffer/task pdos stack */
    /*3B0*/ char _tsp; /* task stack pointer */
    /*3B4*/ char _kil; /* kill self pointer */
    /*3B8*/ long _sfp; /* RESERVED FOR INTERNAL PDOS USE */
    /*3BC*/ char _svt; /* save flag -- 8881 support (x881) */
    /*3BG*/ char _lif; /* RESERVED FOR INTERNAL PDOS USE */
    /*3BE*/ long _trp[16]; /* user TRAP vectors */
    /*3F0*/ long _zdv; /* zero divide trap */
    /*402*/ long _chk; /* CHECK instruction trap */
    /*406*/ long _trv; /* TRAPV Instruction trap */
```
D. Task Control Block Definitions (cont’d)

/*40A*/ long _trc; /* trace vector */
/*40E*/ long _fpa[2]; /* floating point accumulator */
/*41A*/ char _cclp; /* command line pointer */
/*41B*/ char _bum; /* beginning of user memory */
/*422*/ char _eum; /* end user memory */
/*426*/ char _ead; /* entry address */
/*42A*/ char _imp; /* internal memory pointer */
/*430*/ int _ac12; /* assigned input file ID's */
/*432*/ int _len; /* last error number */
/*434*/ int _sf1; /* spool file id */
/*436*/ BYTE _flag; /* task flags (bit 8=command line echo)*/
/*437*/ BYTE _slv; /* directory level */
/*438*/ char _fep; /* file expansion count */
/*439*/ char _sparer; /* reserved for future use */
/*43A*/ char _csc[2]; /* clear screen characters */
/*43C*/ char _pca[2]; /* position cursor characters */
/*43E*/ char _sdk; /* system disk */
/*441*/ BYTE _ext; /* XEXT address */
/*442*/ char _err; /* XERR address */
/*44A*/ char _cmd; /* command line delimiter */
/*44B*/ BYTE _tid; /* task id */
/*44C*/ char _ecf; /* echo flag */
/*44D*/ char _cnt; /* output column counter */
/*44F*/ char _smf; /* memory modified flag */
/*450*/ char _spu; /* spooling unit mask */
/*451*/ BYTE _unt; /* output unit mask */
/*452*/ char _ulp; /* unit 1 port # */
/*453*/ char _ul2; /* unit 2 port # */
/*454*/ char _ul4; /* unit 4 port # */
/*455*/ char _ul8; /* unit 8 port # */
/*456*/ char _sparer2[26]; /* reserved for system use */

/*******************************************************************************/
/* VMEPROM variable area */
/*******************************************************************************/
/*470*/ char linebuf[82]; /* command line buffer */
/*4C2*/ char alinebuf[82]; /* alternate line buffer */
/*514*/ char cmdline[82]; /* alternate cmdline for XGNP */
/*566*/ int allargs, gotargs; /* argc save and count for XGNP */
/*56A*/ int argc; /* argument counter */
/*56C*/ char *argv[MAXARG]; /* pointer to arguments of the cmd line*/
/*589*/ char *odir, *idir; /* I/O redirection args from cmd line */
/*59C*/ int ipost, oport; /* I/O port assignments */
/*5A0*/ char *ladr; /* holds pointer to line in_mwb */
/*5A4*/ LWORD offset; /* base memory pointer */
/*5A8*/ int bpcnt; /* num of defined breakpoints */
/*5D2*/ LWORD bpadr[MAXBP]; /* breakpoint address */
/*5D6*/ WORD bpinst[MAXBP]; /* breakpoint instruction */
/*5E6*/ char bpcmd[MAXBP][11]; /* breakpoint command */
D. Task Control Block Definitions (cont'd)

```c
/*654*/ WORD bpocc[MAXBP]; /* # of times the breakpoint should be*/
/* skipped */
/*668*/ WORD bpcocc[MAXBP]; /* # of times the breakpoint is already*/
/* skipped */
/*67C*/ LWORD bptadr; /* temp. breakpoint address */
/*680*/ WORD bptinst; /* temp. breakpoint instruction */
/*682*/ LWORD bptocc; /* # of times the temp. breakpoint should*/
/* be skipped */
/*684*/ WORD bptcocc; /* # of times the temp. breakpoint is */
/* already skipped */
/*686*/ char bptcmd[11]; /* temp. breakpoint command */
/*691*/ char outflag; /* output messages (yes=1,no=0) */
/*692*/ char namebn[MAXNAME][8]; /* Name buffer, name */
/*698*/ char namebd[MAXNAME][40]; /* Name buffer, data */
/*702*/ WORD errcnt; /* error counter for test .. */
/*704*/ LWORD times, timee; /* start/end time */
/*70C*/ LWORD pregs[N_REGS]; /* storage area of processor regs */
/*710*/ WORD tflag; /* trace active flag */
/*712*/ WORD tcount; /* trace count */
/*714*/ WORD tacount; /* active trace count */
/*716*/ WORD bpact; /* break point active flag */
/*718*/ LWORD savesp; /* save VMEMPROM stack during GO/T etc */
/*71C*/ char VMEMSP[202]; /* Master stack, handle w/ care */
/*720*/ char VMESSP[802]; /* supervisor stack, handle w/ care */
/*724*/ char VMESPUSP[802]; /* vmeprom internal user stack */
/*728*/ LWORD f_freg[3*8]; /* floating point data regs */
/*72A*/ LWORD f_fpsr; /* FPCR reg */
/*72C*/ LWORD f_fpiar; /* FPIAR reg */
/*72E*/ BYTE f_savc[0x3c]; /* FSAVE for null and idle */
/*732*/ BYTE cleso[2]; /* clear to end of screen parameter */
/*734*/ BYTE cleso[2]; /* clear to end of line parameters */
/*736*/ char u_prompt[10]; /* user defined prompt sign */
/*738*/ long c_save; /* save Cache control register */
/*73A*/ BYTE nokill; /* kill task with no input port */
/*73C*/ BYTE u_mask; /* unit mask for echo */
/*73E*/ LWORD sysflg; /* system flags used by VMEMPROM */
/* bit 0: display registers short form*/
/* bit 1: trace without reg. display */
/* bit 2: trace over subroutine */
/* bit 3: trace over subroutine active*/
/* bit 4: trace over range */
/* bit 5: no register initialization */
/* bit 6: output redirection into file*/
/* and console at the same time*/
/*740*/ LWORD t_range[2]; /* start/stop PC for trace over range */
/*742*/ LWORD ex_regs; /* pointer to area for saved regs */
/*744*/ char _tbe[0]; /* task beginning */
```

This page was intentionally left blank
### E. Interrupt Vector Table of VMEPROM

<table>
<thead>
<tr>
<th>Vector Number/s</th>
<th>Vector HEX</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>Reset: Initial Interrupt Stack Pointer</td>
</tr>
<tr>
<td>1</td>
<td>004</td>
<td>Reset: Initial Program Counter</td>
</tr>
<tr>
<td>2</td>
<td>008</td>
<td>Bus Error</td>
</tr>
<tr>
<td>3</td>
<td>00C</td>
<td>Address Error</td>
</tr>
<tr>
<td>4</td>
<td>010</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>5</td>
<td>014</td>
<td>Zero Divide</td>
</tr>
<tr>
<td>6</td>
<td>018</td>
<td>CHK, CHK2 Instruction</td>
</tr>
<tr>
<td>7</td>
<td>01C</td>
<td>FTRAPcc, TRAPcc, TRAPV Instructions</td>
</tr>
<tr>
<td>8</td>
<td>020</td>
<td>Privilege Violation</td>
</tr>
<tr>
<td>9</td>
<td>024</td>
<td>Trace</td>
</tr>
<tr>
<td>10</td>
<td>028</td>
<td>VMEPROM System Calls</td>
</tr>
<tr>
<td>11</td>
<td>02C</td>
<td>Coprocessor Instructions</td>
</tr>
<tr>
<td>12</td>
<td>030</td>
<td>(Unassigned, Reserved)</td>
</tr>
<tr>
<td>13</td>
<td>034</td>
<td>Not used</td>
</tr>
<tr>
<td>14</td>
<td>038</td>
<td>Format Error</td>
</tr>
<tr>
<td>15</td>
<td>03C</td>
<td>Uninitialized Interrupt</td>
</tr>
<tr>
<td>16</td>
<td>040</td>
<td></td>
</tr>
<tr>
<td>THROUGH 23</td>
<td>05C</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>060</td>
<td>Spurious Interrupt</td>
</tr>
<tr>
<td>25</td>
<td>064</td>
<td>AV1</td>
</tr>
<tr>
<td>26</td>
<td>068</td>
<td>AV2</td>
</tr>
<tr>
<td>27</td>
<td>06C</td>
<td>AV3</td>
</tr>
<tr>
<td>28</td>
<td>070</td>
<td>AV4</td>
</tr>
<tr>
<td>29</td>
<td>074</td>
<td>AV5</td>
</tr>
<tr>
<td>30</td>
<td>078</td>
<td>AV6</td>
</tr>
<tr>
<td>31</td>
<td>07C</td>
<td>AV7</td>
</tr>
<tr>
<td>32</td>
<td>080</td>
<td></td>
</tr>
<tr>
<td>THROUGH 47</td>
<td>0BC</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0CD</td>
<td>FPCP Branch or Set on Unordered Condition</td>
</tr>
<tr>
<td>49</td>
<td>0C4</td>
<td>FPCP Inexact Result</td>
</tr>
<tr>
<td>50</td>
<td>0C8</td>
<td>FPCP Divide by Zero</td>
</tr>
<tr>
<td>51</td>
<td>0CC</td>
<td>FPCP Underflow</td>
</tr>
<tr>
<td>52</td>
<td>0D0</td>
<td>FPCP Operand Error</td>
</tr>
<tr>
<td>53</td>
<td>0D4</td>
<td>FPCP Overflow</td>
</tr>
<tr>
<td>54</td>
<td>0D8</td>
<td>FPCP Signaling NaN</td>
</tr>
<tr>
<td>55</td>
<td>0DC</td>
<td>FPCP Unimplemented Data Type</td>
</tr>
<tr>
<td>56</td>
<td>0E0</td>
<td>PMMU Configuration</td>
</tr>
<tr>
<td>57</td>
<td>0E4</td>
<td>PMMU Illegal Operation</td>
</tr>
<tr>
<td>58</td>
<td>0E8</td>
<td>PMMU Access Level Violation</td>
</tr>
<tr>
<td>59</td>
<td>0EC</td>
<td></td>
</tr>
<tr>
<td>THROUGH 63</td>
<td>0FC</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>THROUGH 159</td>
<td>27C</td>
<td></td>
</tr>
</tbody>
</table>

The Interrupt Vector Table of VMEPROM is continued on the next page.
## Interrupt Vector Table of VMEPROM (Continued)

<table>
<thead>
<tr>
<th>Vector Number/s</th>
<th>Vector HEX</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>280</td>
<td>Disk Interrupt Vector</td>
</tr>
<tr>
<td>161</td>
<td>284</td>
<td>Reserved</td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>ISIO-1/2 Interrupt Vector</td>
</tr>
<tr>
<td>168</td>
<td>2A0</td>
<td>J</td>
</tr>
<tr>
<td>169</td>
<td>2A4</td>
<td>J</td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>ISIO-1/2 Interrupt Vectors</td>
</tr>
<tr>
<td>181</td>
<td>2D4</td>
<td>J</td>
</tr>
<tr>
<td>182</td>
<td>2D8</td>
<td>Reserved</td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>191</td>
<td>2FC</td>
<td>J</td>
</tr>
<tr>
<td>192</td>
<td>300</td>
<td>Mailbox 0 (Used by the ACI)</td>
</tr>
<tr>
<td>193</td>
<td>304</td>
<td>Mailbox 1</td>
</tr>
<tr>
<td>194</td>
<td>308</td>
<td>Mailbox 2</td>
</tr>
<tr>
<td>195</td>
<td>30C</td>
<td>Mailbox 3 (Reserved)</td>
</tr>
<tr>
<td>196</td>
<td>310</td>
<td>Mailbox 4 (Used from the EAGLE UART driver)</td>
</tr>
<tr>
<td>197</td>
<td>314</td>
<td>Mailbox 5 (Used from the IBC UART driver)</td>
</tr>
<tr>
<td>198</td>
<td>318</td>
<td>Mailbox 6 (Used from the EAGLE disk driver)</td>
</tr>
<tr>
<td>199</td>
<td>31C</td>
<td>Mailbox 7 (Used from the IBC disk driver)</td>
</tr>
<tr>
<td>200</td>
<td>320</td>
<td>Reserved</td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>223</td>
<td>37C</td>
<td>J</td>
</tr>
<tr>
<td>224</td>
<td>380</td>
<td>Timer</td>
</tr>
<tr>
<td>225</td>
<td>384</td>
<td>Reserved</td>
</tr>
<tr>
<td>226</td>
<td>388</td>
<td>Reserved</td>
</tr>
<tr>
<td>227</td>
<td>38C</td>
<td>Reserved</td>
</tr>
<tr>
<td>228</td>
<td>390</td>
<td>FM81 Refused</td>
</tr>
<tr>
<td>229</td>
<td>394</td>
<td>FM80 Refused</td>
</tr>
<tr>
<td>230</td>
<td>398</td>
<td>FM81 Message</td>
</tr>
<tr>
<td>231</td>
<td>39C</td>
<td>FM80 Message</td>
</tr>
<tr>
<td>232</td>
<td>3A0</td>
<td>ABORT</td>
</tr>
<tr>
<td>233</td>
<td>3A4</td>
<td>ACFAIL*</td>
</tr>
<tr>
<td>234</td>
<td>3A8</td>
<td>SYSFAIL*</td>
</tr>
<tr>
<td>235</td>
<td>3AC</td>
<td>DMA Error</td>
</tr>
<tr>
<td>236</td>
<td>3B0</td>
<td>DMA Normal</td>
</tr>
<tr>
<td>237</td>
<td>3B4</td>
<td>PARITY Error</td>
</tr>
<tr>
<td>238</td>
<td>3B8</td>
<td>Reserved</td>
</tr>
<tr>
<td>239</td>
<td>3BC</td>
<td>Reserved</td>
</tr>
<tr>
<td>240</td>
<td>3C0</td>
<td>LOCAL1</td>
</tr>
<tr>
<td>241</td>
<td>3C4</td>
<td>LOCAL2</td>
</tr>
<tr>
<td>242</td>
<td>3C8</td>
<td>LOCAL3</td>
</tr>
<tr>
<td>243</td>
<td>3CC</td>
<td>LOCAL4</td>
</tr>
<tr>
<td>244</td>
<td>3D0</td>
<td>LOCAL5</td>
</tr>
<tr>
<td>245</td>
<td>3D4</td>
<td>LOCAL6</td>
</tr>
<tr>
<td>246</td>
<td>3D8</td>
<td>LOCAL7</td>
</tr>
<tr>
<td>247</td>
<td>3DC</td>
<td>LOCAL8</td>
</tr>
<tr>
<td>248</td>
<td>3E0</td>
<td>Reserved</td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>254</td>
<td>3F4</td>
<td>J</td>
</tr>
<tr>
<td>255</td>
<td>3FC</td>
<td>Empty Interrupt</td>
</tr>
</tbody>
</table>
F. Benchmark Source Code

***********************************************************************
** Module name: Assembler benchmarks Version: 1.0 **
** date started: 20-Apr-87 M.S. last update: 23-Apr-87 M.S. **
** Copyright (c) 1986/87 FORCE Computers GmbH Munich **
***********************************************************************

section 0
opt alt,P=68020,P=68881
xdef .benchex
xdef .BEN1BEG,.BEN1END
xdef .BEN2BEG,.BEN2END
xdef .BEN3BEG,.BEN3END
xdef .BEN4BEG,.BEN4END
xdef .BEN5BEG,.BEN5END
xdef .BEN6BEG,.BEN6END
xdef .BEN7BEG,.BEN7END
xdef .BEN8BEG,.BEN8END
xdef .BEN9BEG,.BEN9END
xdef .BEN10BEG,.BEN10END
xdef .BEN11BEG,.BEN11END
xdef .BEN12BEG,.BEN12END
xdef .BEN13BEG,.BEN13END
xdef .BEN14BEG,.BEN14END

page

* benchmark execution: benchex(address)
*
movem.l d1-a6,-(a7)
movem.l 15*4(a7),a0
jsr (a0)
movem.l (a7)+,d1-a6
rts

* BENCH #1: DECREMENT LONG WORD IN MEMORY 10.000.000 TIMES
*
LEA.L @010(PC),A0
MOVE.L #10000000,(A0)
@020 SUBQ.L #1,(A0)
BNE.S @020
RTS
@010 DS.L 1

* BENCH #2: PSEUDO DMA 1K BYTES 50.000 TIMES
*
MOVE.L #50000,D2 ; DO 50000 TRANSFERS
@001 MOVE.W #$FF,D3 ; EACH IS 1K BYTES
LEA.L @010(PC),A1 ; A1 POINTS TO SOURCE AND DESTINATION
@002 MOVE.L (A1),(A1)+
DBRA D3,@002
SUBQ.L #1,D2
BNE.S @001
RTS
NOP
@010 NOP
PAGE
* BENCH #3: SUBSTRING CHARACTER SEARCH 100,000 TIMES TAKEN FROM EDN 08/08/85
* 
* MOVE.L #100000,D4  
@002 MOVE.L #15,D0  
MOVE.L #120,D1  
LEA.L EDN1DAT(PC),A1  
LEA.L EDN1DAT1(PC),A0  
BSR.S EDN1  
SUBQ.L #1,D4  
BNE.S @002  
RTS

****** BEGIN EDN BENCH #1 ******
EDN1 MOVEM.L D3/D4/A2/A3,-(A7)  
SUB.W D0,D1  
MOVE.W D1,D2  
SUBQ.W #2,D0  
MOVE.B (A0)+,D3  
@010 CMP.B (A1)+,D3  
@012 DBEQ D1,@010  
BNE.S @090  
MOVE.L A0,A2  
MOVEM.L A1,A3  
MOVE.W D0,D4  
BMI.S @030  
@020 CMP.B (A2)+,(A3)+  
DBNE D1,@020  
BNE.S @012  
@030 SUB.W D1,D2  
@032 MOVEM.L (A7)+,D3/D4/A2/A3  
RTS  
@090 MOVEQ.L #-1,D2  
BRA.S @032

****** END EDN BENCH #1 ******
EDN1DAT DC.B ’00000000000000000000000000000000’  
DC.B ’00000000000000000000000000000000’  
EDN1DAT1 DC.B ’HERE IS A MATCH000000000000000’  
PAGE

* BENCH #4: BIT TEST/SET/RESET 100,000 TIMES TAKEN FROM EDN 08/08/85
* 
MOVE.L #100000,D4  
LEA.L EDN2DAT(PC),A0  
@010 MOVEQ.L #1,D0  ; TEST  
MOVEQ.L #10,D1  
BSR.S EDN2  
MOVEQ.L #1,D0  
MOVEQ.L #11,D1  
BSR.S EDN2  
MOVEQ.L #1,D0  
MOVE.W #123,D1  
BSR.S EDN2  
MOVEQ.L #2,D0  ; SET  
MOVEQ.L #10,D1  
BSR.S EDN2
MOVEQ.L #1,D0
MOVEQ.L #11,D1
BSR.S EDN2
MOVEQ.L #1,D0
MOVE.W #123,D1
BSR.S EDN2
MOVEQ.L #3,D0 ; RESET
MOVEQ.L #10,D1
BSR.S EDN2
MOVEQ.L #1,D0
MOVEQ.L #11,D1
BSR.S EDN2
MOVEQ.L #1,D0
MOVE.W #123,D1
BSR.S EDN2
SUBQ.L #1,D4
BNE.S @010
RTS

* EDN2 SUB.W #2,D0
BEQ.S @020
SUBQ.W #1,D0
BEQ.S @030

@010 *
BFTST (A0){D1:1}
DC.W $E8D0
DC.W $0841
SNE D2
RTS

@020 *
BFSET (A0){D1:1}
DC.W $EEED0
DC.W $0841
SNE D2
RTS

@030 *
BFTST (A0){D1:1}
DC.W $E8D0
DC.W $0841
SNE D2
RTS

EDN2DAT DC.L 0,0,0,0
PAGE *
*BENCH #5: BIT MATRIX TRANSPOSITION 100.000 TIMES
*TAKEN FROM EDN 08/08/85
*
MOVE.L #100000,D4
LEA.L EDN3DAT(PC),A0
@002 MOVE.L #7,D0
MOVEQ.L #0,D1
BSR.S EDN3
SUBQ.L #1,D4
BNE.S @002
RTS
EDN3  MOVEM.L D1-D7,-(A7)
MOVE.L D1,D2
MOVE.W D0,D7
SUBQ.W #2,D7
@010  ADDQ.L #1,D1
MOVE.L D1,D3
ADD.L D0,D2
MOVE.L D2,D4
@020  BFEXTU (A0){D3:1},D5
BFEXTU (A0){D4:1},D6
BFINS D5,(A0){D4:1}
BFINS D6,(A0){D3:1}
ADD.L D0,D3
ADDQ.L #1,D4
CMP.L D3,D4
BNE.S @020
DBRA D7,@010
MOVEM.L (A7)+,D1-D7
RTS
EDN3DAT DC.B %01001001
DC.B %01011100
DC.B %10001110
DC.B %10100101
DC.B %00000001
DC.B %01110010
DC.B %10000000
EVEN
PAGE

* BENCH #6: CACHE TEST - 128KB PROGRAM IS EXECUTED 1000 TIMES
* CAUTION: THIS BENCHMARK NEEDS 128 KBYTE MEMORY
*
LEA.L @010(PC),A2
MOVE.L #$203A0000,D1 ; OPCODE FOR MOVE.L ($0,PC),D0
MOVE.L #$20000/4,D2 ; LENGTH IS 128 KBYTE
@004  MOVE.L D1,(A2)+ ; LOAD OPCODE TO MEMORY
SUBQ.L #1,D2
BNE.S @004
MOVE.W #$4E75,(A2) ; APPEND RTS
* PROGRAM IS NOW LOADED -- START 1000 TIMES
MOVE.L #1000,D3
@008  BSR.S @010
SUBQ.L #1,D3
BNE.S @008
RTS
*
@010  DC.L 0  ; PROGRAM WILL START HERE
PAGE
*
* BENCH #7: FLOATING POINT 1.000.000 ADDITIONS
*
MOVE.L #1000000,D5
FMOVE.L #0,FP0
FMOVE.L #1,FP1
@010  FADD.X FP0,FP1
SUBQ.L #1,D5
BNE.S @010
RTS
(cont’d)

* BENCH #8: FLOATING POINT 1.000.000 SINUS

MOVE.L #1000000,D5
FMOVE.L #1,FP1
@010 FSIN.X FP1
SUBQ.L #1,D5
BNE.S @010
RTS

* BENCH #9: FLOATING POINT 1.000.000 MULTIPLICATIONS

MOVE.L #1000000,D5
FMOVE.L #1,FP0
FMOVE.L #1,FP1
@010 FMUL.X FP0,FP1
SUBQ.L #1,D5
BNE.S @010
RTS

* PDOS BENCHMARK #1: CONTEXT SWITCHES

MOVE.L #1000000,D6
@000 XSWP ;CONTEXT SWITCH
SUBQ.L #1,D6 ;DONE?
BGT.S @000 ;N
RTS

* PDOS BENCHMARK #2: EVENT SET

MOVEQ.L #32,D1 ;SELECT EVENT 32
MOVE.L #1000000,D6
@000 XSEV ;SET EVENT
SUBQ.L #1,D6 ;DONE?
BGT.S @000 ;N
RTS

* PDOS BENCHMARK #3: CHANGE TASK PRIORITY

MOVEQ.L #-1,D0 ;SELECT CURRENT TASK
MOVEQ.L #64,D1 ;SET PRIORITY TO 64
MOVE.L #1000000,D6
@000 XSTP ;SET PRIORITY
SUBQ.L #1,D6 ;DONE?
BGT.S @000 ;N
RTS
* PDOS BENCHMARK #4: SEND TASK MESSAGE
*  CLR.L  D0 ;SELECT TASK #0
  LEA.L  MES01(PC),A1 ;POINT TO MESSAGE
  MOVE.L  #100000,D6
  @000  XSTM ;SEND MESSAGE
         XKTM ;READ MESSAGE BACK
         SUBQ.L  #1,D6 ;DONE?
         BGT.S @000 ;N
         RTS
  MES01  DC.B  'BENCH #13',0
         EVEN
         PAGE

* PDOS BENCHMARK #5: READ TIME OF DAY
*  MOVE.L  #100000,D6
  @000  EQU  *
         XRTP
         SUBQ.L  #1,D6 ;DONE?
         BGT.S @000 ;N
         RTS
         end
APPENDIX G

G. Special Locations

The following table describes some special locations in the EPROM. These locations define the default setup of the name of the startup file, user program location and RAM disk addresses.

The locations shown in the table can be changed by the user to adapt VMEPROM to every environment. To make the necessary changes, please conduct the following steps:

1. Read the EPROMs with an EPROM programmer
2. Modify the code
3. Burn new EPROMs and keep the old ones in a safe location
4. Insert the new EPROMs in the CPU board and verify the changes

G1. VMEPROM Configuration Table

The Configuration Table or User Alterable Memory Locations contains several entries which can be altered in order to modify the VMEPROM environment to meet the requirements of a customer. The base address of the table can be obtained by invoking the VMEPROM command `info`.

```
? info
FORCE IBC-20 REV 2.10 with Gate Array FGA-002 at $FFD00000
VMEPROM Version 3.00 at $FFE1001C
Processor is a MC68020

Address to customize VMEPROM
  Configuration table at $FFE1B180 (base address of configuration table)
...
```

The structure of the configuration table is shown on the next page and the entries are described in the following pages.
Table 1: Structure of the User Alterable Memory Locations

The following table is provided by the VMEPROM.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0₁₆</td>
<td>22</td>
<td>&quot;SY$STRT&quot;</td>
<td>NUL terminated string containing the name of the startup file</td>
</tr>
</tbody>
</table>
| 16₁₆   | 2    | 8₁₀, 2048₁₀, 40800000₁₆ | Disk number of the first RAM disk entry  
Size of the RAM disk in 256 byte sectors  
Base address of the RAM disk |
| 2      | 2    | 8₁₀, 2048₁₀, 40700000₁₆ | Disk number of the second RAM disk entry  
Size of the RAM disk in 256 byte sectors  
Base address of the RAM disk |
| 2      | 2    | 8₁₀, 256₁₀, FFC800000₁₆ | Disk number of the third RAM disk entry  
Size of the RAM disk in 256 byte sectors  
Base address of the RAM disk |
| 2E₁₆   | 18   | "SY$DSK" | NUL terminated string containing the default RAM disk name to be used when the RAM disk is being initialized |
| 40₁₆   | 4    | 40800000₁₆, FF000000₁₆, FFC80000₁₆ | These four entries contain addresses where execution continues after the VMEPROM kernel has been initialized |
| 50₁₆   | 4    | "USER" | Identification used by disk drivers |
| 54₁₆   | 1    | 03₁₆ | WAITFLAG |
| 55₁₆   | 1    | 01₁₆ | MEDMA |
| 56₁₆   | 1    | 07₁₆ | Reserved |
| 57₁₆   | 5    | FF₁₆ | Reserved |
| 5C₁₆   | 2    | 16₁₀ | Reserved (number of hashing buffers: not supported) |
| 5E₁₆   | 4    | 00000000₁₆ | Reserved (amount of memory dedicated to the MEtask: not supported) |
| 62₁₆   | 1    | B0₁₆ | RDSKSEL |
| 63₁₆   | 1    | 01₁₆ | RDSKLSZ |
| 64₁₆   | 1    | 0F₁₆ | Reserved |
Offset 0<sub>16</sub>: Startup Filename
This entry contains the ASCII encoded name of the startup file to be executed by the VMEPROM shell when bit 1 of the lower rotary switch is cleared (0).

The name of the startup file must be a valid VMEPROM file name, must be terminated by a NUL character, should not be longer than 19 characters, and must be left justified. In the case of a name shorter than 19 characters, the unused characters must be filled with NUL characters.
By default, the entry contains the string "SY$STRT".

Offset 40<sub>16</sub>: Program Start Table
This table contains four addresses where execution continues after the VMEPROM kernel has been initialized. Depending on the state of the bits 2 and 3 of the lower rotary switch, one of the four entries is fetched to continue execution at the particular address.

Offset 50<sub>16</sub>: This entry contains the string "USER" which indicates the validity of the data in the following entries.

Offset 54<sub>16</sub>: Wait Flag
The least significant bits (bits 0 and 1) are used to control whether VMEPROM waits for the availability of the hard disk.

The state of the first bit (bit 0) is only considered when the second bit (bit 1) is set (1). VMEPROM waits for the availability of the hard disk when bit 1 is set (1) and does not wait when the bit is cleared (0). In the former case, VMEPROM evaluates the state of the least significant bit. If bit 0 is set, then VMEPROM notifies the user that it is waiting for the hard disk to become available by displaying a message on the console. Otherwise, bit 0 is cleared and VMEPROM works in silent mode. In the latter case, VMEPROM does not wait until the hard disks are ready.

Offset 55<sub>16</sub>: Start DMA Task
The least significant bit (bit 0) of this entry specifies whether the DMA task will be started by the Application Command Interface. If this bit is set (1), then the ACI starts the DMA task after reset; otherwise, when this bit is cleared (0), the ACI does not start the DMA task.
All other bits (bits 1 to 7) must be cleared.

Offset 56<sub>16</sub>: This entry is reserved and should not be altered!

Offset 57<sub>16</sub>: This entry is reserved and should not be altered!
Offset 5C₁₆:
The following two bytes are reserved and should not be altered!

Offset 5E₁₆:
The following four bytes are reserved and should not be altered!

The following four entries in the configuration table relate to the configuration of RAM disks provided by the VMEPROM:

Offset 62₁₆: RDSKSEL
The state of this entry specifies at which physical address the RAM disk has to be located. The bits 4 and 5 are used to select one of four possibilities, which are listed in the following table.

<table>
<thead>
<tr>
<th>RDSKSEL</th>
<th>Base Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 5</td>
<td>Bit 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>40800000₁₆</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>40700000₁₆</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>FFC80000₁₆</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>top-of-memory</td>
</tr>
</tbody>
</table>

All parameters of the RAM disk -- size, base address, and the proper disk number -- are obtained from the RAM disk table, beginning at offset 16₁₆ within the configuration table, depending on the state of bits 4 and 5.

However, if the RDSKSEL does not specify location of the RAM disk at top-of-memory, then the VMEPROM always allocates 32 KB of memory on the top-of-memory, which is to be used for an on-board RAM disk but does not initialize the RAM disk.

The most significant bit (bit 7) of the entry RDSKSEL specifies whether the RAM disk has to be initialized. If the most significant bit is set (1), then the RAM disk is not initialized. In the case that the most significant bit is cleared (0), the RAM disk is being initialized by the firmware. In the latter case, all data in the RAM disk are lost!
Offset 63_{16}: RDSKSEL
In case of an on-board RAM disk (RDSKSEL = XX11XXXX), the size of such a RAM disk can be customized by this entry. It specifies the number of 32KB pages the RAM disk consists of, but should not exceed the number of four 32KB pages. If this entry specifies more than four pages, then the firmware automatically limits the number of pages to four. When no page is specified (RDSKLSZ = 0), then the firmware assumes 32KB of memory is to be allocated for the RAM disk.

Offset 16_{16}: RAM Disk Table
The RAM disk table contains the description of three RAM disks which are evaluated by the VMEPROM depending on the state of the bits 4 and 5 of the entry RDSKSEL. Each RAM disk descriptor consists of the following:

1. The disk number assigned to the RAM disk,
2. The size of the RAM disk specified in 256 byte sectors, and
3. The address where to locate the RAM disk.

The parameters of the RAM disk, which are located on top-of-memory, are defined by the entry RDSKLSZ in the configuration table (size), and the other parameters are known by VMEPROM during the startup phase.

Offset 2E_{16}: RAM Disk Name
This entry includes the ASCII encoded name of the RAM disk selected by the bits 4 and 5 of the entry RDSKSEL. The name is assigned to the particular RAM disk only when the most significant bit (bit 7) of the entry RDSKSEL is cleared (0). The name has to be terminated by a NUL character, should not be longer than 16 characters, and has to be left justified. In case of a name shorter than 16 characters, the unused bytes have to be filled with NUL characters. By default, the entry contains the string "SY$DSK".

Offset 64_{16}: RAM Disk Name
This entry is reserved and should not be altered!
APPENDIX H

H. Minimum Demands for Device Driver Tasks in Order to Run with VMEPROM

H.1 Device Driver Tasks for Serial Devices

The following commands have to be supported in order that VMEPROM works properly with the device driver task:

OPEN
VMEPROM executes the OPEN command with a data exchange mode of $C0000000. Therefore, the device driver task has to support Direct Memory Access. Furthermore, it has to have the possibility to transfer the data directly into the applications (VMEPROMs) memory.

Positive return values indicate a successful OPEN.

READ
VMEPROM always tries to read exactly 1 character. The read mode is set to $00000002. The WAIT bit is cleared. Therefore, the device driver task is not allowed to wait until the character is available.

Any return value except 0 indicates a READ error.

WRITE
VMEPROM always tries to write exactly 1 character. The write mode is set to $00000002. The WAIT bit is cleared. Therefore, the device driver task is not allowed to wait until the character can be sent.

Any return value except 0 indicates a WRITE error.

CLOSE
The CLOSE command is executed without any additional parameter.

The return value is not used.
SERVICE

Service codes from -1024 to -2047 are reserved for serial drivers; the codes from -1024 to -1279 are reserved for VMEPROM.

Only one service code is used from VMEPROM. It is service number -1026. It has to set the UART parameter.

The following service parameters have to be supported:

service parameter[0]: to define the baudrate used

<table>
<thead>
<tr>
<th>VALUE</th>
<th>BAUDRATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>150</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
</tr>
<tr>
<td>5</td>
<td>1200</td>
</tr>
<tr>
<td>6</td>
<td>2400</td>
</tr>
<tr>
<td>7</td>
<td>4800</td>
</tr>
<tr>
<td>8</td>
<td>9600</td>
</tr>
<tr>
<td>9</td>
<td>19200</td>
</tr>
<tr>
<td>10</td>
<td>38400</td>
</tr>
</tbody>
</table>

service parameter[1]: to define the number of data bits per character

<table>
<thead>
<tr>
<th>VALUE</th>
<th>NUMBER OF DATA BITS PER CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
service_parameter[2]: to define the number of stop bits

<table>
<thead>
<tr>
<th>VALUE</th>
<th>NUMBER OF STOP BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

service_parameter[3]: to define the parity to be used

<table>
<thead>
<tr>
<th>VALUE</th>
<th>PARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>even</td>
</tr>
<tr>
<td>2</td>
<td>odd</td>
</tr>
</tbody>
</table>

service_parameter[4]: to define the flow control to be used

<table>
<thead>
<tr>
<th>VALUE</th>
<th>FLOW CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no handshake</td>
</tr>
<tr>
<td>1</td>
<td>XON/XOFF</td>
</tr>
<tr>
<td>2</td>
<td>RTS/CTS</td>
</tr>
<tr>
<td>3</td>
<td>DTR/DSR/DCD</td>
</tr>
</tbody>
</table>

Any return value except 0 indicates that the device driver task is not able to set the requested parameter.
H.2 Device Driver Tasks for Block Devices

H.2.1 Floppy Devices

The following commands have to be supported in order that VMEPROM works properly with the device driver task:

OPEN
VMEPROM executes the OPEN command with a data exchange mode of $C0000000. Therefore, the device driver task has to support Direct Memory Access. Furthermore, it has to have the possibility to transfer the data directly into the applications (VMEPROMs) memory.

Positive return values are indicating a successful OPEN.
READ
The READ command is executed with a read mode of $80000000. Because of this the device driver task has to wait until the data is read.

The parameters used are:

_remnant[0]: the drive number (0 or 1)
_remnant[1]: reserved (any value should be ignored)

The following return values are allowed:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read successfully completed</td>
</tr>
<tr>
<td>-32</td>
<td>Record not found</td>
</tr>
<tr>
<td>-33</td>
<td>Address mark not found</td>
</tr>
<tr>
<td>-34</td>
<td>Write protect error</td>
</tr>
<tr>
<td>-35</td>
<td>Sector not found</td>
</tr>
<tr>
<td>-36</td>
<td>Overrun error</td>
</tr>
<tr>
<td>-37</td>
<td>CRC error on the disk</td>
</tr>
<tr>
<td>-38</td>
<td>Illegal sector</td>
</tr>
<tr>
<td>-39</td>
<td>Parameters wrong</td>
</tr>
<tr>
<td>-40</td>
<td>Format error</td>
</tr>
<tr>
<td>-41</td>
<td>Timeout</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-49</td>
<td></td>
</tr>
</tbody>
</table>
WRITE

The WRITE command is executed with a write mode of $80000000. Because of this the device driver task has to wait until the data is written.

The parameters used are:

- _remnant[0]: the drive number (0 or 1)
- _remnant[1]: reserved (any value should be ignored)

The following return values are allowed:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Write successfully completed</td>
</tr>
<tr>
<td>-32</td>
<td>Record not found</td>
</tr>
<tr>
<td>-33</td>
<td>Address mark not found</td>
</tr>
<tr>
<td>-34</td>
<td>Write protect error</td>
</tr>
<tr>
<td>-35</td>
<td>Sector not found</td>
</tr>
<tr>
<td>-36</td>
<td>Overrun error</td>
</tr>
<tr>
<td>-37</td>
<td>CRC error on the disk</td>
</tr>
<tr>
<td>-38</td>
<td>Illegal sector</td>
</tr>
<tr>
<td>-39</td>
<td>Parameters wrong</td>
</tr>
<tr>
<td>-40</td>
<td>Format error</td>
</tr>
<tr>
<td>-41</td>
<td>. . . . . . . . . . . . . . . . . . .</td>
</tr>
<tr>
<td>-49</td>
<td>Timeout</td>
</tr>
</tbody>
</table>
CLOSE
The CLOSE command is executed without any additional parameter.

The return value is not used.

SERVICE
Service codes from -2048 to -3071 are reserved for floppy drivers; the codes from -2048 to -2303 are reserved for VMEPROM.

The following services have to be supported from the device driver task:

<table>
<thead>
<tr>
<th>SERVICE CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2049</td>
<td>Set Floppy Parameter</td>
</tr>
<tr>
<td>-2050</td>
<td>Format Floppy</td>
</tr>
</tbody>
</table>

The possible return values are listed in the READ/WRITE command description.

Parameters for the set floppy parameter service:

- service parameter[0]: drive number (0 or 1)
- service parameter[1]: number of cylinders (80)
- service parameter[2]: sectors/cylinder (32)
- service parameter[3]: bytes/sector (coded) (1)

<table>
<thead>
<tr>
<th>VALUE</th>
<th>Bytes/Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>1024</td>
</tr>
<tr>
<td>4</td>
<td>2048</td>
</tr>
<tr>
<td>5</td>
<td>4096</td>
</tr>
</tbody>
</table>
- service parameter[4]: number of heads (2)
- service parameter[5]: RW gap ($20)
- service parameter[6]: format gap ($36)
- service parameter[7]: density (1)

<table>
<thead>
<tr>
<th>VALUE</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>High Density</td>
</tr>
<tr>
<td>1</td>
<td>Double Density</td>
</tr>
</tbody>
</table>

- service parameter[8]: step rate (1)

Parameters for the format floppy service:

- service parameter[0]: drive number (0 or 1)

- service parameter[1]: address of an interleave table

  The interleave table must have as many entries as the floppy has sectors/track, i.e. the following table has an interleave of 0

  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

  while the next one has an interleave of 1.

  1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16

  Both have 16 sectors/track. The size of every entry is 1 byte.
H.2.2 SCSI Devices

The following commands have to be supported in order that VMEPROM works properly with the device driver task.

**OPEN**

VMEPROM executes the OPEN command with a data exchange mode of $C0000000. Therefore, the device driver task has to support Direct Memory Access. Furthermore, it has to have the possibility to transfer the data directly into the applications (VMEPROMs) memory.

The parameters used are:

- **remnant[0]: Buffer count**
  
  If the device driver task is able to cache data this entry defines how many buffers should be used.

- **remnant[1]: Buffer size**
  
  If the device driver task is able to cache data this entry defines the size of each buffer in Bytes.

- **remnant[2]: Controller SCSI ID**
  
  This entry defines which SCSI ID the controller should have. Positive return values indicate a successful OPEN.
READ
The READ command is executed with a read mode of $80000000. Because of this the device driver task has to wait until the data is read.

The parameters used are:

_remnant[0]: SCSI bus ID as returned from the get device list service.

_remnant[1]: Logical block size
VMEPROM uses a block size of 256 bytes.

The following return values are allowed:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read successfully completed</td>
</tr>
<tr>
<td>-50</td>
<td>SCSI error</td>
</tr>
<tr>
<td>-51</td>
<td>Illegal SCSI bus phase</td>
</tr>
<tr>
<td>-52</td>
<td>Illegal SCSI command</td>
</tr>
<tr>
<td>-53</td>
<td>Timeout</td>
</tr>
<tr>
<td>-54</td>
<td>Illegal drive ID</td>
</tr>
</tbody>
</table>
WRITE

The WRITE command is executed with a write mode of $80000000. Because of this the device driver task has to wait until the data is written.

The parameters used are:

_remnant[0]: SCSI bus ID as returned from the get device list service

_remnant[1]: Logical block size

VMEPROM uses a block size of 256 bytes.

The following return values are allowed:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read successfully completed</td>
</tr>
<tr>
<td>-50</td>
<td>SCSI error</td>
</tr>
<tr>
<td>-51</td>
<td>Illegal SCSI bus phase</td>
</tr>
<tr>
<td>-52</td>
<td>Illegal SCSI command</td>
</tr>
<tr>
<td>-53</td>
<td>Timeout</td>
</tr>
<tr>
<td>-54</td>
<td>Illegal drive ID</td>
</tr>
</tbody>
</table>

CLOSE

The CLOSE command is executed without any additional parameter.

The return value is not used.
SERVICE

Service codes from -3072 to -4095 are reserved for floppy drivers; the codes from -3072 to -3327 are reserved for VMEPROM.

The following services have to be supported from the device driver task:

<table>
<thead>
<tr>
<th>SERVICE CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3073</td>
<td>Get Device List</td>
</tr>
<tr>
<td>-3074</td>
<td>Flush All Hashing Buffers</td>
</tr>
<tr>
<td>-3092</td>
<td>Transparent Mode</td>
</tr>
<tr>
<td>-3097</td>
<td>Format Disk</td>
</tr>
</tbody>
</table>

Any return value except 0 indicates an error.

Parameters for the Get Device List service:

- **input parameter:**
  - service_parameter[0]: address of a buffer for the returned data
  - service_parameter[1]: maximum length of the buffer
- **returned data:** status

Structure of the returned data:

```c
typedef struct SCSI_CTRL
{
    unsigned char id; /* SCSI bus ID of the device */
    unsigned char lun; /* logical unit number */
    unsigned char dev_type; /* device type as returned by the INQUIRY command */
    unsigned char flags;
    unsigned long last_block; /* last logical block of the device */
    unsigned long blocksize; /* physical blocksize of the device */
    char dev_name[24]; /* vendor and product information */
} SCSI_CONTROL;

struct
{
    unsigned long dev_count;
    SCSI_CONTROL scntrl[6];
} GDL_PAR;
```
Parameters for the flush service:

  input parameter:
  nothing

  returned data: status

Parameters for the transparent mode service:

  input parameter:

  service_parameter[0]: SCSI bus ID as returned from the get device list service
  service_parameter[1]: SCSI command (byte 0-3)
  service_parameter[2]: SCSI command (byte 4-7)
  service_parameter[3]: SCSI command (byte 8-11)
  service_parameter[4]: pointer to data buffer
  service_parameter[5]: transfer count

  returned data: data returned from the SCSI device/status

Parameters for the format disk service:

  input parameter:

  service_parameter[0]: SCSI bus ID as returned from the get device list service

  returned data: status
THE APPLICATION COMMAND INTERFACE
PROGRAMMING GUIDE
This page was intentionally left blank
Table of Contents

1. Introduction ................................................. 1-1
   1.1 The Logical Devices ........................................... 1-4
   1.2 The Command Control Buffers .................................... 1-5

2. The Complete Description of All Commands Provided by The Application Command Interface ................................. 2-1
   2.1 The OPEN Command ......................................... 2-1
   2.2 The CLOSE Command ....................................... 2-11
   2.3 The READ Command ........................................ 2-13
   2.4 The WRITE Command ....................................... 2-17
   2.5 The SERVICE Command ..................................... 2-21
       2.5.1 The Get Logical Device Number Service .................... 2-25

3. Command Chaining ........................................... 3-1
   3.1 The CCB_ALLOCATE Command ................................. 3-3
   3.2 The CCB_FREE Command ..................................... 3-5

4. Error Codes ................................................. 4-1
   4.1 Common Error Codes .......................................... 4-1
   4.2 Error Codes Related To The OPEN Command ........................ 4-1
   4.3 Error Codes Related To The CLOSE Command ....................... 4-2
   4.4 Error Code Related To The READ Command ........................ 4-2
   4.5 Error Code Especially Related To The WRITE Command ................ 4-2
   4.6 Error Codes Related To The SERVICE Command ..................... 4-2
   4.7 Error Codes Especially Related To The CCB_ALLOCATE Command ........ 4-3
   4.8 Error Codes Especially Related To The CCB_FREE Command ............ 4-3

5. The following example shows how to communicate with the ACI ........ 5-1

   Figure 1: The Access Control Flags of the Command Control Buffer 1-6
   Figure 2: The inquiry and response mode ................................... 2-2
   Figure 3: The data exchange mode ...................................... 2-7
   Figure 4: The read mode .............................................. 2-14
   Figure 5: The write mode .............................................. 2-18

List of Tables

Table 1: The inquiry mode major and minor interrupt numbers ..................... 2-4
Table 2: The response mode major and minor interrupt numbers ..................... 2-6
1. Introduction

Each base board equipped with one or more EAGLE module slots provides a unique software interface - called the Application Command Interface (ACI) - through which the application communicates with specific devices on the EAGLE modules. Furthermore, the interface offers the capability to gain various information about the EAGLE modules and the particular devices on the modules.

All communication through the Application Command Interface is done by the use of special data packets named Command Control Buffers (CCB). These Command Control Buffers are provided and managed by the Application Command Interface. Depending on the contents of such a Command Control Buffer, issued through the Application Command Interface, the underlying software processes the Command Control Buffer and carries out the requested command. The Application Command Interface provides the following five commands:

1. The OPEN command to establish a logical connection between the application and a specific device.
2. The CLOSE command to release an existing logical connection between the application and a specific device.
3./4. The READ and WRITE commands used to initiate data exchanges via an existing logical connection between the application and a device.
5. The SERVICE command to gain generic information about the devices accessible through the Application Command Interface. This command is also used to modify device parameters, to get use of special services provided by a logical group of devices; or to control the operating mode of a certain device driver dealing with a particular device.

The status information about the command issued through the Application Command Interface is passed to the application through the same Command Control Buffer used to send the command through the interface.

A command is "issued" through the Application Command Interface by generating a MAILBOX 0 interrupt on the board providing the Application Command Interface. When the attention of the Application Command Interface has been gained by such an interrupt, then the underlying software verifies the consistency of the contents of the issued Command Control Buffer; passes the packet to the entity dealing with the processing of the command; and finally the entity returns all status information through the processed Command Control Buffer to report the course of the command execution to the application. In general, the entity "returning" the Command Control Buffer through the Application Command Interface uses certain semaphores within the Command Control Buffer to indicate the completion of the issued command, and depending on the state of certain parameters, it probably gains the attention of the application by generating an interrupt described by corresponding parameters.
As mentioned above the application accesses devices on an EAGLE module via a logical connection, rather than directly. Therefore, each device accessible through the Application Command Interface is identified by a unique logical device number which is provided by the interface.

A base board providing the Application Command Interface deposits the NUL terminated string "ACI" beginning at offset $0$ of the board's main memory accessible from the VMEbus; and the VMEbus address of the first Command Control Buffer (CCB 0), provided by the Application Command Interface, is loaded into the long word at offset $4$. Thus, the application intending to communicate with devices through the Application Command Interface, or to get generic information about available devices, has to look for the "ACI" identifier within the VMEbus' standard (A24) and extended (A32) address range.

Any application has to verify whether the base board the application is running on provides the Application Command Interface, too.

If the application has found a board providing the interface, it has to use the first Command Control Buffer, addressed by the content of the long word at offset $4$ of the board's memory, either to issue the SERVICE command to get information about the available devices or other information about the EAGLE modules; or to issue the OPEN command to establish a logical connection between the application and a specific device.

However, before the application uses the first Command Control Buffer to issue a command through the Application Command Interface it has to gain the ownership of the first Command Control Buffer.

The detailed structure of a Command Control Buffer is described in the subsection "The Command Control Buffers".

The Command Control Buffer contains some semaphores to be used to control the access to the buffer, and to indicate various states of the Command Control Buffer. To gain the ownership of the Command Control Buffer a semaphore has to be set to indicate that the buffer is already in use by an application. Due to this fact, the application has to verify the state of this semaphore, and if the semaphore is cleared, that means the Command Control Buffer is available, the application has to set it to prevent the Command Control Buffer from being acquired by another application.

When the application has the ownership of the first Command Control Buffer, it has to prepare the buffer to issue the particular command. The application can only issue the OPEN command or the SERVICE command, to get generic information, through the Application Command Interface. All other commands (CLOSE, READ, and WRITE) are refused by the Application Command Interface because no logical connection between the application and a device exists.
Depending on the command to be issued, the application has to prepare the first Command Control Buffer and has to set another semaphore that indicates that the Command Control Buffer is ready to be passed through the Application Command Interface. To inform the Application Command Interface about the readiness of the first Command Control Buffer used to issue the particular command (OPEN or SERVICE), the application has to generate the MAILBOX 0 interrupt on the appropriate base board.

Now the application has to verify cyclically (polling) the state of the semaphore indicating the readiness of the Command Control Buffer to issue a command, to determine that the command has been carried out by the underlying software. When the command has been carried out, the underlying software returns all status information through the first Command Control Buffer and clears the semaphore, indicating the completion of the issued command. The semaphore described acts as a "BUSY" semaphore set by the application, to indicate that the Command Control Buffer is "passed" to the Application Command Interface in order to be processed, and cleared by the Application Command Interface, to signal that the Command Control Buffer has been processed and is "returned" to the application.

If the OPEN command has been issued through the Application Command Interface, then the first Command Control Buffer contains the address of a Command Control Buffer allocated by the Application Command Interface which is associated with the logical connection between the application and the appropriate device. The application has to use this Command Control Buffer to issue subsequent commands through the Application Command Interface (READ, WRITE, CLOSE, and SERVICE).

In case of the SERVICE command the Command Control Buffer contains further information, depending upon the requested service. Of course, the READ and WRITE commands also need additional parameters.

Independent from the issued command, the first Command Control Buffer has to be released by the application by clearing the semaphore which indicates that the buffer is already in use, to allow other applications to gain the ownership of the first Command Control Buffer and to issue commands through the Application Command Interface.
1.1 The Logical Devices

The devices on available EAGLE modules cannot be accessed from the VMEbus directly, but the Application Command Interface provides a method to access devices on a "higher logical" level. Each device accessible through the Application Command Interface is identified by an unique "logical device number" that has been assigned to the device by the Application Command Interface. Such a logical device number consists of a major device number and a minor device number. The major device number packs up a number of devices with the same characteristics, and the minor device number identifies each device in such a group of devices packed up under the major device number.

In general, devices are divided into two classes: the first class represents devices which can be shared among a number of applications (SHARABLE devices), which means that multiple applications can access the device simultaneously (e.g. SCSI Controller, FD Controller, Ethernet Controller, etc.). Logical connections to a SHARABLE device can be established by multiple applications simultaneously. The second class contains devices which cannot be shared among applications (NON_SHARABLE devices), and only one application can establish a logical connection to such a device.

The device classes can be distinguished by the minor device number assigned to the corresponding device: a minor device number in the range 0 to 31 identifies a NON-SHARABLE device (which means up to 32 devices are packed up under one single major device number); and the minor device number -1 specifies a SHARABLE device.

Furthermore, devices in the classes are divided into groups of devices with the same characteristics (device type): devices which allow communication via a serial communication line (e.g. ethernet, FDDI, RS-232, etc.), devices which communicate via a parallel "bus" (e.g. ordinary parallel I/O peripheral, IEEE-488 Controller, etc.), devices which are attached to mass storage devices (e.g. SCSI Controller, FD Controller, etc.). Thus the Application Command Interface offers accesses to generic SERIAL-, PARALLEL-, and MASS STORAGE devices.

To establish a logical connection to a device the application has to issue the OPEN command through the Application Command Interface with the appropriate logical device number of the device the application wants to communicate with. The Application Command Interface returns a Command Control Buffer associated with the particular device to the application and the application has to use this Command Control Buffer to issue subsequent commands to the "device".
1.2 The Command Control Buffers

As mentioned previously the Command Control Buffer is the basic data structure to issue commands through the Application Command Interface. This data structure of 256 bytes size consists of two logical parts.

The first part (44 bytes) is used to store global information for the device driver dealing with the device the Command Control Buffer is associated with, to control the access to the Command Control Buffer and to reflect the state of a Command Control Buffer.

The second part (212 bytes) is exclusively used to specify the command to be issued through the Application Command Interface, as well as the parameters that accompanies the command. All status information reflecting the course of the processed command are passed through this area to the application.

The generic structure of a Command Control Buffer is described below (using the C programming language elements):

```c
typedef struct _ccb
{
    unsigned long _access_control_flags;
    long (*ME_system_call)();
    struct _ccb *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command_or_status;
    unsigned long remnant[52];
} CCB;
```

The first eleven entries in the data structure described above are common to all Command Control Buffers, independent from the command being issued through the Application Command Interface. The structure of the remaining 53 entries depend on the command issued, and whether the Command Control Buffer is "passed" to the Application Command Interface or "returned" to the application through the Application Command Interface.

**unsigned long _access_control_flags**

This entry represents the Access Control Field consisting of semaphores to control the access to the Command Control Buffer and to reflect the state of the Command Control Buffer.
The semaphores depicted in Figure 1 are defined and described in the following.

- The ALLOCATE semaphore indicates whether a Command Control Buffer is already acquired. If the ALLOCATE semaphore is cleared then the application may gain the ownership of the Command Control Buffer by setting this semaphore. When the semaphore is set it marks the Command Control Buffer as already allocated by another application.

- The BUSY semaphore indicates whether the Command Control Buffer is ready to be processed by the Application Command Interface. The application has to set this semaphore to signal the readiness of the Command Control Buffer to be issued through the Application Command Interface. The BUSY semaphore is cleared when the command has been carried out and the Command Control Buffer is "returned" to the application. Thus, the application may get use of the BUSY semaphore to detect the completion of a command.

- The FINAL semaphore marks the last Command Control Buffer available in the list of Command Control Buffers managed by the Application Command Interface. This semaphore has not to be affected by the application.

- The PROCESS semaphore is used by the Application Command Interface for internal purpose and signal that the command issued through the Application Command Interface has been accepted by the interface, but the command has not been completed (in-service). When the Command Control Buffer is "returned" to the application the semaphore is cleared. Because this semaphore is exclusively used by the Application Command Interface for its own purpose, it should never be affected by an application.

**Figure 1: The Access Control Flags of the Command Control Buffer**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Busy</td>
<td>Process</td>
<td>Final</td>
<td>Reserved</td>
<td>...</td>
</tr>
</tbody>
</table>
long (*ME_system_call)()
This entry contains the address of a routine supplied by the Application Command Interface which provides specific services. This address is exclusively used by a device driver dealing with the device associated with the Command Control Buffer, and should not be altered by the application!

struct _ccb *ccb_link
This entry addresses a Command Control Buffer chained to this Command Control Buffer. If no Command Control Buffer is chained then this entry contains the value zero.

The application may issue a command to cause to chain up a certain number of Command Control Buffers to this Command Control Buffer. If the application likes to get rid of the Command Control Buffers chained to this Command Control Buffer it has to issue a command to release all Command Control Buffers chained to the Command Control Buffer.

The application should not affect this entry!

long last_command
This entry contains the command code of the last command issued through the Application Command Interface.

The application should not affect this entry!

unsigned long _remnant[7]
These entries are reserved for future use and should not be affected by the application.

long command_or_status
This entry is used by the application to specify the command to be "passed" through the Application Command Interface (the type of the Command Control Buffer); and the entries _remnant[0] to _remnant[51] contain further command parameters. When the Command Control Buffer is "returned" through the Application Command Interface this entry contains the status and the entries _remnant[0] to _remnant[51] contain further status information.

In general, the zero integer value (OK) indicates that the command has been completed successfully, whereas a negative integer value reports an error. The values -1 to -31 are dedicated exclusively to the Application Command Interface to indicate common errors. All other values beginning with the value -32 are returned by the device driver dealing with the device the Command Control Buffer is associated with.
2. The Complete Description of All Commands Provided by The Application Command Interface

The following subsections describe each command provided by the Application Command Interface in detail and discuss the appropriate structure of the Command Control Buffers to issue the particular command through the Application Interface, as well as the structure of the Command Control Buffer "returned" through the interface to the application.

2.1 The OPEN Command

The OPEN command requests to establish a logical connection between the application and a physical device; the appropriate Command Control Buffer is structured as presented below.

Whenever an OPEN command is issued through the Application Command Interface the underlying software verifies whether it is necessary to initialize the specific physical device. If a physical device can be owned by more than one application, like floppy disk controllers, or SCSI controllers, the certain device is being initialized only on the receipt of the very first OPEN command. In contrast, a physical device, which may be owned by only one single application, like a serial channel of a serial communication controller, is initialized upon the receipt of every OPEN command.

```c
typedef struct _ccb_open_command
{
    unsigned long _access_control_flags;
    long (*ME_system_call)();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned long logical_device_number;
    unsigned long inquiry_mode;
    unsigned long response_mode;
    unsigned long data_exchange_mode;
    unsigned long response_mode_address;
    unsigned long _remnant[47];
} CCB_OPEN_COMMAND;
```
_access_control_flags:
The BUSY semaphore has to be set to indicate the readiness of the Command Control Buffer to be processed; all other semaphores within the Access Control Field have to be left unaffected.

command:
The value $00$ indicates that the Command Control Buffer is used to issue the OPEN command through the Application Command Interface.

logical_device_number:
The logical device code denotes the device the application likes to communicate with. The Application Command Interface translates this code using all information provided by the EAGLE Module Software Interface to determine the appropriate physical device. The application can obtain a list of logical device numbers, relating to a group of physical devices with the same functional characteristics using the SERVICE command GET LOGICAL DEVICE NUMBER.

inquiry_mode:
The inquiry mode describes the way the application prefers to gain the attention of the Application Command Interface when it will issue subsequent commands. Virtually, the Application Command Interface’s attention is gained by the generation of a specific interrupt on the corresponding base board which may be one of the following interrupts:

- one of the seven VMEbus interrupts, or
- one of the two FORCE Message Broadcast interrupts, or
- one of the eight Mailbox interrupts.

The least significant eight bits of the inquiry mode contain the major interrupt number and the minor interrupt number as shown in Figure 2. The major interrupt number specifies the interrupt class - one of the interrupts listed above -, whereas the minor interrupt number specifies which of the interrupts in the class is being used. Refer to Table 1 for a list of the different major and minor interrupt numbers.

---

**Figure 2: The inquiry and response mode**

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Vector Number</td>
<td>IRQ Level</td>
<td>Major Interrupt Number</td>
<td>Minor Interrupt Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The interrupt request level to be assigned to the particular interrupt is contained by bits 8 through 15 and has to be one of the MC680XX interrupt request levels. The Application Command Interface uses this value to set the corresponding Interrupt Control Register of the FORCE Gate Array-002 on the base board.

If one of the VMEbus interrupts is specified to gain the attention of the Application Command Interface then bits 16 through 23 have to contain the exception vector number provided by the VMEbus interrupter during the interrupt cycle. The most significant eight bits of the inquiry mode are reserved and should be cleared.

**response_mode:**

The response mode describes the way the application prefers to be informed about the completion of a command and may identify one of the following four modes:

- The POLLING mode where the application has to verify the state of the BUSY semaphore within the Access Control Field of the certain Command Control Buffer to detect the completion of a command.

- The MAILBOX interrupt mode where the Application Command Interface generates one of the eight mailbox interrupts on the board on which the application is running. Obviously, this mode can be selected only if a FORCE Gate Array FGA-002A is on the board where the application runs.

- The VMEbus interrupt mode where the Application Command Interface initiates an interrupt cycle on the VMEbus to inform the application about the completion of a command.

- The FORCE Message Broadcast interrupt mode where the Application Command Interface executes a FMB cycle on the VMEbus to inform the application about the completion of a command. Obviously, this mode can be selected only if a FORCE Gate Array FGA-002A is on the board where the application runs.
Table 1: The inquiry mode major and minor interrupt numbers

<table>
<thead>
<tr>
<th>Major Interrupt Number</th>
<th>Minor Interrupt Number</th>
<th>Interrupt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>$0</td>
<td>VMEbus interrupt 1</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>VMEbus interrupt 2</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>VMEbus interrupt 3</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>VMEbus interrupt 4</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>VMEbus interrupt 5</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>VMEbus interrupt 6</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>VMEbus interrupt 7</td>
</tr>
<tr>
<td>$2</td>
<td>$0</td>
<td>FMB channel 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>FMB channel 1</td>
</tr>
<tr>
<td>$3</td>
<td>$0</td>
<td>Mailbox 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>Mailbox 1</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>Mailbox 2</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>Mailbox 3</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>Mailbox 4</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>Mailbox 5</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>Mailbox 6</td>
</tr>
<tr>
<td></td>
<td>$7</td>
<td>Mailbox 7</td>
</tr>
</tbody>
</table>
The least significant eight bits of the response mode contain the major interrupt number and the minor interrupt number as shown in Figure 2. The major interrupt number specifies the interrupt class - one of the interrupts listed above -, whereas the minor interrupt number specifies which of the interrupts in the class is being used. Refer to table 2 for a list of the different major and minor interrupt numbers.

In contrast to the inquiry mode it is possible to specify the POLL mode; in this case the application has to detect the completion of a command upon the state of the BUSY semaphore within the Access Control Field of the particular Command Control Buffer.

The interrupt request level is reserved for the response mode and should be cleared.

If one of the VMEbus interrupts is specified to inform the application about the completion of a command then bits 16 through 23 have to contain the exception vector number provided by the VMEbus interrupter during the interrupt cycle. The most significant eight bits of the response mode are reserved and should be cleared.
Table 2: The response mode major and minor interrupt numbers

<table>
<thead>
<tr>
<th>Major Interrupt Number</th>
<th>Minor Interrupt Number</th>
<th>Interrupt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$0</td>
<td>No interrupt, POLL mode</td>
</tr>
<tr>
<td>$1</td>
<td>$0</td>
<td>VMEbus interrupt 1</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>VMEbus interrupt 2</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>VMEbus interrupt 3</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>VMEbus interrupt 4</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>VMEbus interrupt 5</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>VMEbus interrupt 6</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>VMEbus interrupt 7</td>
</tr>
<tr>
<td>$2</td>
<td>$0</td>
<td>FMB channel 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>FMB channel 1</td>
</tr>
<tr>
<td>$3</td>
<td>$0</td>
<td>Mailbox 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>Mailbox 1</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>Mailbox 2</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>Mailbox 3</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>Mailbox 4</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>Mailbox 5</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>Mailbox 6</td>
</tr>
<tr>
<td></td>
<td>$7</td>
<td>Mailbox 7</td>
</tr>
</tbody>
</table>
**data_exchange_mode:**

The data exchange mode defines the way the data has to be interchanged between the application and a physical device and describes the location of the data to be transferred. As shown in Figure 3 below, the most significant two bits specify the data exchange mode: the DMA semaphore specifies whether the data has to be transferred by Direct Memory Access or by the Microprocessor; and the GLOBAL semaphore identifies whether to transfer data via the VMEbus to, or from a buffer provided by the application, or via the local data paths to, or from a buffer offered by the device driver.

In particular, if the GLOBAL semaphore is set then the data is transferred via the VMEbus by either the Direct Memory Access Controller or by the Microprocessor according to the state of the DMA flag. If the DMA flag is set then the Direct Memory Access Controller transfers the data, otherwise the microprocessor carries out the data transfer. The direction of the data transfer depends on the data transfer command - READ or WRITE - initiated by the application. If the GLOBAL flag is cleared then the application assumes that the device driver provides a buffer used to accumulate the data received from a physical device or to store the data to be transferred to a physical device. Thus, in this case the data transfer between the application and a physical device proceeds in the two steps: in the first step the application has to lead the Application Command Interface to supply an internal buffer used to store the data to be transferred to a physical device, or to accumulate the data received from a physical device. Depending upon the data transfer to be carried out, the application has to move the data from its own buffer to the internal buffer at the beginning of the WRITE command; or it has to copy the data from the internal buffer to its private buffer at the end of the READ command.

---

**Figure 3: The data exchange mode**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>CPU</td>
<td>LOCAL</td>
<td>GLOBAL</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**response_mode_address:**

If the response mode either specifies one of the mailbox interrupts or one of the FMB interrupts to be used to inform the application about the completion of a command then the response mode address has to contain the address of the particular mailbox or FMB channel to be accessed from the VMEbus to gain the application’s attention.
_remnant:
This data area may be used by the device driver for additional parameters. For further
information please refer to the detailed description of the device driver.

When the OPEN command has been carried out the status of the completion of the command
is returned through the same Command Control Buffer used to issue the command. The
structure of the corresponding Command Control Buffer is structured as described below.

```c
typedef struct _ccb_open_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    CCB *ccb;
    long ccb_number;
    unsigned long ACI_inquiry_address;
    unsigned long _remnant[ 49 ];
} CCB_OPEN_STATUS;
```

_access_control_flags:
The BUSY and the PROCESS semaphore are both cleared to signal the completion of the
command. All other semaphores are unaffected.

status:
The status reports the course of the command and indicates one of the following cases:

ACI_OK:
Indicates the successful termination of the command and the other entries within the
Command Control Buffer contain further information.

ACI_E_ILLEGAL_COMMAND:
An illegal command code has been specified.

ACI_E_INCONSISTENT_COMMAND_CHAIN:
Inconsistent command chain.

ACI_E_BUS_ERROR:
A BUS / ADDRESS ERROR occurred within a device driver.
ACI_E_OPEN_CCB_ALREADY_IN_USE:
An attempt to establish a logical connection to a physical device is refused by the Application Command Interface due to the fact that the Command Control Buffer is already used for a logical connection to a device.

ACI_E_OPEN_ILLEGAL_INQUIRY_MODE:
An illegal inquiry mode has been specified. Probably, an invalid major or minor interrupt number, or an illegal Interrupt Request Level has been specified, or an illegal Exception Vector Number has been specified. The value is also returned when the data within the inquiry mode are not consistent. For example, if the MAILBOX mode is specified but one or more of the most significant 16 bits are set.

ACI_E_OPEN_ILLEGAL_RESPONSE_MODE:
An illegal response mode has been specified. Probably, an invalid major or minor interrupt number, or an illegal Interrupt Request Level has been specified, or an illegal Exception Vector Number has been specified. The value is also returned when the data within the response mode are not consistent. For example, if the MAILBOX mode is specified but one or more of the most significant 16 bits are set.

ACI_E_OPEN_ILLEGAL_DATA_EXCHANGE_MODE:
An illegal data exchange mode has been specified. This status is returned whenever one or more of the least significant 30 bits are set.

ACI_E_OPEN_ILLEGAL_LOGICAL_DEVICE_NUMBER:
An illegal logical device number has been specified which cannot be translated to its corresponding physical device code by the Application Command Interface.

ACI_E_OPEN_INSUFFICIENT_CCBS:
The Application Command Interface is not able to allocate a Command Control Buffer within its internal Command Control Buffer list.

ACI_E_OPEN_DEVICE_ALREADY_IN_USE:
Another application already owns the physical device and no other can gain the ownership of this device until the certain application releases the logical connection to the device.

ACI_E_OPEN_INSUFFICIENT_MEMORY:
The Application Command Interface cannot allocate the memory required by a device driver when the device driver has to be activated upon the receipt of an OPEN.

ACI_E_OPEN_CANNOT_ACTIVATE_DEVICE_DRIVER:
The Application Command Interface cannot activate the device driver dealing with the physical device.
*ccb:
Addresses the Command Control Buffer allocated by the Application Command Interface. The assigned Control Buffer has to be used by the application to issue subsequent commands through the Application Command Interface.

ccb_number:
Contains the number of the assigned Command Control Buffer and has to be used whenever the application will gain the attention of the Application Command Interface by a FORCE Message Broadcast cycle.

ACI_inquiry_address:
If the inquiry mode specifies to gain the attention of the Application Command Interface by either a mailbox interrupt or a FMB interrupt then it contains according to the major and minor interrupt number of the inquiry mode the address of the particular mailbox or FMB channel to be accessed from the VMEbus.

_remnant:
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.

Because the OPEN command has to be issued through the Command Control Buffer #0, the application has to release the Command Control Buffer after it has gained its own Command Control Buffer by clearing the ALLOCATE semaphore within the Access Control Field. All subsequent commands are issued through the Application Command Interface using the assigned Command Control Buffer.
2.2 The CLOSE Command

The CLOSE command requests to release a logical connection between the application and a physical device, and depending on the type of the physical device to reset the device. After the CLOSE command has been completed, the application still owns the Command Control Buffer used to issue commands through the Application Command Interface. To get rid of the Command Control Buffer, the application has to clear the ALLOCATE semaphore in the Access Control Field to return the Command Control Buffer to the Application Command Interface.

The particular Command Control Buffer is structured as described below.

```c
typedef struct _ccb_close_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long command;
    unsigned long _remnant[ 52 ];
} CCB_CLOSE_COMMAND;
```

_access_control_flags:
The BUSY semaphore has to be set to indicate the readiness of the Command Control Buffer to be processed; all other semaphores within the Access Control Field have to be left unaffected.

_command:
The value $0C indicates that the command control buffer is used to issue the CLOSE command through the Application Command Interface.

_remnant:
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.
After the CLOSE command has been carried out, the status of the completion of the command is returned through the same Command Control Buffer used to issue the command. The corresponding Command Control Buffer is structured as described below.

```c
typedef struct _ccb_close_status {
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    unsigned long _remnant[ 52 ];
} CCB_CLOSE_STATUS;
```

**_access_control_flags:**  
The BUSY and the PROCESS semaphore are both cleared to signal the completion of the command. All other semaphore are unaffected.

**status:**  
The status reports the course of the command and indicates one of the following cases:

**ACI_OK:**  
Indicates the successful termination of the command

**ACI_E_ILLEGAL_COMMAND:**  
An illegal command code has been specified.

**ACI_E_INCONSISTENT_COMMAND_CHAIN:**  
Inconsistent command chain

**ACI_E_BUS_ERROR:**  
A BUS / ADDRESS ERROR occurred within a device driver.

**ACI_E_CLOSE_NO_CONNECTION:**  
The logical connection to the device is already released

**ACI_E_CLOSE_CANNOT_DEACTIVATE_DEVICE_DRIVER:**  
The Application Command Interface cannot deactivate the device driver.

**_remnant:**  
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.
2.3 The READ Command

The READ command initiates a data exchange between a device and the application. The data is transferred from a device to the application. If any data have to be read from a block oriented device then blocks of data are transferred; in case of a character oriented device only bytes can be received from the device. The number of blocks or bytes to be read has to be specified too. The Command Control Buffer to issue a READ command is structured as described below.

```c
typedef struct _ccb_read_command
{
  unsigned long _access_control_flags;
  long (*ME_system_call)();
  CCB *ccb_link;
  long last_command;
  unsigned long _reserved[7];
  long command;
  unsigned char *buffer;
  unsigned long count;
  unsigned long block_number;
  unsigned long read_mode;
  unsigned long _remnant[48];
} CCB_READ_COMMAND;
```

_access_control_flags:
The BUSY semaphore has to be set to indicate the readiness of the Command Control Buffer to be processed; all other semaphores within the Access Control Field have to be left unaffected.

command:
The value $04$ indicates that the command control buffer is used to issue the READ command through the Application Command Interface.

*buffer:
Addresses the buffer where the data read from the device have to be stored.

count:
Specifies either the number of blocks to be read from a block oriented device or specifies the number of bytes to be read from a character oriented device.
block_number:
If any data have to be read from a block oriented device then this entry specifies the number of the block where to start reading the number of blocks specified by count. In case of a character oriented device this entry is negligible.

In particular, the entry block_number is interpreted in different ways depending on the certain device driver: a device driver dealing with a block oriented device will use this entry to determine the block number where to start reading the number of blocks specified by the entry count. In contrast to the mentioned above, a device driver dealing with a character oriented device will only consider the information contained by the entry count.

read_mode:
The read mode specifies the conditions under which the READ command has to be carried out. As shown in Figure 4 it contains one flag to specify the mode of operation. This flag is valid for all device drivers. The usage of all reserved flags is device driver dependent. For further information please refer to the detailed description of the device driver.

The WAIT flag controls whether the READ command has to be carried out either in the wait or the status mode. If this flag is set, the wait mode is selected. In this case the corresponding device driver does not inform the application about the completion of the command until all data blocks or bytes have been read properly or a fail state causes to terminate the operation before all required data have been transferred.

In the status mode - the WAIT flag is cleared - the device driver reports the successful completion of the command only if just as much blocks or bytes are already available as specified by count and transfers the data to the specified buffer. If the number of available data blocks or bytes is less than the required number, the device driver reports an error but enters the number of the data blocks or bytes currently available into the entry count of the Command Control Buffer used to issue the READ command to the device driver. Thus, the application can use this information to read all available data by a subsequent READ command.

Figure 4: The read mode

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.

When the READ command has been carried out by the device driver the completion status is returned through the same Command Control Buffer used to issue the command. The structure of the corresponding Command Control Buffer is described below.

```c
typedef struct _ccb_read_status
{
    unsigned long _access_control_flags;
    long (*ME_system_call)();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long read_mode;
    unsigned long _remnant[48];
} CCB_READ_STATUS;
```

__access_control_flags:__
The BUSY and the PROCESS semaphore are both cleared to signal the command completion. All other semaphores are unaffected.

__status:__
The status reports the state of the completion of the command and either indicates the successful completion or the termination of the command due to the recognition of an error. In the former case a zero is returned; in the latter case a negative value is returned. The following error codes are returned by the Application Command Interface directly.

**ACI_OK:**
Indicates the successful termination of the command

**ACI_E_ILLEGAL_COMMAND:**
An illegal command code has been specified.
ACI_E_INCONSISTENT_COMMAND_CHAIN:
Inconsistent command chain

ACI_E_BUS_ERROR:
A BUS / ADDRESS ERROR occurred within a device driver.

ACI_E_READ_NO_CONNECTION:
The logical connection to a device does not exist
For device driver dependent error codes please refer to the detailed description of the particular device driver.

*buffer:
This entry is not affected by the device driver and still addresses the beginning of the buffer where the data read from the device have been stored.

count:
Contains the number of data blocks and bytes read from the device. In case of any error detected by the device driver the number of bytes may be less than the number specified by the application.

read_mode:
This entry is not affected by the device driver and still contains the read mode as specified by the application.

_remnant:
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.
2.4 The WRITE Command

The WRITE command initiates a data exchange between a device and the application. The data is transferred from the application to a device. If any data have to be written to a block oriented device then blocks of data are transferred; in case of a character oriented device only bytes can be transmitted to the device. The number of blocks or bytes to be written have to be specified too. The Command Control Buffer to issue a WRITE command is structured as described below.

```
typedef struct _ccb_write_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[48];
} CCB_WRITE_COMMAND;
```

_access_control_flags:
The BUSY semaphore has to be set to indicate the readiness of the Command Control Buffer to be processed; all other semaphores within the Access Control Field have to be left unaffected.

command:
The value $08$ indicates that the command control buffer is used to issue the WRITE command through the Application Command Interface.

*buffer:
Addresses the buffer which contains the data to be written to the device.

count:
Specifies either the number of blocks to be written to a block oriented device or specifies the number of bytes to be written to a character oriented device.
block_number:
If any data have to be written to a block oriented device then this entry specifies the number of the block where to start writing the number of blocks specified by count. In case of a character oriented device this entry is negligible.

In particular, the entry block_number is interpreted in different ways depending on the certain device driver: a device driver dealing with a block oriented device will use this entry to determine the block number where to start writing the number of blocks specified by the entry count. In contrast to the mentioned above, a device driver dealing with a character oriented device will only consider the information contained by the entry count.

write_mode:
The write mode specifies the conditions under which the WRITE command has to be carried out. As shown in Figure 5 it contains one flag to specify the mode of operation. This flag is valid for all device drivers. The usage of all reserved flags is device driver dependent. For further information please refer to the detailed description of the device driver.

The WAIT flag controls whether the WRITE command has to be carried out either in the wait or the status mode. If this flag is set, the wait mode is selected. In this case the corresponding device driver does not inform the application about the completion of the command until all data blocks or bytes have been written properly or a fail state causes to terminate the operation before all required data have been transferred. In the status mode - the WAIT flag is cleared - the device driver reports the successful completion of the command only if just as much blocks or bytes can be written to the device as specified by count and transfers the data to the specific device from the buffer. If the number of data blocks or bytes which can be written to the device is less than the required number, the device driver reports an error but enters the number of the data blocks or bytes, that could be written to the device, into the entry count of the Command Control Buffer used to issue the WRITE command to the device driver. Thus, the application can use this information to write the possible amount of data to the device by a subsequent WRITE command.

Figure 5 The write mode

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>RESERVE D</td>
<td>RESERVE D</td>
<td>RESERVE D</td>
<td>RESERVE D</td>
<td>... RESERVE D</td>
</tr>
</tbody>
</table>

2-18
_remnant:

This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.

When the WRITE command has been carried out by the device driver the status of the completion of the command is returned through the same Command Control Buffer used to issue the command. The structure of the corresponding Command Control Buffer is described below.

```c
typedef struct _ccb_write_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[ 48 ];
} CCB_WRITE_STATUS;
```

_access_control_flags:

The BUSY and the PROCESS semaphore are both cleared to signal the completion of the command. All other semaphores are unaffected.

status:

The status reports the state of the completion of the command and either indicates the successful completion or the termination of the command due to the recognition of an error. In the former case a zero is returned; in the latter case a negative value is returned. The following error codes are returned by the Application Command Interface directly.

ACI_OK:

Indicates the successful termination of the command

ACI_E_ILLEGAL_COMMAND:

An illegal command code has been specified.
ACI_E_INCONSISTENT_COMMAND_CHAIN:
Inconsistent command chain

ACI_E_BUS_ERROR:
A BUS / ADDRESS ERROR occurred within a device driver.

ACI_E_WRITE_NO_CONNECTION:
The logical connection to a device does not exist.

For device driver dependent error codes please refer to the detailed description of the particular device driver.

*buffer:
This entry is not affected by the device driver and still addresses the beginning of the buffer containing the data which have been written to the device.

count:
Contains the number of data blocks and bytes written to the device. In case of any error detected by the device driver the number of bytes may be less than the number specified by the application.

write_mode:
This entry is not affected by the device driver and still contains the write mode as specified by the application.

_remnant:
This data area may be used by the device driver for additional parameters. For further information please refer to the detailed description of the device driver.
2.5 The SERVICE Command

The SERVICE command requests special services provided by the Application Command Interface and a specific device driver. The Application Command Interface provides services to control the device driver’s parameter, such as task priority etc., or to allocate additional memory which is dedicated to a logical connection; and a device driver provides services to modify the hardware parameter of a peripheral (changing the transmission rate of a serial communication controller, to enable or disable special functions implemented in the peripheral, like timers, counters, etc.) or to change the operating mode of the device driver. The structure of the Command Control Buffer to issue a SERVICE command is described below.

```c
typedef struct _ccb_service_command
{
    unsigned long _access_control_flags;
    long (*ME_system_call)();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    long service;
    unsigned long service_parameter[51];
} CCB_SERVICE_COMMAND;
```

_access_control_flags:
The BUSY semaphore has to be set to indicate the readiness of the Command Control Buffer to be processed; all other semaphores within the Access Control Field have to be left unaffected.

command:
The value $10$ indicates that the command control buffer is used to issue the SERVICE command through the Application Command Interface.
service:
Specifies the proper service to be carried out by the Application Command Interface or the appropriate device driver. A positive value identifies a service required of the Application Command Interface, whereas a negative value designates a service to be provided by the device driver. (Please refer to the appropriate "EAGLE Module’s Firmware User’s Manual" to get detailed information about the services provided by the device drivers dealing with the devices on the particular EAGLE module.)

The services listed in the table below are provided by the Application Command Interface and the appropriate code has to be specified in the entry "service" to issue the particular service request to the Application Command Interface.

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Logical Device Numbers</td>
<td>1</td>
</tr>
</tbody>
</table>

Services provided by the Application Command Interface.
service_parameter:
Depending on the required service further parameters are defined by this entry. The number and type of these parameters depend on the specific device driver.

```c
typedef struct _ccb_service_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    unsigned long service_parameter[ 52 ];
} CCB_SERVICE_STATUS;
```

_access_control_flags:
The BUSY and the PROCESS semaphore are both cleared to signal the completion of the command. All other semaphores are unaffected.

status:
The status reports the state of the completion of the command and either indicates the successful completion or the termination of the command due to the recognition of an error. In the former case a zero is returned; in the latter case a negative value is returned.

The following error codes are returned by the Application Command Interface directly.

ACI_OK:
Indicates the successful termination of the command.

ACI_E_ILLEGAL_COMMAND:
An illegal command code has been specified.

ACI_E_INCONSISTENT_COMMAND_CHAIN:
Inconsistent command chain.

ACI_E_BUS_ERROR:
A BUS / ADDRESS ERROR occurred within a device driver.
ACI_E_SERVICE_NO_CONNECTION:
The logical connection to a device does not exist.

ACI_E_SERVICE_NOT_SUPPORTED:
Indicates that the specific device driver does not support any SERVICE command.

ACI_SERVICE_UNKNOWN_SERVICE:
Unknown service requested.

For device driver dependent error codes please refer to the detailed description of the particular device driver.

service_parameter:
Depending on the required service further information is returned to the application through this area of the Command Control Buffer. The number of parameters and their meaning depends on the specific device driver. (Please, refer to the detailed description of the particular device driver).

The Application Command Interface provides services to get generic information about devices on available EAGLE modules. These services are described in the following subsection in detail, as well as the information returned by the Application Command Interface.
2.5.1 The Get Logical Device Number Service

The application has to issue the **Get Logical Device Number** service command to obtain a list of logical device numbers of devices of a particular type (e.g. a device that exchanges data via serial communication lines, a device that exchanges data through a parallel interface, etc.). The Application Command Interface returns a list of logical device numbers identifying all devices on the available EAGLE modules that are of the same type as specified by a parameter of the issued SERVICE command. (For a detailed description of these bits, refer to the "EAGLE Module Specification.")

The Application Command Interface returns a table of logical device numbers to the application, and each logical device number consists of two bytes. The most significant byte represents the major device number assigned to device and the least significant byte specifies the maximum number of minor devices packed up under the major device number.

Assuming the Application Command Interface has returned a logical device number $0203$ (major device number = 2, minor device numbers are ranging from 0 to 3), then this value has to be interpreted in the following way: the most significant byte of this value represents the major device number (in this case 2) which corresponds to a device on an available EAGLE module that is of the same type as specified by a parameter of the SERVICE command. The least significant byte (in this case 3) indicates the minor device number of the "last" device packed up under the major device number. Thus, four devices are packed up under one major device number; the minor device number 0 corresponds to the first minor device, the minor device number 1 corresponds to the second minor device, the minor device number 2 corresponds to the third minor device, and last but not least the minor device number 3 corresponds to the fourth minor device packed up under the major device number. The end of the table is indicated by the value $0000$ (major device number = 0, minor device number = 0).

Further parameters have to be passed to the Application Command Interface through the parameter area of the certain Command Control Buffer as described below:

**unsigned long parameter[ 0 ]**
Contains the type of device. (For a detailed description of these bits, refer to the "EAGLE Module Specification.")

**unsigned long parameter[ 1 ]**
Addresses a location within the VMEbus address space where the table of logical device numbers has to be placed by the Application Command Interface. If this entry is cleared, then the Application Command Interface places the logical device numbers within the same Command Control Buffer beginning at the location parameter[ 1 ].
3. Command Chaining

The Application Command Interface supports the capability to issue a sequence of commands through the interface which are executed in successive order. The commands are passed through the Application Command Interface in a chain of Command Control Buffers and each Command Control Buffer is used to issue a single command. The Application Command Interface informs the application about the completion of all commands in the chain only until the last command has been executed successfully, or it informs the application about the abnormal termination of a command when a fail state has been detected.

A command chain is built up when the application issues the CCB_ALLOCATE command through the Application Command Interface via an already existing logical connection to a device. The CCB_ALLOCATE leads the Application Command Interface to allocate a given number of Command Control Buffers and to chain these buffers to the Command Control Buffer associated with the logical connection.

The entry [ccb_link] within the first part of each Command Control Buffer addresses the following Command Control Buffer and the NULL pointer identifies the last Command Control Buffer in the chain (A ‘single’ Command Control Buffer is always the first and last Command Control Buffer in a ‘chain’ consisting of only one Command Control Buffer).

To get rid of the Command Control Buffers chained to a Command Control Buffer associated with the logical connection the application has to issue the CCB_FREE command to ‘return’ the occupied Command Control Buffers to the Application Command Interface.

The following constraints apply to the command chains:

1. Only READ and WRITE commands are allowed within the command chain. SERVICE commands which affect the device driver only can be issued through the Application Command Interface within a command chain.

2. Only the first Command Control Buffer of the chain can be used to issue the CCB_FREE command.

3. The CCB_ALLOTTED command can be used only if the application already has issued an OPEN command through the Application Command Interface and received its own Command Control Buffer associated with the logical connection.
Therefore, the following steps are recommended to build up a command chain:

1. First, a logical connection has to be established between the application and a specific device using the OPEN command.

2. When the application has established a logical connection, and has received its own Command Control Buffer through the Application Command Interface, it can issue the CCB_ALLOCATE command to acquire a specific number of Command Control Buffers.

3. The application must have prepared all Command Control Buffers in the chain - according to the rules mentioned above - before the chain is passed through the Application Command Interface.

4. Once the completion of all commands in the chain has been indicated, the application has to verify the status of each issued command, and then may release the Command Control Buffers in the chain by issuing a CCB_FREE command through the first Command Control Buffer of the chain.

5. The application has to issue the CLOSE command using the remaining Command Control Buffer to release the logical connection to the device.
3.1 The CCB_ALLOCATE Command

The CCB_ALLOCATE command is used to acquire a specific number of Command Control Buffers which will be chained to the Command Control Buffer associated with the logical connection.

The particular Command Control Buffer is structured as described below.

```c
typedef struct _ccb_allocate_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long command;
    long ccb_number;
    unsigned long reserved[ 50 ];
} CCB_ALLOCATE_COMMAND;
```

_access_control_flags:
The BUSY flag has to be set to indicate the readiness of the Command Control Buffer to be processed; all other flags within the Access Control Field have to be left unaffected.

command:
The value $18$ indicates that the command control buffer is used to issue the CCB_ALLOCATE command through the Application Command Interface.

ccb_number:
Number of Command Control Buffers to be allocated and linked up to a chain.
After the CCB_ALLOCATE command has been carried out, the status of the completion of the command in the same Command Control Buffer used to issue the command. The corresponding Command Control Buffer is structured as described below.

```c
typedef struct _ccb_allocate_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    long ccb_number;
    CCB *chain_head;
    unsigned long reserved[ 51 ];
} CCB_ALLOCATE_STATUS;
```

_access_control_flags:
The BUSY and the PROCESS flags are both cleared to signal the completion of the command. All other flags are unaffected.

ccb_link:
On successful completion of the command ccb_link contains a pointer to the next Command Control Buffer in the chain. Otherwise this entry is cleared.

status:
The status reports the course of the command and indicates one of the following cases:

ACI_OK:
Indicates the successful termination of the command.

ACI_E_ILLEGAL_COMMAND:
An illegal command code has been specified.

ACI_E_INCONSISTENT_COMMAND_CHAIN:
Inconsistent command chain.

ACI_E_BUS_ERROR:
Reserved
**ACI_E_ALLOCATE_ILLEGAL_NUMBER_OF_CCBS:**
An illegal number of Command Control Buffers to be allocated has been specified.

**ACI_E_ALLOCATE_INSUFFICIENT_CCBS:**
No more Command Control Buffers available.

**ccb_number:**
Specifies the number of Command Control Buffers which have been allocated.

**chain_head:**
Addresses the Command Control Buffer which is the first CCB in the chain.

### 3.2 The CCB_FREE Command

The CCB_FREE command is used to release all Command Control Buffers of a chain except the first Command Control Buffer of the chain.

The particular Command Control Buffer is structured as described below.

```c
typedef struct _ccb_free_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long command;
    unsigned long reserved[ 52 ];
} CCB_FREE_COMMAND;
```

**_access_control_flags:**
The BUSY flag has to be set to indicate the readiness of the Command Control Buffer to be processed; all other flags within the Access Control Field have to be left unaffected.

**command:**
The value $1C$ indicates that the command control buffer is used to issue the CCB_FREE command through the Application Command Interface.

After the CCB_FREE command has been carried out, the status of the completion of the command is returned through the same Command Control Buffer used to issue the command.
The corresponding Command Control Buffer is structured as described below.

```
typedef struct _ccb_free_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call )();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned long reserved[52];
} CCB_FREE_STATUS;
```

_**access_control_flags:**

The BUSY and the PROCESS flags are both cleared to signal the completion of the command. All other flags are unaffected.

**status:**

The status is always zero and indicates the successful termination of the command.
4. Error Codes

This section lists all error codes which are returned through the Application Command Interface to indicate the fail states detected by the Application Command Interface. All error codes returned by a particular device driver, dealing with a specific device on an EAGLE module, are described in the appropriate "Firmware User’s Manual" of the EAGLE module.

4.1 Common Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_OK</td>
<td>0</td>
</tr>
<tr>
<td>ACI_E_ILLEGAL_COMMAND</td>
<td>-1</td>
</tr>
<tr>
<td>ACI_E_INCONSISTENT_COMMANDCHAIN</td>
<td>-2</td>
</tr>
<tr>
<td>ACI_E_BUS_ERROR</td>
<td>-3</td>
</tr>
</tbody>
</table>

4.2 Error Codes Related To The OPEN Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_E_OPEN_CCB_ALREADY_ASSOCIATED</td>
<td>-5</td>
</tr>
<tr>
<td>ACI_E_ILLEGAL_INQUIRY_MODE</td>
<td>-6</td>
</tr>
<tr>
<td>ACI_E_ILLEGAL_RESPONSE_MODE</td>
<td>-7</td>
</tr>
<tr>
<td>ACI_E_OPEN_ILLEGAL_DATA_EXCHANGE_MODE</td>
<td>-8</td>
</tr>
<tr>
<td>ACI_E_OPEN_ILLEGAL_LOGICAL_DEVICE_NUMBER</td>
<td>-9</td>
</tr>
<tr>
<td>ACI_E_OPEN_INSUFFICIENT_CCBS</td>
<td>-10</td>
</tr>
<tr>
<td>ACI_E_OPEN_DEVICE_ALREADY_IN_USE</td>
<td>-11</td>
</tr>
<tr>
<td>ACI_E_OPEN_INSUFFICIENT_MEMORY</td>
<td>-13</td>
</tr>
<tr>
<td>ACI_E_OPEN_CANNOT_ACTIVATE_DEVICE_DRIVER</td>
<td>-14</td>
</tr>
</tbody>
</table>
### 4.3 Error Codes Related To The CLOSE Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_E_CLOSE_NO_CONNECTION</td>
<td>-5</td>
</tr>
<tr>
<td>ACI_E_CLOSE_CANNOT_DEACTIVATE_DEVICE_DRIVER</td>
<td>-6</td>
</tr>
</tbody>
</table>

### 4.4 Error Code Related To The READ Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_E_READ_NO_CONNECTION</td>
<td>-5</td>
</tr>
</tbody>
</table>

### 4.5 Error Code Especially Related To The WRITE Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_E_WRITE_NO_CONNECTION</td>
<td>-5</td>
</tr>
</tbody>
</table>

### 4.6 Error Codes Related To The SERVICE Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_E_SERVICE_NO_CONNECTION</td>
<td>-5</td>
</tr>
<tr>
<td>ACI_E_SERVICE_NOT_SUPPORTED</td>
<td>-6</td>
</tr>
<tr>
<td>ACI_E_SERVICE_UNKNOWN_SERVICE</td>
<td>-7</td>
</tr>
</tbody>
</table>
4.7 Error Codes Especially Related To The CCB_ALLOCATE Command

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI_EALLOCATE_ILLEGAL_NUMBER_OF_CCBS</td>
<td>-4</td>
</tr>
<tr>
<td>ACI_EALLOCATE_INSUFFICIENT_CCBS</td>
<td>-5</td>
</tr>
</tbody>
</table>

4.8 Error Codes Especially Related To The CCB_FREE Command

None
5. The following example shows how to communicate with the ACI

NOTE: This example has to run on the same board where the ACI is implemented. The communication with the ACI is done in polled mode. This example is programmed to run in a PDOS environment. It can easily be ported to any operating system.

```c
#include "XLIB.h"
#define MAILBOX 0xffd80000L
#define DPR_BASE 0x80000000L
#define ACI_IDENTIFIER 0x41434900L
#define OPEN 0x00L
#define READ 0x04L
#define WRITE 0x08L
#define CLOSE 0x0CL
#define SERVICE 0x10L
#define ALLOCATE 31
#define BUSY 30
#define GET_LOGICAL_DEVICE_NUMBER 1L
#define POLL 0x00
#define MBOX0 0x30
#define IRQL2 0x200L

struct _ccb_t
{
    unsigned long _access_control_flags;
    long (* _ME_system_call ) ();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command_or_status;
    unsigned long _remnant[52];
};

struct _ccb_open_command
{
    unsigned long _access_control_flags;
    long (* _ME_system_call ) ();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned long logical_device_number;
    unsigned long inquiry_mode;
    unsigned long response_mode;
    unsigned long data_exchange_mode;
    unsigned long response_mode_address;
    unsigned long remnant[47];
};
```

struct _ccb_sopen_status
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    struct _ccb_t *ccb;
    long ccb_number;
    unsigned long ACL_inquiry_address;
    unsigned long remnant[49];
};

struct _ccb_close_command
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned long release_state; /* !!!! always cleared !!!! */
    unsigned long _remnant[51];
};

struct _ccb_sclose_status
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned long _remnant[52];
};

struct _ccb_read_command
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long read_mode;
    unsigned long _remnant[48];
};

struct _ccb_sread_status
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long read_mode;
    unsigned long _remnant[48];
};
struct _ccb_write_command
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long command;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[48];
};

struct _ccb_swrite_status
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[48];
};

struct _ccb_cservice_command
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    unsigned long last_command;
    unsigned long _reserved[7];
    long command;
    long service;
    unsigned long parameter[51];
};

struct _ccb_sservice_status
{
    unsigned long _access_control_flags;
    long (*_ME_system_call)();
    struct _ccb_t *ccb_link;
    long last_command;
    unsigned long _reserved[7];
    long status;
    unsigned long _remnant[52];
};

/*
Forwards
*/
static void get_ccb();
static void put_ccb();
static void do_mbox0();
static void wait_not_busy();
static unsigned long do_service();
static short check_device();
static long open_device();
static unsigned long do_floppy_parameter();
static unsigned long do_me_read();
static unsigned long do_me_write();
static long close_device();
call: main()
in : nothing
out : nothing
description:
'main' first waits until the ME has written its identifier. Then, the address of the first CCB is fetched. With this CCB the ACI is asked if there is a floppy device driver task available. If yes, this task is opened. Furthermore a service call for the floppy device driver task is executed. At the end the first CCB0 is released.
called subroutines: get_ccb(), check_device(), open_device(),
set_floppy_parameter(), put_ccb(), do_me_read(),
do_me_write(), close_device()
static void get_ccb(ccb_ptr)
struct _ccb_cservice_command *ccb_ptr;
{ while ( XTAS((char *)&ccb_ptr->_access_control_flags) != 0 )
    /* allocating the CCB with a TAS */
    ;
    /* instruction */
} /* end of 'get_ccb()' */
/* call: put_ccb(ccb_ptr) */
in : ccb_ptr -> address of CCB which is no longer used
out : nothing
description:
put_ccb() makes the previous allocated CCB accessible to other
tasks.
called subroutines: none
*/
static void put_ccb(ccb_ptr)
struct _ccb_cservice_command *ccb_ptr;
{ ccb_ptr->_access_control_flags &= ~(1L << ALLOCATE); /* the CCB is free for other */
} /* end of 'put_ccb()' */
/* call: do_mbox0(ccb_address) */
in : ccb_address -> CCB address
out : Nothing
description:
do_mbox0() initiates a Mailbox 0 interrupt. If the CCB is onboard
the interrupt will come to myself. If the CCB is offboard the
interrupt will be generated at this board.
called subroutines: none
*/
static void do_mbox0(ccb_address)
register unsigned long ccb_address;
{ if ( ccb_address < DPR_BASE ) /* ME onboard? */
    { while (*((char *)MAILBOX < 0 ) /* initiate an onboard Mailbox 0 */
        ;
        /* interrupt */
    } /* do not forget this bracket */
    else /* the CCB is not on this board */
    { while ( *(char *)(0xfcff0000 | ((ccb_address >> 16) & 0xff00) )<0)
        /* initiate a VMEbus Mailbox 0 */
            ;
            /* interrupt */
    } /* end if */
} /* end of 'do_mbox0()' */
/* call: wait_not_busy(ccb_ptr) */
in : ccb_ptr -> address of CCB which is used
out : nothing
description:
wait_not_busy() waits until someone (hopefully the ME) clears
the BUSY bit
called subroutines: none
*/
static void wait_not_busy(ccb_ptr)
struct _ccb_cservice_command *ccb_ptr;
{ while ( ccb_ptr->access_control_flags & (1L << BUSY) )
    ; /* we're waiting until the ME has */
    /* cleared the BUSY bit */
} /* end of 'wait_not_busy' */
/*
call: do_service(ccb_ptr, service_number)
in : ccb_ptr -> CCB address
    service_number -> number of the requested service call
out : error number
description:
called subroutines: do_mbox0(), wait_not_busy()
*/
static unsigned long do_service(ccb_ptr, service_number)
register struct _ccb_cservice_command *ccb_ptr;
unsigned long service_number;
{ ccb_ptr->command = SERVICE; /* we do a SERVICE call */
  ccb_ptr->service = service_number; /* set requested service number */
  ccb_ptr->access_control_flags |= 1L << BUSY;
  do_mbox0(ccb_ptr); /* we have to set the BUSY bit */
  /* and to initiate a Mailbox 0 */
  wait_not_busy(ccb_ptr); /* we're waiting until the ME has */
  /* done its job */
  return(((struct _ccb_sservice_status *)ccb_ptr)->status);
} /* end of do_service() */
/*
call: check_device(ccb_ptr, device, destination)
in : ccb_ptr -> address of CCB which is to use
device -> device mask
destination -> to where the data is to send
out : Major/Minor number of the (first) device or 0 if none
description:
check_device() checks if the accessed target has I/O device of the
type requested in 'device'.
called subroutines: do_mbox0(), wait_not_busy()
*/
static short check_device(ccb_ptr, device, destination)
register struct _ccb_cservice_command *ccb_ptr;
unsigned long device;
register short *destination;
{
    ccb_ptr->command = SERVICE; /* we do a SERVICE call */
    ccb_ptr->service = GET_LOGICAL_DEVICE_NUMBER; /* we want to get logical device */
    ccb_ptr->parameter[0] = device; /* numbers */
    ccb_ptr->parameter[1] = (unsigned long)destination; /* of these devices */
    ccb_ptr->_access_control_flags |= 1L << BUSY; /* set destination of the list */
    do_mbox0(ccb_ptr); /* we have to set the BUSY bit */
    wait_not_busy(ccb_ptr); /* and to initiate a Mailbox 0 */
    if (destination == (short *)0)
    {
        /* is the destination in the CCB ? */
        destination = (short *)(&(ccb_ptr->parameter[1]));
        /* yes, then we have set this address*/
        return(*destination); /* return Major/Minor number */
    }
    return(*destination); /* is the destination in the CCB ? */
}

/*
call: open_device(ccb_ptr, major_minor)
in : ccb_ptr -> address of CCB which is to use
    major_minor -> Major/Minor number of the device
out : ME return value in the CCB
description:
open_device() tries to open an I/O device. The device number is
given in 'major_minor'.
called subroutines: do_mbox0(), wait_not_busy()
*/
static long open_device(ccb_ptr, major_minor)
register struct _ccb_open_command *ccb_ptr;
short major_minor;
{
    ccb_ptr->command = OPEN; /* we do a OPEN call */
    ccb_ptr->logical_device_number = (unsigned long)major_minor; /* set device wanted */
    ccb_ptr->inquiry_mode = IRQL2 | MBOX0; /* interrupt level 2/ Mailbox 0 */
    ccb_ptr->response_mode = POLL; /* set response mode */
    ccb_ptr->data_exchange_mode = 0xc0000000; /* the device driver task has to */
    /* transfer the data directly with */
    /* DMA */
    ccb_ptr->_access_control_flags |= 1L << BUSY; /* we have to set the BUSY bit */
    do_mbox0(ccb_ptr); /* and to initiate a Mailbox 0 */
    wait_not_busy(ccb_ptr); /* we're waiting until the ME has */
    return(((struct _ccb_sopen_status *)ccb_ptr)->status); /* done its job */
} /* end of open_device() */
/*
call: set_floppy_parameter(ccb_ptr, drive)
in : ccb_ptr -> CCB address
drive -> floppy drive number
out : STATUS as returned from the ME in the CCB
description:
set_floppy_parameter executes a set floppy parameter service.
called subroutines: do_service()
*/

static unsigned long set_floppy_parameter(ccb_ptr, drive)
    register struct _ccb_cservice_command *ccb_ptr;
    unsigned long drive;
{
    ccb_ptr->parameter[0] = drive; /* set drive number */
    ccb_ptr->parameter[1] = 80; /* set number of cylinder */
    ccb_ptr->parameter[2] = 32; /* set sectors/cylinder */
    ccb_ptr->parameter[3] = 1; /* set bytes/sector (coded) */
    ccb_ptr->parameter[4] = 2; /* set number of heads */
    ccb_ptr->parameter[5] = 0x20; /* set R/W gap */
    ccb_ptr->parameter[6] = 0x36; /* set format gap */
    ccb_ptr->parameter[7] = 1; /* set density */
    ccb_ptr->parameter[8] = 1; /* set step rate */
    return(do_service(ccb_ptr, -2049L)); /* execute service */
}

/*
call: do_me_read(ccb_ptr, block, buffer, drive)
in : ccb_ptr -> CCB address
    block -> requested block number
    buffer -> address of source data
    drive -> drive number
out : STATUS as returned from the ME in the CCB
description:
do_me_read() reads exactly one block from the given drive.
   It waits until the ME has returned a status. The block size is
   fixed to 256Bytes.
called subroutines: wait_not_busy(), do_mbox0()
*/
static unsigned long do_me_read(ccb_ptr, block, buffer, drive);
register struct _ccb_read_command *ccb_ptr;
unsigned long block;
unsigned char *buffer;
unsigned long drive;
{
    ccb_ptr->command = READ; /* we do a READ call */
    ccb_ptr->buffer = buffer; /* set read buffer */
    ccb_ptr->count = 1; /* we want to read 1 block */
    ccb_ptr->block_number = block; /* block number to read */
    ccb_ptr->read_mode = 0x80000000; /* we want to wait for the data */
    ccb_ptr->remnant[0] = drive; /* set drive number */
    ccb_ptr->remnant[1] = 256L; /* set block size */
    ccb_ptr->access_control_flags |= 1L << BUSY; /* we have to set the BUSY bit */
    do_mbox0(ccb_ptr); /* and to initiate a Mailbox 0 */
    wait_not_busy(ccb_ptr); /* we're waiting until the ME has */
    /* done its job */
    return(((struct _ccb_sopen_status *)ccb_ptr)->status); /* return error value */
} /* end of do_me_read() */

/*
call: do_me_write(ccb_ptr, block, buffer, drive)
in : ccb_ptr -> CCB address
    block -> requested block number
    buffer -> address where the data is to store
    drive -> drive number
out : STATUS as return from the ME in the CCB

description:
do_me_write() writes exactly one block to the given drive.
It waits until the ME has returned a status. The block size is
fixed to 256Bytes.
called subroutines: wait_not_busy(), do_mbox0()
*/
static unsigned long do_me_write(ccb_ptr, block, buffer, drive);
register struct _ccb_write_command *ccb_ptr;
unsigned long block;
unsigned char *buffer;
unsigned long drive;
{
    unsigned long error;
    ccb_ptr->command = WRITE; /* we do a WRITE call */
    ccb_ptr->buffer = buffer; /* set write buffer */
    ccb_ptr->count = 1; /* we want to write 1 block */
    ccb_ptr->block_number = block; /* block number to write */
    ccb_ptr->write_mode = 0x80000000; /* we want to wait until written */
    ccb_ptr->remnant[0] = drive; /* set drive number */
    ccb_ptr->remnant[1] = 256L; /* set block size */
    ccb_ptr->access_control_flags |= 1L << BUSY; /* we have to set the BUSY bit */
    do_mbox0(ccb_ptr); /* and to initiate a Mailbox 0 */
    wait_not_busy(ccb_ptr); /* we're waiting until the ME has */
    /* done its job */
    return(((struct _ccb_sopen_status *)ccb_ptr)->status); /* return error value */
} /* end of do_me_write() */
/call: close_device(ccb_ptr)
in : ccb_ptr  -> address of CCB which is to use
out : ME return value in the CCB
description:
close_device() simply executes a CLOSE command to the given CCB.
The response mode is not of interest because we simply poll the
answer.
called subroutines: do_mbox0(), wait_not_busy()

*/
static long close_device(ccb_ptr)
register struct _ccb_close_command *ccb_ptr;
{ unsigned long error;

    ccb_ptr->command = CLOSE; /* we do a CLOSE call */
    ccb_ptr->_access_control_flags |= 1L << BUSY; /* we have to set the BUSY bit */
    do_mbox0(ccb_ptr); /* and to initiate a Mailbox 0 */
    /* interrupt */
    wait_not_busy(ccb_ptr); /* we're waiting until the ME has */
    /* done its job */
    error = ((struct _ccb_sclose_status *)ccb_ptr)->status; /* get close status */
    put_ccb(ccb_ptr); /* this CCB is no longer used */
    /* return status */
} /* end of close_device() */

5-10
RAM PORT
TABLE OF CONTENTS

1. Accessing the RAM port through the ACI ........................................... 7-1
   1.1 Acquire The RAM port .......................................................... 7-1
   1.2 Reading Data From The RAM port .............................................. 7-9
   1.3 Writing Data To The RAM port ............................................... 7-12

2. Accessing The RAM Port From VMEPROM ........................................ 7-15

3. The Internal Structure Of The RAM Port .......................................... 7-17
INTRODUCTION

The Application Command Interface (ACI) provides a RAM port that can be used as a character oriented input/output port of any VMEPROM task running on the same board as the ACI. Within the VMEPROM environment, the RAM port is assigned to a specific task using one of the appropriate commands offered by VMEPROM. Thus, an application running on another board in the system communicates with the task via the backplane; this means that the application sends VMEPROM commands through the RAM port to the task and receives the responses of the task through the RAM port as well.

1 On the IBC-20 board, the RAM port is either:
   • assigned to the task #0 depending on the state of the third bit of the rotary switch on the front panel;
   • or assigned to every other task using the appropriate VMEPROM commands (CT, ASSIGN, and so on).

2 The data passed through RAM port depends on what the certain task expects as input; a VMEPROM task expects proper VMEPROM commands such as It, md, etc.; whereas a user-written task interprets the data in another context. Independent of the context any data is exchanged through the RAM port "byte per byte".
1. Accessing the RAM port through the ACI

Before any data can be exchanged through the RAM port, an application has to gain the ownership of the RAM port in the same manner as an application establishes a logical connection between itself and a specific device. First, the application has to issue the OPEN command through the ACI specifying the RAM port as the device to be opened. If the application has gained the ownership of the RAM port, then it exchanges data between itself and the RAM port using the READ and WRITE commands provided by the ACI. The number of bytes which can be read from or written to the RAM port using the appropriate commands is limited to one byte, and any attempt to read or write more than one byte will be refused by the ACI. Also, any attempt to issue the SERVICE command to the RAM port will be refused by the ACI, because the RAM port driver does not support this feature. To release the RAM port the CLOSE command has to be issued. In the following subsections all commands to gain the ownership of the RAM port, to exchange data between an application and the RAM port, and to release the RAM port are described in detail.

1.1 Acquire The RAM port

The OPEN command requests the establishment of a logical connection between an application and the RAM port; the appropriate CCB is structured as presented in Figure 1.

Whenever an OPEN command is issued through the Application Command Interface to ‘open’ the RAM port, the ACI verifies whether the RAM port is still available and in this case it takes possession of the RAM port. If the RAM port is already owned by another application, the attempt to acquire the RAM port is refused by the ACI.

typedef struct _ccb_open_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long __reserved[ 7 ];
    long command;
    unsigned long logical_device_number;
    unsigned long inquiry_mode;
    unsigned long response_mode;
    unsigned long data_exchange_mode;
    unsigned long response_mode_address;
    unsigned long __remnant[ 47 ];
} CCB_OPEN_COMMAND;

Figure 1: Structure of the CCB used to gain RAM port ownership
_access_control_flags:
The BUSY flag has to be set to indicate the readiness of the Command Control Buffer to be processed; all other flags within the Access Control Field have to be left unaffected.

command:
The structure member contains the code $00 to mark the OPEN command.

logical_device_code:
Because the RAM port is permanently available through the ACI, the major and minor device number of the RAM port are always the same: both the major device number of $0 and the minor device number -4 ($FC) specify the RAM port. (The ACI keeps track of the major device numbers of all devices available on present EAGLE modules; and due to the fact that the RAM port is managed by the ACI directly and because it is permanently available through the ACI independent of the presence of any EAGLE module, the ACI orders the RAM port at the beginning of its internal device list. Therefore, the major device number assigned to the RAM port by the ACI is $0 and the minor device number -4 denotes the proper RAM port.)

inquiry_mode:
The inquiry mode describes the way an application prefers to gain the attention of the ACI when it issues subsequent commands. Virtually, the ACI’s attention is gained by the generation of a specific interrupt on the corresponding IBC board which may be one of the following interrupts:

- one of the eight Mailbox interrupts,
- one of the seven VMEbus interrupts, or
- one of the two FORCE Message Broadcast interrupts

The least significant eight bit of the inquiry mode contain the major interrupt number and the minor interrupt number as shown in the Figure 2. The major interrupt number specifies the interrupt class - one of the interrupts listed above - whereas the minor interrupt number specifies which of the interrupts in the class is being used. Refer to Table 1 which lists the different major and minor interrupt numbers.

The interrupt request level to be assigned to the particular interrupt is contained by bits 8 through 15 and has to be one of the MC680X0 interrupt request levels. The ACI uses this value to set the corresponding Interrupt Control Register.

If one of the VMEbus interrupts is specified to gain the attention of the ACI then bits 16 through 23 have to contain the exception vector number provided by the VMEbus interrupter during the interrupt cycle. The most significant eight bits of the inquiry mode are reserved and should be reset.

```
<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>43</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>vector number</td>
<td>IRQ level</td>
<td>major</td>
<td>minor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 2: The inquiry and response mode
response_mode:
The response mode describes the way an application prefers to be informed about the completion of a command by the ACI and may identify one of the following four modes:

<table>
<thead>
<tr>
<th>Major Interrupt Number</th>
<th>Minor Interrupt Number</th>
<th>Interrupt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>$0</td>
<td>VMEbus interrupt 1</td>
</tr>
<tr>
<td>$1</td>
<td>$1</td>
<td>VMEbus interrupt 2</td>
</tr>
<tr>
<td>$2</td>
<td>$2</td>
<td>VMEbus interrupt 3</td>
</tr>
<tr>
<td>$3</td>
<td>$3</td>
<td>VMEbus interrupt 4</td>
</tr>
<tr>
<td>$4</td>
<td>$4</td>
<td>VMEbus interrupt 5</td>
</tr>
<tr>
<td>$5</td>
<td>$5</td>
<td>VMEbus interrupt 6</td>
</tr>
<tr>
<td>$6</td>
<td>$6</td>
<td>VMEbus interrupt 7</td>
</tr>
<tr>
<td>$2</td>
<td>$0</td>
<td>FMB channel 0</td>
</tr>
<tr>
<td>$2</td>
<td>$1</td>
<td>FMB channel 1</td>
</tr>
<tr>
<td>$3</td>
<td>$0</td>
<td>Mailbox 0</td>
</tr>
<tr>
<td>$3</td>
<td>$1</td>
<td>Mailbox 1</td>
</tr>
<tr>
<td>$3</td>
<td>$2</td>
<td>Mailbox 2</td>
</tr>
<tr>
<td>$3</td>
<td>$3</td>
<td>Mailbox 3</td>
</tr>
<tr>
<td>$4</td>
<td>$4</td>
<td>Mailbox 4</td>
</tr>
<tr>
<td>$5</td>
<td>$5</td>
<td>Mailbox 5</td>
</tr>
<tr>
<td>$6</td>
<td>$6</td>
<td>Mailbox 6</td>
</tr>
<tr>
<td>$7</td>
<td>$7</td>
<td>Mailbox 7</td>
</tr>
</tbody>
</table>

Table 1: The inquiry mode major and minor interrupt numbers
The POLLING mode where an application has to verify the state of the BUSY flag within the Access Control Field of the certain Command Control Buffer to detect the completion of a command.

The MAILBOX interrupt mode where the ACI generates one of the eight mailbox interrupts on the board on which the application is running. Obviously, this mode can be selected only if a FORCE Gate Array FGA-002A is on the board where the application runs.

The VMEbus interrupt mode where the ACI initiates an interrupt cycle on the VMEbus to inform an application about the completion of a command.

The FORCE Message Broadcast interrupt mode where the ACI executes a FMB cycle on the VMEbus to inform an application about the completion of a command. Obviously, this mode can be selected only if a FGA-002A is on the board where the application runs.

The least significant eight bit of the response mode contain the major interrupt number and the minor interrupt number as shown in the Figure 2. The major interrupt number specifies the interrupt class - one of the interrupts listed above - whereas the minor interrupt number specifies which of the interrupts in the class is being used. Refer to Table 2 which lists the different major and minor interrupt numbers.

In contrast to the inquiry mode it is possible to specify the POLL mode; in this case the application has to detect the completion of a command upon the state of the BUSY flag within the Access Control Field of the particular Command Control Buffer.

The interrupt request level to be assigned to the particular interrupt is contained by the bit 8 through 15 and has to be one of the MC680X0 interrupt request levels. If one of the VMEbus interrupts is specified to inform the application about the completion of a command then bits 16 through 23 have to contain the exception vector number provided by the VMEbus interrupter during the interrupt cycle. The most significant eight bits of the response mode are reserved and should be reset.
<table>
<thead>
<tr>
<th>Major Interrupt Number</th>
<th>Minor Interrupt Number</th>
<th>Interrupt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$0</td>
<td>No interrupt, POLL mode</td>
</tr>
<tr>
<td>$1</td>
<td>$0</td>
<td>VMEbus interrupt 1</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>VMEbus interrupt 2</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>VMEbus interrupt 3</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>VMEbus interrupt 4</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>VMEbus interrupt 5</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>VMEbus interrupt 6</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>VMEbus interrupt 7</td>
</tr>
<tr>
<td>$2</td>
<td>$0</td>
<td>FMB channel 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>FMB channel 1</td>
</tr>
<tr>
<td>$3</td>
<td>$0</td>
<td>Mailbox 0</td>
</tr>
<tr>
<td></td>
<td>$1</td>
<td>Mailbox 1</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>Mailbox 2</td>
</tr>
<tr>
<td></td>
<td>$3</td>
<td>Mailbox 3</td>
</tr>
<tr>
<td></td>
<td>$4</td>
<td>Mailbox 4</td>
</tr>
<tr>
<td></td>
<td>$5</td>
<td>Mailbox 5</td>
</tr>
<tr>
<td></td>
<td>$6</td>
<td>Mailbox 6</td>
</tr>
<tr>
<td></td>
<td>$7</td>
<td>Mailbox 7</td>
</tr>
</tbody>
</table>
data_exchange_mode:
The data exchange mode defines the way the data has to be interchanged between the application and a physical device and describes the location of the data to be transferred. As shown in Figure 3 below, the most significant two bits specify the data exchange mode: the DMA semaphore specifies whether the data has to be transferred by Direct Memory Access or by the Microprocessor; and the GLOBAL semaphore identifies whether to transfer data via the VMEbus to, or from a buffer provided by the application, or via the local data paths to, or from a buffer offered by the device driver.

In particular, if the GLOBAL semaphore is set then the data is transferred via the VMEbus by either the Direct Memory Access Controller or by the Microprocessor according to the state of the DMA flag. If the DMA flag is set then the Direct Memory Access Controller transfers the data, otherwise the microprocessor carries out the data transfer.

The direction of the data transfer depends on the data transfer command - READ or WRITE - initiated by the application. If the GLOBAL flag is cleared then the application assumes that the device driver provides a buffer used to accumulate the data received from a physical device or to store the data to be transferred to a physical device. Thus, in this case the data transfer between the application and a physical device proceeds in the two steps: in the first step the application has to lead the Application Command Interface to supply an internal buffer used to store the data to be transferred to a physical device, or to accumulate the data received from a physical device. Depending upon the data transfer to be carried out, the application has to move the data from its own buffer to the internal buffer at the beginning of the WRITE command; or it has to copy the data from the internal buffer to its private buffer at the end of the READ command.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>CPU</td>
<td>LOCAL</td>
<td>GLOBAL</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Figure 3: The data exchange mode

application_address:
If the response mode either specifies one of the mailbox or FMB interrupts to be used to inform the application about the completion of a command then the application address has to contain the address of the particular mailbox or FMB channel to be accessed from the VMEbus to gain the application’s attention.

When the OPEN command has been carried out by the ACI it returns the status of the completion of the command in the same Command Control Buffer used to issue the command. The structure of the corresponding Command Control Buffer is structured as presented in Figure 4.

```c
typedef struct _ccb_open_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( );
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    CCB *ccb;
    long ccb_number;
    unsigned long ACT_inquiry_address;
    unsigned long _remnant[ 49 ];
} CCB_OPEN_STATUS;
```

Figure 4: Structure of the CCB returned through ACI in response to attempt to open RAM port
_access_control_flags:
The **BUSY** and the **PROCESS** flags are both reset to signal the completion of the command. All other flags are unaffected.

**status:**
The **status** reports the course of the command and indicates one of the following cases:

0: indicates the successful termination of the command and the other entries within the Command Control Buffer contain further information.

-1: indicates that an illegal **command code** has been specified.

-2: inconsistent command chain

-3: a BUS/ADDRESS ERROR occurred within a DEVICE DRIVER TASK

-4: reserved

-5: is as in **OPEN** of the IBC Programming User's Guide (Section 6).

-6: an illegal inquiry mode has been specified. Probably, an invalid major or minor interrupt number, or an illegal **Interrupt Request Level** has been specified, or an illegal **Exception Vector Number** has been specified. The value is also returned when the data within the **inquiry mode** are not consistent. For example, if the **MAILBOX mode** is specified but one or more of the most significant 16 bits are set.

-7: an illegal **response mode** has been specified. Probably, an invalid major, or minor interrupt number, or an illegal **Interrupt Request Level** has been specified, or an illegal **Exception Vector Number** has been specified. The value is also returned when the data within the **response mode** are not consistent. For example, if the **MAILBOX mode** is specified but one or more of the most significant 16 bits are set.

-8: an illegal **data exchange mode** has been specified. This status is returned whenever one or more of the least significant 29 bits are set.

-9: an illegal **logical device number** has been specified which cannot be translated to its corresponding physical device code by the ACI.

-10: signals that the ACI is not able to allocate a Command Control Buffer within its internal Command Control Buffer list.

-11: indicates that another application already owns the physical device and no other can gain the ownership of this device until the certain application releases the logical connection to the device.

-12: **reserved for internal use**

-13: indicates that the ACI cannot allocate the memory required by a device driver task when the device driver has to be activated upon the receipt of an **OPEN**.

-14: indicates that the ACI cannot create the device driver task.
Addresses the Command Control Buffer allocated by the ACI. The assigned Control Buffer has to be used by an application to issue subsequent commands through the Application Command Interface.

contains the number of the assigned Command Control Buffer and has to be used whenever an application will gain the attention of the ACI by a FORCE Message Broadcast cycle.

If the inquiry mode specifies to gain the attention of the ACI by either a mailbox interrupt or a FMB interrupt then it contains according to the major and minor interrupt number of the inquiry mode the address of the particular mailbox or FMB channel to be accessed from the VMEbus.

Because the OPEN command has to be issued through the Command Control Buffer #0, an application has to release the Command Control Buffer after it has gained its own Command Control Buffer by resetting the ALLOCATE flag within the Access Control Field. All subsequent commands are issued through the Application Command Interface using the assigned Command Control Buffer.
1.2 Reading Data From The RAM port

The READ command initiates a data exchange between the *character oriented* RAM port and an application and the data is transferred from the RAM port to an application.

The Command Control Buffer to read data from the RAM port is structured as described in Figure 5.

```c
typedef struct _ccb_read_command {
    unsigned long   _access_control_flags;
    long            ( *ME_system_call ) ( );
    CCB            *ccb_link;
    long            last_command;
    unsigned long   _reserved[ 7 ];
    long            command;
    unsigned char   *buffer;
    unsigned long   count;
    unsigned long   block_number;
    unsigned long   read_mode;
    unsigned long   _remnant[ 48 ];
} CCB_READ_COMMAND;
```

Figure 5: Structure of CCB used to read data from RAM port

_**_access_control_flags:_*

The BUSY flag has to be set to indicate the readiness of the Command Control Buffer to be processed; all other flags within the Access Control Field have to be left unaffected.

_**command:_*

The structure member contains the code $04$ to identify the READ command.

_**buffer:_*

addresses the buffer where the data byte read from the RAM port has to be stored.

_**count:_*

The ACI allows only one byte to be read from the RAM port at the time and refuses any attempt to read more or less than one byte. Thus, the count has to specify always one byte ($1$).

_**block_number:_*

Because the RAM port is a character oriented device this entry is not considered and should be cleared.

_**read_mode:_*

Each read access to the RAM port is carried out in the status mode independent of the state of the WAIT flag. Thus, any attempt to read a byte from the RAM port either returns an available data byte, or is refused if no data is available. It is recommendable to clear all bits.
When the READ command has been carried out by the ACI the status of the completion of the command is returned within the same Command Control Buffer used to issue the command. The structure of the corresponding Command Control Buffer is presented in Figure 6.

```c
typedef struct _ccb_read_status
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ( ) ;
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long read_mode;
    unsigned long _remnant[ 48 ];
} CCB_READ_STATUS;
```

**Figure 6: Structure of CCB returned through ACI in response to attempt to read data from RAM port**

_**_access_control_flags:_*

The BUSY and the PROCESS flags are both reset to signal the completion of the command. All other flags are unaffected.

_**status:_*

The status reports the state of the completion of the command and either indicates the successful completion or the termination of the command due to the recognition of an error. In the former case the zero value is returned; in the latter case a negative value is returned. The different error codes which may be returned are described below in detail:

- **0:** indicates the successful termination of the command and the other entries within the Command Control Buffer contain further information.
- **-1:** illegal command code.
- **-2:** inconsistent chain.
- **-3:** a BUS/ADDRESS ERROR occurred within a DEVICE DRIVER TASK
- **-4:** reserved.
- **-5:** indicates an attempt to read a data byte from the RAM port but the RAM port's internal 'transmit' buffer does not contain any data.
- **-6:** reserved for future use.
- **-7:** indicates an attempt to read more than one data byte or to read less than one data byte from the RAM port at the time.
*buffer:
   This entry is not affected and still addresses the beginning of the buffer where the data byte read from the RAM port has been stored.

count:
   contains the number of data bytes read from the RAM port (always one). In case of any error detected by the ACI the number of data bytes may be less than the number of data to be read as specified by the application.

block_number:
   This entry is not affected and still contains the read mode as specified by the application.

read_mode:
   This entry is not affected and still contains the read mode as specified by the application.
1.3 Writing Data To The RAM port

The WRITE command initiates a data exchange between the character oriented RAM port and an application and the data is transferred from the application to the RAM port.

The Command Control Buffer to write data to the RAM port is structured as described in Figure 7.

```c
typedef struct _ccb_write_command
{
    unsigned long _access_control_flags;
    long ( *ME_system_call ) ();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long command;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[ 48 ];
} CCB_WRITE_COMMAND;
```

Figure 7: Structure of CCB used to write data to RAM port

_access_control_flags:
The BUSY flag has to be set to indicate the readiness of the Command Control Buffer to be processed; all other flags within the Access Control Field have to be left unaffected.

command:
The structure member contains the code $08 to identify the WRITE command.

*buffer:
Addresses the buffer where the data byte to be written to the RAM port is stored.

count:
The ACI allows only one byte to be written to the RAM port at the time and refuses any attempt to write more or less than one byte. Thus, the count has to specify always one byte ($1).

block_number:
Because the RAM port is a character oriented device this entry is not considered and should be cleared.

write_mode:
Each write access to the RAM port is carried out in the status mode independent of the state of the WAIT flag. Thus, any attempt to write a byte to the RAM port either is accepted, or is refused if no more data can be accumulated by the RAM port. So, it is recommendable to clear all bits.

When the WRITE command has been carried out by the ACI the status of the completion of the command is returned within the same Command Control Buffer used to issue the command. The structure of the corresponding Command Control Buffer is presented in Figure 8.
typedef struct _ccb_write_status
{
    unsigned long _access_control_flags;
    long (*ME_system_call)();
    CCB *ccb_link;
    long last_command;
    unsigned long _reserved[ 7 ];
    long status;
    unsigned char *buffer;
    unsigned long count;
    unsigned long block_number;
    unsigned long write_mode;
    unsigned long _remnant[ 48 ];
} CCB_WRITE_STATUS;

Figure 8: Structure of CCB returned through ACI in response to attempt to write data to RAM port

_access_control_flags:
The BUSY and the PROCESS flags are both reset to signal the completion of the command. All other flags are unaffected.

status:
The status reports the state of the completion of the command and either indicates the successful completion or the termination of the command due to the recognition of an error. In the former case the zero value is returned; in the latter case a negative value is returned. The different error codes which may be returned are described below in detail:

0: indicates the successful termination of the command and the other entries within the Command Control Buffer contain further information.

-1: illegal command code.

-2: inconsistent chain.

-3: a BUS/ADDRESS ERROR occurred within a DEVICE DRIVER TASK

-4: reserved.

-5: indicates that an attempt to write data bytes to the RAM port has been refused due to the fact that the RAM port has been ‘locked’.

-6: indicates that an attempt to write data bytes to the RAM port has been refused due to the fact that the internal ‘receive’ buffer of the RAM port cannot accumulate further data.

-7: indicates an attempt to write more than one data byte or to write less than one data byte to the RAM port at the time.
*buffer:
   This entry is not affected and still addresses the beginning of the buffer where the data byte read from
   the RAM port has been stored.

count:
   Contains the number of data bytes written to the RAM port (always one). In case of any error detected
   by the ACI the number of data bytes may be less than the number of data to be written as specified
   by the application.

block_number:
   This entry is not affected and still contains the read mode as specified by the application.

write_mode:
   This entry is not affected and still contains the write mode as specified by the application.
2. Accessing The RAM Port From VMEPROM

VMEPROM is equipped with a UART driver to exchange data via the RAM port and to alter the operating mode of the RAM port\(^3\). This RAM port UART driver is constructed like all other standard VMEPROM (PDOS) UART drivers and thus provides the same functions.

In contrast to the standard UART drivers the 'port' flags related to the RAM port UART driver affect it in a different way. As shown in Figure 9 the 'port' flags consists of eight bits and the RAM port UART driver considers only the C-flag and the I-flag; all other flags are ignored by the driver. The C-flag is interpreted by the kernel rather than by the RAM port driver. And the kernel determines upon the state of this flag how to treat control characters, like CTRL-C, ESC, etc., received via the RAM port. To modify the 'port' flags the VMEPROM command \texttt{bp} has to be used and the state of the certain flags are specified as an argument in the argument list of the command. In the following list each flag and its effect on the RAM port UART driver is described in detail:

- **S:** The \textit{control flow by software} flag specifies whether the data flow via the 'serial' data communication line has to be managed by the XON/XOFF protocol. If this flag is set then the XON and XOFF characters are used to control data flow via the serial data communication line; otherwise the XON/XOFF protocol is not used.

- **C:** The \textit{ignore control character} flag either leads the appropriate routine of the VMEPROM kernel dealing with the character input to interprete received control characters, or to pass the control characters through the kernel without any processing. If the flag is set then all received control characters are passed to the application directly; otherwise the kernel interprets the control characters CTRL-C, CTRL-X, ESC.

- **D:** The \textit{control flow by hardware} flag specifies whether the data flow via the 'serial' data communication line has to be managed by the specific hardware handshake signals. If this flag is set then the DTR signal is used to control data flow via the serial data communication line; otherwise no hardware handshake protocol is used.

- **8:** The \textit{size of character} flag denotes the number of bits used to represent a character to be received or transmitted via the serial data communication line. If the flag is set then the character's size is eight bits; otherwise seven bits are used to represent a character.

- **I:** The \textit{not interrupt driven input} flag controls whether the receipt of a character is indicated by a hardware interrupt. If this flag is set then the receipt of a character is not indicated by an interrupt; otherwise a hardware interrupt is generated to indicate the receipt of a character.

- **P:** The \textit{even parity enable} flag indicates to generate an even parity bit for each character to be transmitted via the serial data communication line and to check the even parity of each character received via the serial data communication line. If this flag is set then the even parity generation and verification is done for each received and transmitted character; otherwise the parity generation and verification is disabled.

\(^3\)The VMEPROM command "bp" can be used to obtain the port number of the RAM port.
**H:** reserved for the VMEPROM kernel's internal purpose

**F:** reserved for the VMEPROM kernel's internal purpose

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>H</td>
<td>P</td>
<td>I</td>
<td>8</td>
<td>D</td>
<td>C</td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 9: RAM Port UART Driver’s ‘port’ Flags
3. The Internal Structure Of The RAM Port

The RAM port provided by the ACI consists of an internal 32 bits width semaphore register and two 128 byte width circular buffers - the 'receive' and 'transmit' buffer - each equipped with two pointers to manage insertion and removal of data. Both, the RAM port driver of the ACI and the RAM port UART driver provided by VMEPROM have access to the internal flag register, the ‘receive’ buffer and the ‘transmit’ buffer of the RAM port as depicted in Figure 10.

Application ACI’s RAM port driver VMEPROM
Command RAM port UART driver
Interface

Figure 10: Internal Structure of the RAM port

Within the context of the RAM port the receive describes the process of writing data through the Application Command Interface to the RAM port’s receive buffer; and the transmit relates to the process of reading data through the ACI from the RAM port’s transmit buffer.

Every access to the RAM port through the ACI and the RAM port’s operating mode are controlled by the specific flags in the internal semaphore register. As shown in Figure 11 the most significant two bits in this register are in use and described below:
• The RPINTR flag either causes to pass directly a received character to the appropriate routine of the VMEPROM kernel dealing with character input, or to store the received character in the RAM port’s internal receive buffer. If the RPINTR flag is cleared then all received data bytes are placed in the receive buffer as long as enough room is available in the buffer. In the case that the RPINTR flag is set then all received characters are passed directly to the kernel of VMEPROM via a specific call.

The RPINTR flag is modified upon the state of the I-flag in the RAM port’s ‘port’ flag whenever the routine UxDB of the VMEPROM’s RAM port UART driver is called (I-flag = 0 -> RPINTR = 1; I-flag = 1 -> RPINTR = 0;)

• The RPLOCK flag is used to refuse any attempt to write further data to the RAM port through the Application Command Interface. If the RPLOCK flag is reset then data bytes can be written to the RAM port; otherwise any attempt to write data to the RAM port is refused by the ACI’s RAM port driver. This flag is set by the UxHW routine provided by the VMEPROM RAM port UART driver to cause to refuse an further attempt to write data bytes to the RAM port from the VMEbus side. The RAM port UART driver’s routine UxLW resets the RPLOCK flag to enable the receipt of further data via the RAM port.4

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPINTR</td>
<td>RPLOCK</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Figure 11: The semaphore register of the RAM port

---

4 The routines UxHW (Signal High Water) and UxLW (Signal Low Water) provided by the VMEPROM RAM port UART driver are called by the VMEPROM kernel depending on the state of the internal type-ahead buffer. Please refer to the “PDOS Developer’s Reference” for more detailed information.
1. HISTORY OF MANUAL REVISIONS

<table>
<thead>
<tr>
<th>Revision No.</th>
<th>Description</th>
<th>Date of Last Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This manual describes the IBC-20 revision 2 firmware.</td>
<td>FEB/05/1993</td>
</tr>
</tbody>
</table>