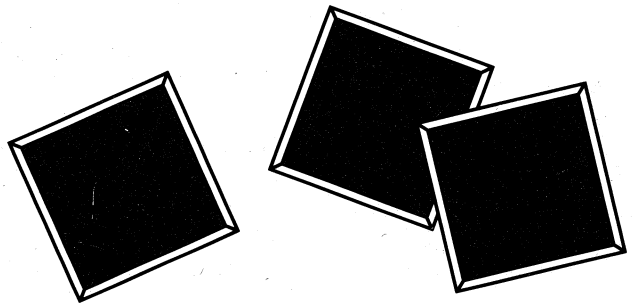
MVME332XT
Serial Intelligent Peripheral Controller Firmware User's Manual



MOTOROLA INC.

MVME332XTFW/D2
MARCH 1992

MVME332XT
SERIAL INTELLIGENT PERIPHERAL CONTROLLER
FIRMWARE USER'S MANUAL

The information in this document has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Furthermore, Motorola reserves the right to make changes to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of others.

SYSTEM V/68 and VMEmodule are trademarks of Motorola Inc.

UNIX is a registered trademark of AT&T.

Second Edition

Copyright 1988, 1992 by Motorola Inc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 GENERAL

This document describes the software interface to the MVME332XT Intelligent Peripheral Controller (IPC). It is intended as an aid in writing device drivers for the I/O devices attached to the MVME332XT. The point of view taken in this document is that of the host CPU. There are many approaches that can be taken in interfacing to the MVME332XT. This document is not intended to specify the structure of the device drivers, as this structure will vary from system to system and programmer to programmer. This user's manual is applicable for any operating system environment the MVME332XT may be used in.

The MVME332XT IPC gives the host system one common interface to two different types of peripheral devices.

The MVME332XT firmware:

- Supports one parallel printer with the Centronics interface, up to 2000 LPM.
- Supports eight RS-232C asynchronous serial ports with the RS-232C interface, up to 19.2K baud without handshaking protocol, and up to 38.4K with software handshake (XON/XOFF) or hardware handshake (RTS/CTS).
- Includes one standard UNIX line discipline 0 and one fast RAW line discipline 1.
- Includes five downloadable line disciplines and download support.

Figure 1-1 illustrates an MVME332XT IPC with the maximum configuration of peripheral devices that it will support. To the host system, these peripherals appear to be on independent controllers. Each device driver can be written independently and does not require any knowledge of the other devices on the MVME332XT.

The protocol for sending commands to all devices is identical, so routines to do this can be shared between all drivers. The protocol consists of enqueueing a packet on a singly linked list referred to in this document as a pipe and then issuing an attention interrupt to the MVME332XT. The packets are returned to the host with status in the same way. The IPC enqueues the packet on another identical pipe and interrupts the host. These pipes are initially established using a handshake protocol through registers in the MVME332XT CSR space.

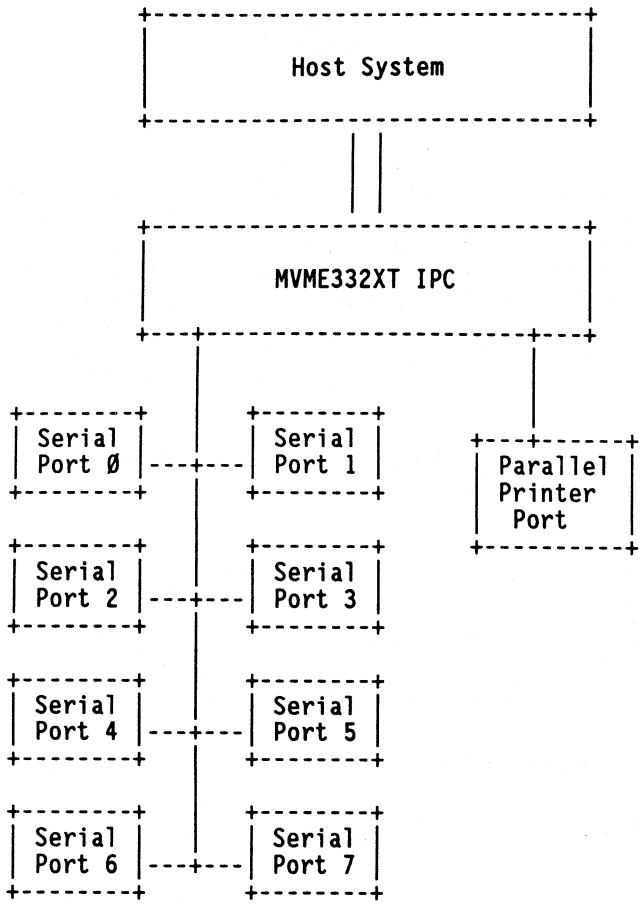


FIGURE 1-1. MVME332XT FIRMWARE FEATURES

1.2 CONVENTIONS

1.2.1 General Conventions

In this manual the following conventions are used:

1. X = don't care bit position.
2. Øx, or $ = hexadecimal value.
3. % = binary value.

1.2.2 Command Packet Conventions

In the command packet descriptions the following conventions are used:

1. Opcodes are hexadecimal.
2. Examples show the hexadecimal values required in the packet to perform the specified operation. These values are enclosed in square brackets. For example, if the command is a logical read, the field includes "[2ØØ1]".
3. Where bit positions are not indicated, the entire field delimited by "|" is used.
4. Unused and optional fields are left blank.
5. msw = most significant word.
6. lsw = least significant word.
7. X = don't care bit position.

1.3 GLOSSARY

Packet

A packet is the block of memory containing the command sent by the host CPU to the IPC. These packets are allocated in the MVME332XT's dual-port memory. Some IPCs read packets in place in global memory. Other IPCs transfer the contents of the packet to local RAM before examining its contents. The MVME332XT IPC firmware requires that all packet, channel, and envelope data structures have to be on its dual-port memory to reduce the bus traffic.

Pipe

A pipe resembles a queue with the exception that there is always a null entry in a pipe to buffer the sender from the receiver. This change allows both the sender and the receiver to access the pipe simultaneously without the need for a semaphore mechanism. A pipe is implemented as a singly linked list since it is always traversed in only one direction. The sender always enqueues entries on the tail of the pipe. The receiver always dequeues entries from the head of the pipe.

Channel

A channel consists of a pair of pipes, a command pipe and a status pipe. A host CPU communicates with an IPC by enqueueing command packets on the command pipe and dequeueing status packets from the status pipe.

CSR

CSR stands for Control/Status Register and refers to address space that can be accessed from both the VMEbus and the local IPC bus. This space is used to convey control and status information.

Device Driver

All the software and firmware in the host and peripheral controller dedicated to the service of a particular device type, such as disk, tape, or serial ports.

IPC

IPC stands for Intelligent Peripheral Controller and refers to boards like the MVME332XT and MVME321 that have their own CPU's on-board. The presence of a dedicated CPU allows these controllers to perform much more complex functions than purely hardware controllers are capable of. This in turn takes overhead off the host CPU and allows it to do other tasks more efficiently.

Kernel

The kernel is the multi-tasking executive routine that controls the execution of all other routines in the IPC. It is responsible for creating processes, killing processes, sending messages from one process to another, deciding which process will be allowed to run next, and several other activities.

Process

A process is a routine that controls a resource or set of resources. It invokes the primitives that interface directly with the hardware, monitors the status of the hardware, signals other processes when a requested action completes, and controls the transfer of data from one place in the system to another. A process has its own CPU register image and is subject to be time sliced or preempted when its quantum time expires.

Memory Management

Allocation of available memory to specific routines for their sole use. This allocation may be static (i.e., until the system is reset, as in the case of the queues used by the kernel) or dynamic (i.e., until the routine deallocates it for use by other routines, as in the case of data buffers associated with a particular command packet).

Message

A message is generally any information that is sent from one place in memory to another. In the MVME332XT messages most frequently mean information sent from one process to another process via the

kernel utilities "send" and "receive". The most common message is a pointer to a structure. This minimizes the amount of data movement while giving the recipient full access to the data it needs.

PCB

PCB stands for Process Control Block and refers to the data block used by the IPC kernel to maintain information about active processes. This information tells the kernel about processes that are waiting for signals from other processes, the priority of each process, and other information needed to run the process.

Primitive

Primitives are the routines that interface directly with the device controller chips on the IPC. These routines provide a common interface to the rest of the software so it does not need to know the idiosyncrasies of each device in the system.

Queue

A queue is simply a waiting line in which data or routines wait for their turn to be moved or allowed to execute.

Semaphore

A semaphore implements an interlock mechanism that permits two independent processes to access and modify the same structure in a controlled way. This semaphore mechanism prevents two processes from trying to modify a structure simultaneously.

1.4 REFERENCE DOCUMENTATION

The following publications may provide additional information. They provide additional detail about the hardware and the application of the MVME332XT firmware. If not shipped with this product, they may be purchased from Motorola's Literature Distribution Center, 616 West 24th Street, Tempe, Arizona 85282; telephone (6Ø2)994-6561.

DOCUMENT TITLE	MOTOROLA PUBLICATION NUMBER
MVME332XT Intelligent Peripheral Controller User's Manual	MVME332XT
MVME71Ø Eight-Channel Serial I/O Module User's Manual	MVME71Ø
SYSTEM V/68 Administrator's Reference Manual	MU43812SAR
SYSTEM V/68 User's Reference Manual	MU4381ØUR
SYSTEM V/68 Programmer's Reference Manual	MU43814PR
SYSTEM V/68 Programmer's Guide	MU43815PG
ADC Kernel Firmware Manual (Preliminary)	MADCKERNEL

CHAPTER 2
SCOPE

This manual contains information on how a CPU board sends commands to the MVME332XT. It also describes all serial port commands, printer port commands, and IPC configuration commands. In addition to the commands and interface protocol, this manual also contains information on how to implement a custom line discipline and how to download it into the MVME332XT's local memory for execution. Some information is provided on hardware and firmware architecture to aid in understanding the operation of the MVME332XT. Summaries of commands and status are provided to make it easier for the experienced user to find frequently referenced information.

The CPU/IPC interface on the MVME332XT is the BPP or Buffered Pipe Protocol. Readers who are already familiar with this interface may skip the HOST/IPC INTERFACE chapter and proceed directly to the sections that deal with MVME332XT specifics.

CHAPTER 3
ERROR HANDLING

3.1 ERROR HANDLING

The "error" field of the packet is cleared when the firmware receives the packet and is set only when an error condition occurs.

Since the MVME332XT firmware handles very high level commands as in the SYSTEM V/68 Input/Output system calls, all returned errors will conform to the UNIX error specification. Some error codes specified in the SYSTEM V/68 Programmer's Guide are used by the firmware, but some may be used in the downloadable line disciplines. The discussion in this section only refers to the error returned by the on-board firmware.

Table 3-1 details all general error codes for the MVME332XT.

TABLE 3-1. MVME332XT ERROR CODES

Name	Code	Descriptions
EIO	5	I/O error. Some physical I/O error. This error indicates that an abnormal hardware condition has occurred that prevents future access to the device.
ENXIO	6	No such device or address. I/O on a device which does not exist or I/O is beyond the limit of the device.
ENOMEM	12	Not enough space. Some commands such as "create a table" for ISP require allocation of local memory. If this request is not satisfied, this error will be returned to the host.
EEXIST	17	Device or address exists. Attempts to create a existing table for the ISP will receive this error code.
EINVAL	22	Invalid argument. One or more command parameters are invalid.

CHAPTER 4
HARDWARE ARCHITECTURE

4.1 GENERAL

The MVME332XT board is an MC68Ø1Ø based Intelligent Peripheral Controller (IPC) designed as a high performance asynchronous serial I/O and parallel printer I/O interface for VMEbus-based microcomputers. A block diagram of the MVME332XT board appears in Figure 4-1.

The MVME332XT consists of the following major components:

CPU - MC68Ø1Ø

The CPU for the MVME332XT is a 12.5 MHz MC68Ø1Ø, running no wait states out of EPROM and RAM.

EPROM

The MVME332XT supports two 28-pin JEDEC sockets for EPROM devices. Sizes of EPROM support range from 32K x 8 to 64K x 8. The firmware resides in the first half of two 64K x 8 devices.

Parallel Interface and Timer - MC6823Ø

The PIT is used on the MVME332XT to provide a timer tick function for the firmware kernel, as well as printer interface and bus interface functions.

Serial Controller - MK68564

The eight serial channel interfaces are implemented with four MOSTEK MK68564 Serial Controllers. Each device supports up to two independent ports. The additional features of the chip include modem status, interrupt vectoring, and direct addressable registers.

Local RAM

The MVME332XT provides 196K bytes of static RAM for intermediate character buffers, firmware stack, and firmware data structures. Over 1ØØK bytes are available for downloadable area.

VMEbus Interrupter

Interrupt to the host is generated by a discrete register. Only one interrupt source can be presented to the host at a time. The local CPU has to wait for this register to become zero prior to issuing a different interrupt level.

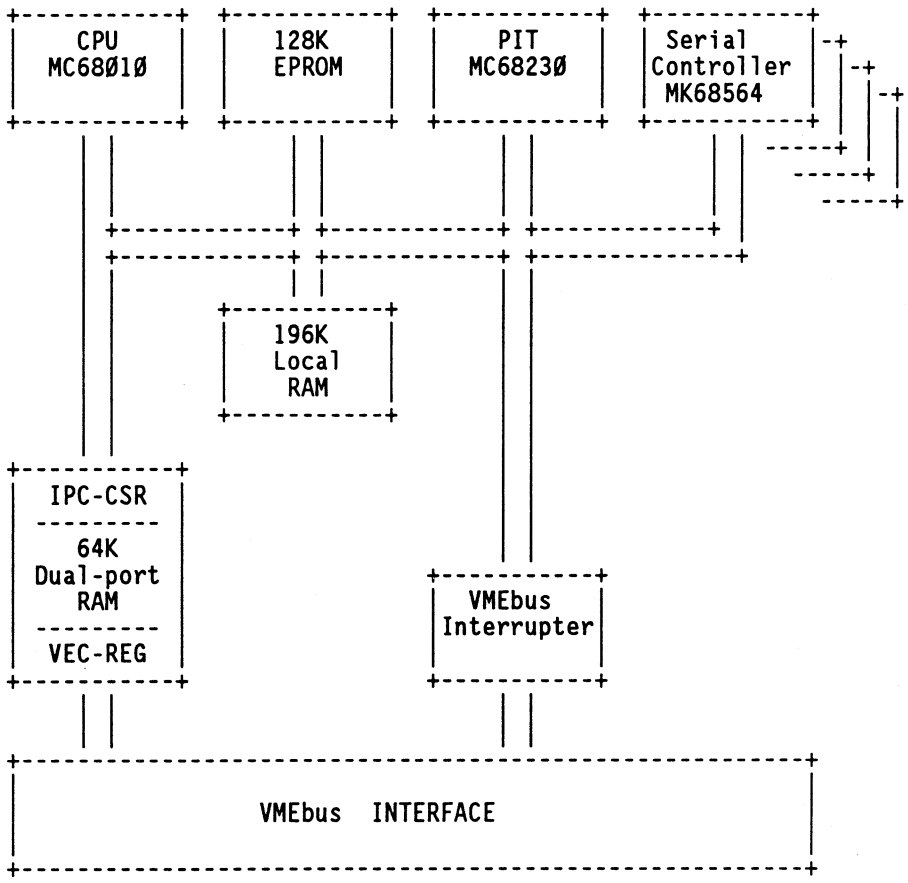


FIGURE 4-1. MVME332XT HARDWARE ARCHITECTURE

Dual-Port RAM

The MVME332XT provides 64K bytes of dual-port RAM that can be shared between the host and the local CPU for packets, channels, envelopes, and ring buffers.

IPC Control Status Register (IPC_CSR)

The first eight words of the Dual-Port RAM are assigned to the IPC Control Status Register (IPC_CSR) which is used by the host in initialization and catastrophic error conditions, as well as sending an attention interrupt to the MVME332XT.

Interrupt Vector Register (VEC_CSR)

The last eight words of the Dual-Port RAM are assigned to the VMEbus Interrupt Vector Register (VEC_REG) which is used by the local CPU to place a vector during the interrupt acknowledge cycle. One word for each level of interrupt, but only lower byte (d7 to dØ) is significant.

VMEbus Interface

The VMEbus interface consists of a bus requester, address and data transceivers and latches, and VMEbus control drivers and receivers.

4.2 CPU BUS OPERATIONS

The MVME332XT hardware is designed around two primary buses. The first bus is the VMEbus interface and is the connection into the system. The second bus is the local processor bus which contains the local CPU, as well as all devices. Normal access to the EPROMs and buffer RAM requires no wait states to the processor. Access times to the other devices are dependent on the cycle time of each device and vary from two to eight wait states.

Only dual-port RAM is accessible from the host. The local CPU can run concurrently while the host accesses the dual-port RAM.

4.3 VMEbus OPERATIONS

VMEbus operations on the MVME332XT are performed by the local CPU via a 16-MEG byte window. Any access to the top 1-MEG byte (address $XXfØØØØØ to $XXffffff) of a window will not go to the bus. Instead, it will be decoded for the local resources. To change the window, the local CPU writes a new value to the VME Address Register (VADR).

The current implementation of the on-board firmware does not access the bus at all. Data transfers between the host and the firmware are done in the dual-port RAM to reduce the bus traffic and to avoid the wait condition while the bus is too busy with other controllers such as disk and tape. Actual bus interface control such as bus request and bus timing are defined in the following paragraphs.

4.3.1 Bus Release

The VMEbus interface provides only one method of bus request operation. This is Release On Request (ROR). In this mode, the bus requester is activated when the local CPU accesses the bus. The bus request is generated and, upon receipt of the bus grant, the transfer occurs. It can be configured for fairness mode operation to reduce bus starvation problems in heavily loaded VME systems.

4.3.2 Bus Error Handling

The MVME332XT generally recovers from VMEbus error conditions by terminating the current operation and returning bus error information to the host. The exception to this is during any channel operations. During these operations, a bus error is considered catastrophic and causes the firmware to assert the VMEbus signal SYSFAIL*, place appropriate status in the CSR, dump all register contents to the dump area, and then halt. During normal data transfer commands, the bus error condition is handled as a fatal error to the command and the failing address along with bus error status are returned to the host in the status packet.

4.4 SERIAL DEVICE CONSIDERATIONS

The MVME332XT requires a MVME71Ø transition module to distribute the RS-232C signals from the P2 connector of the VMEbus backplane to the eight DB-25 connectors. The MVME71Ø also allows the user to configure each port as a DTE or DCE interface.

To ensure proper connection between the RS-232C cable's shield and the VME enclosure, the user should utilize cables terminated with connectors, such that the connector's metal shell is connected to the cable shield lead. The MVME71Ø's serial port connectors connect pin 1 to the VME system's chassis ground via the MVME71Ø front panel. The user is warned not to rely upon a connection between the cable's shield lead and the cable's DB-25 connector pin 1 for proper chassis ground connection. The chassis ground traces on the MVME71Ø circuit board will not handle excessive currents (greater than .5A) flowing between each MVME71Ø DB-25 connector and the chassis ground into which it is installed.

Note that the MVME71Ø's front panel must make electrical contact with the VME system's chassis ground (which ultimately should make electrical contact with the three wire AC line cord's ground return) to ensure proper EMI shielding.

Also note that the MK68564 does not have a DSR input, so the EIA DSR signal is not supported in the MVME71Ø's "Connect to Modem" configuration. In the MVME71Ø's "Connect to Terminal" configuration, the EIA DSR signal is sourced by the MK68564's DTR output signal. Refer to the MVME71Ø User's Manual for more details.

4.5 PRINTER DEVICE CONSIDERATIONS

The MVME332XT parallel printer port, accessible via the front panel mounted Centronics connector, is implemented with the MC6823Ø PIT (Parallel Interface and Timer). The printer connector provides the 36-pin conductor Centronics interface and is of the EMI shielded variety for reduced electrical interference outside the VME system

enclosure. Note that the MVME332XT's front panel must make electrical contact with the VME system's chassis ground to ensure proper EMI shielding.

The user should consult the Centronics compatible printer user's manual for the proper connect type for the printer end of the cable.

4.6 MVME332XT MEMORY MAP

Table 4-1 gives the local address map as seen by the MVME332XT processor.

TABLE 4-1. MVME332XT LOCAL MEMORY MAP

Hex Location	Device
$000000-$efffff	VMEbus window
$f00000-$f2ffff	Local RAM
$f30000-$f3ffff	Dual-port RAM
$f30000-$f30010	Dual-port IPC Control Status Register (IPC_CSR)
$f3fff0-$f3ffff	Vector register (VEC_REG)
$f80000-$f8001f	MK68564 #1 registers
$f82000-$f8201f	MK68564 #2 registers
$f84000-$f8401f	MK68564 #3 registers
$f86000-$f8601f	MK68564 #4 registers
$f88000-$f88010	Debug Service Port
$f90021	Auxiliary Control Register (AUX)
$f90023	VMEbus Interrupt Register (VIR)
$f90025	VMEbus Addr Extension Register (VAER)
$f90029	Status Register (STAT)
$f9002f	Diagnostic LED Register
$fa0000-$f80040	MC68230 PIT registers
$fa0014	VMEbus Address Modifier Register (VAM)
$fc0000-$fdffff	ROM space

The following chart gives the base address for the dual-port IPC CSR registers and dual-port RAM to be used by the device drivers that communicate with the IPC. These addresses are in the VMEbus D16 space.

Hex Location	Device
$ffXX0000-$ffXXffff	Dual-port IPC CSR registers and dual-port RAM

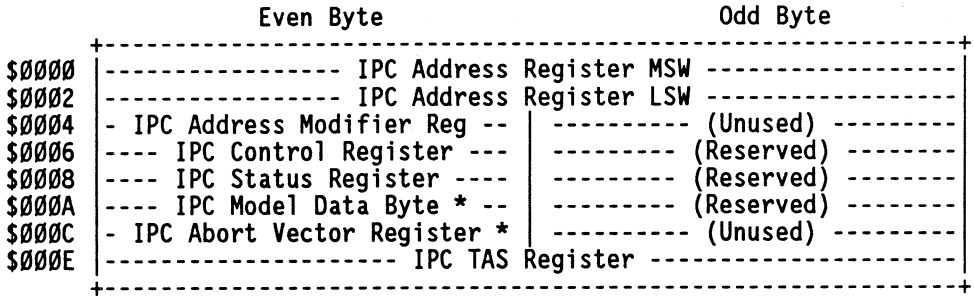
XX = Address bits set by switch settings on the MVME332XT.

4.7 IPC CONTROL/STATUS REGISTERS (IPC_CSR)

The registers in the IPC CSR space are accessible by both the host CPU and the local CPU are illustrated in Figure 4-2. For CSR commands the registers used by the host CPU are:

1. IPC TAS Register
2. IPC Address Register
3. IPC Status Register
4. IPC Control Register

For Buffered Pipe Protocol commands, the only register used by the host CPU is the IPC Control Register. Figure 4-2 illustrates the offsets and sizes for each register in the dual-port IPC CSR Registers.



* Unused on VMEbus designs

FIGURE 4-2. DUAL-PORT IPC REGISTER (IPC_CSR)

4.7.1 IPC_CSR Register Descriptions

IPC Address Register

A 32-bit register in which the host CPU loads the address of the channel header structure prior to issuing a "create channel" command.

IPC Address Modifier Register

An 8-bit register in which the host CPU loads the address modifier used by the IPC to access the channel header structure for a "create channel" command. This register must be loaded with the proper address modifier prior to any data transfer to or from host memory.

IPC Control Register

This register contains several bits used to control operation of the IPC. The busy bit indicates to the host CPU that the IPC is not ready to accept CSR commands yet. The IPC turns this bit off after completing its power-up initialization sequence. The reset bit forces the IPC into a reset state. The only way out of the reset state is for the host to clear the bit. The attention bit is set by the host CPU to interrupt the IPC when there is a command packet pending on the BPP command channel.

Figure 4-3 illustrates the bit definitions of the Control Register in the IPC_CSR space. Busy is used during the reset sequence to indicate to the host system that the MVME332XT has completed its initialization and is ready for operation. This bit is only valid when the MVME332XT is not asserting SYSFAIL*.

The reset IPC bit can be used by the host to reset the MVME332XT. To do so, the host sets this bit, waits for 500ms, and then clears the bit.

The attention bit is used by the host to generate interrupts to the MVME332XT firmware to indicate a command needs processing. The bit will be reset by the MVME332XT and should be viewed as write only by the host.

The last bit defined in the Control Register is the inhibit SYSFAIL* bit. This bit is used by the host to disable the MVME332XT from driving the VMEbus SYSFAIL* signal. The host can use this feature to aid in locating a failed module which is asserting SYSFAIL*.

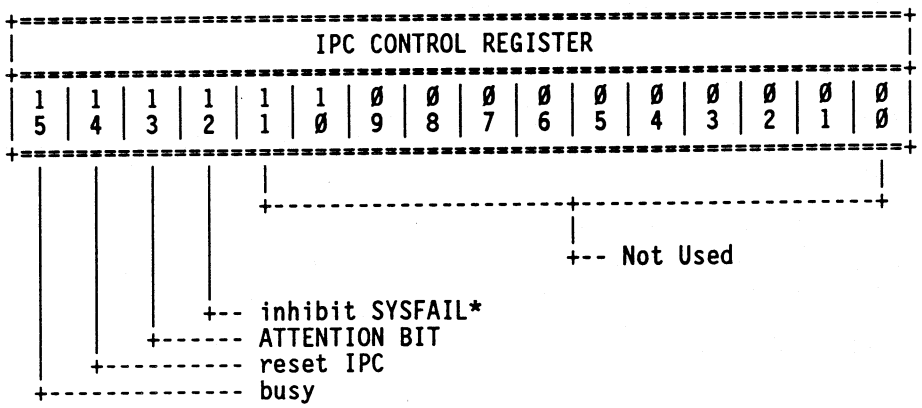


FIGURE 4-3. CONTROL REGISTER BIT DEFINITIONS

IPC Status Register

The IPC returns status to the host CPU for CSR commands in this register.

The IPC Status Register bit definitions are illustrated in Figure 4-4. The next section describes possible errors that the IPC can report to the host CPU through the IPC status register. These errors occur either as the IPC comes up from reset or during the execution of a CSR command.

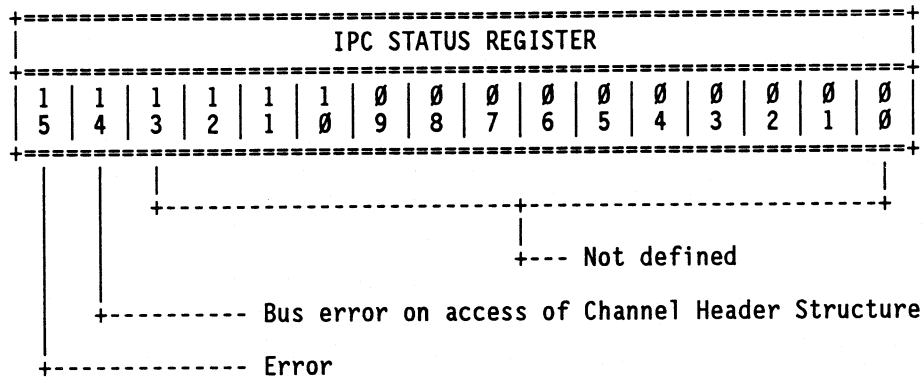


FIGURE 4-4. STATUS REGISTER BIT DEFINITIONS

IPC Model Data Byte

Not used on the MVME332XT. (Always read as a zero.)

IPC Abort Vector Register

The host CPU loads the vector number to be used by the IPC when it interrupts the host CPU on an error condition.

IPC TAS Register

The TAS or Test And Set register is used by the host CPU to send CSR commands to the IPC. It is used mainly to create one or more BPP pipes so the host can queue up commands to and receive status back from the IPC.

The CSR Command Register bit definitions are illustrated in Figure 4-5. The first four bits are used as handshake flags for CSR commands.

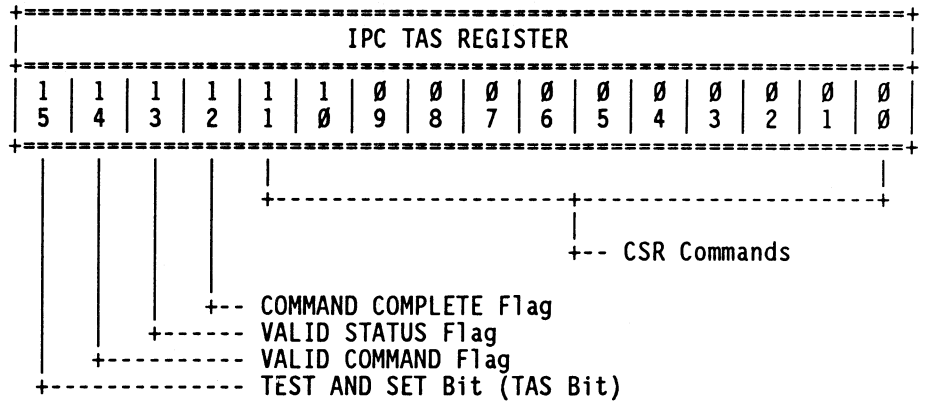


FIGURE 4-5. CSR COMMAND INTERFACE FLAGS IN TAS REGISTER

4.7.1.1 CSR Commands Possible Errors

Table 4-2 details the possible errors for the CSR commands.

TABLE 4-2. CSR COMMAND POSSIBLE ERRORS

CSR STATUS	BIT	DESCRIPTION
$8000	15	Error - an error occurred during a CSR command sequence.
$4000	14	Bad Channel Header Address - either the channel header address or address modifier was bad. The IPC could not read the channel header structure.
$FFFF	all	This status is the result of one of the following conditions: A. The IPC could not create a channel because there are no more free channels in its local queue. B. The IPC received an invalid CSR command. C. The IPC received a command to delete a channel that does not exist. D. The IPC received a command to delete a channel that has already been deleted.

CHAPTER 5
FIRMWARE ARCHITECTURE

5.1 MVME332XT START-UP SEQUENCE

Upon system start-up or system reset, a confidence test is performed to verify the basic functionality of the main hardware components of the MVME332XT. This includes the CPU, ROM, local CSRs, local RAM, dual-port RAM, MC6823Ø PIT, and four SIO MK68564s. If no errors are detected during this testing, control is passed to the firmware. If an error is detected, an infinite loop is entered and the firmware is never executed. Information indicating that a confidence test failure occurred and the progress of the test when the error was detected can be obtained by the Host by examining the Composite Status Word (CSW) of the Confidence Test Descriptor residing at the offset Øx1Ø of the dual-port RAM. The most significant byte will contain the test number of the test that failed and the least significant byte will contain a progress code indicating the last segment of the test to complete successfully. Refer to Appendix C for a table of these values.

NOTE: The confidence test does support Warm Start after power-on reset and will skip the entire confidence test (if the warm-start pattern written to memory on the successful completion of the power-on test is intact and readable) when subsequent resets are performed.

5.2 FIRMWARE OPERATION

The firmware architecture for the MVME332XT IPC consists of a realtime multi-tasking kernel and several individual programs called processes that control various operations of the MVME332XT. This kernel is a proprietary design of the Austin Design Center that provides process creation, scheduling, resource protection, and inter-process communications in the firmware environment. All of the firmware has been developed in "C" running under SYSTEM V/68 and is resident in EPROMs on the MVME332XT.

Figure 5-1 illustrates all processes that always exist in the firmware in either of two states, sleeping or running. Note that there are four processes per physical device, the GATE, CTL, OCP, and ICP processes. These processes share a data structure that is private to a device. All processes of the same type (such as GATE Ø to GATE 8) share the same code, but work on different data structures. A process goes to sleep when it has to wait for a resource to be available, or its time slice expires, or it voluntarily gives up the CPU since it has nothing else to do. A process is scheduled to run by the kernel when a resource becomes available or it has a new time slice to run.

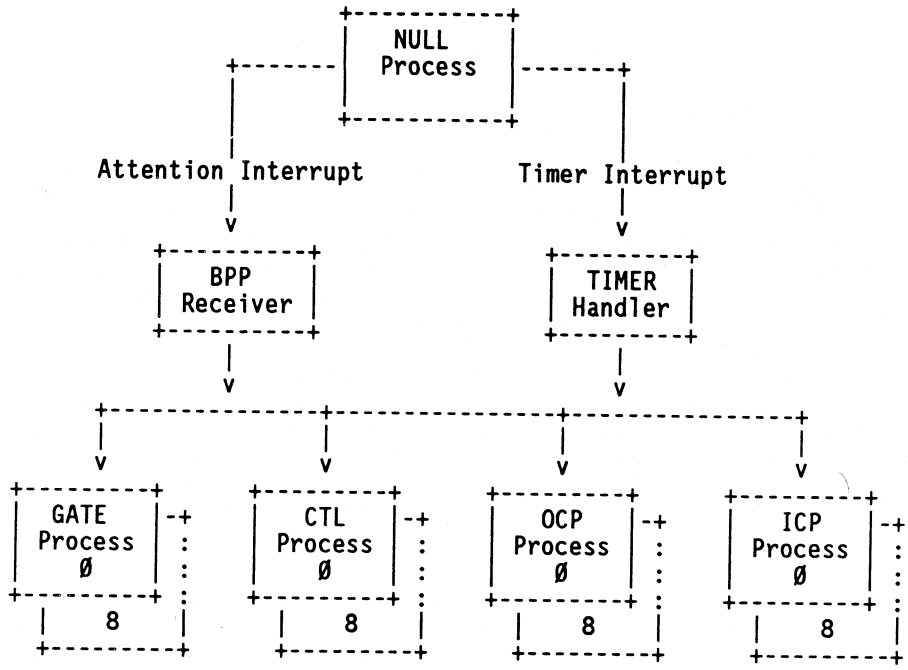


FIGURE 5-1. FIRMWARE PROCESSES

A general understanding of the firmware operation and its interaction with the kernel can be gained by following the flow of a typical "open" command.

1. In the idle state, the special kernel process "NULL" is active waiting for an event.
2. The host queues an appropriate command packet into the command pipe of the channel established from "create channel" command (CSR command), then signals the IPC by setting the attention bit in the MVME332XT's IPC_CSR.
3. The firmware BPP Receiver is activated to run in the context of the local attention bit interrupt. The BPP Receiver dequeues the command packet from the channel, parses it, then queues it to the appropriate process' command queue, and finally wakes the process up for execution. For the "open" commands, the command is queued in the command queue of the GATE process.
4. The GATE process is activated upon receipt of the command. It further parses the command and allocates resources.

5. The GATE process returns status and signals the host with an interrupt when it completes the open function, then goes back to sleep waiting for the next command.
6. The firmware returns to the idle state.

With the firmware kernel, it is possible to overlap many of the functions cited in this example which improves board efficiency and performance.

The processes included in the firmware are:

NULL Process

This process performs initialization tasks at startup and then is active in the idle state as a "place holder" waiting for an event.

BPP Receiver

The Buffer Pipe Protocol (BBP) Receiver runs in the context of the attention interrupt. Its function is to receive a packet in a BPP channel and then dispatch it to the appropriate process.

TIMER Handler

The Timer Handler runs in the context of the timer interrupt. Its basic function is management of all processes in the system, such as scheduling a process to run.

GATE Process

The GATE process handles open and close commands. The main function of this process is to synchronize the open and close requests, to prevent opening a device, and closing a device at the same time.

CTL Process

The CTL process handles all control functions, such as changing a baud rate, enable handshaking, get or set the current configuration, or downloading.

OCP Process

The Output Character Processing (OCP) process performs character translation and mapping for all output characters from the host buffer to the device.

ICP Process

The Input Character Processing (ICP) process performs character echoing, erasing, mapping, translating, signaling and escape

sequencing for all input characters from the devices to the host buffer.

5.3 WRITE/READ OPERATION

Figure 5-2 illustrates the overall architecture of the read and write operations. Associated with each device are two processes which are awakened to run by the Timer to perform character processing for each direction. The Timer runs at the clock interrupt level, checking each ring buffer, then awaking the associated process if the ring is not empty.

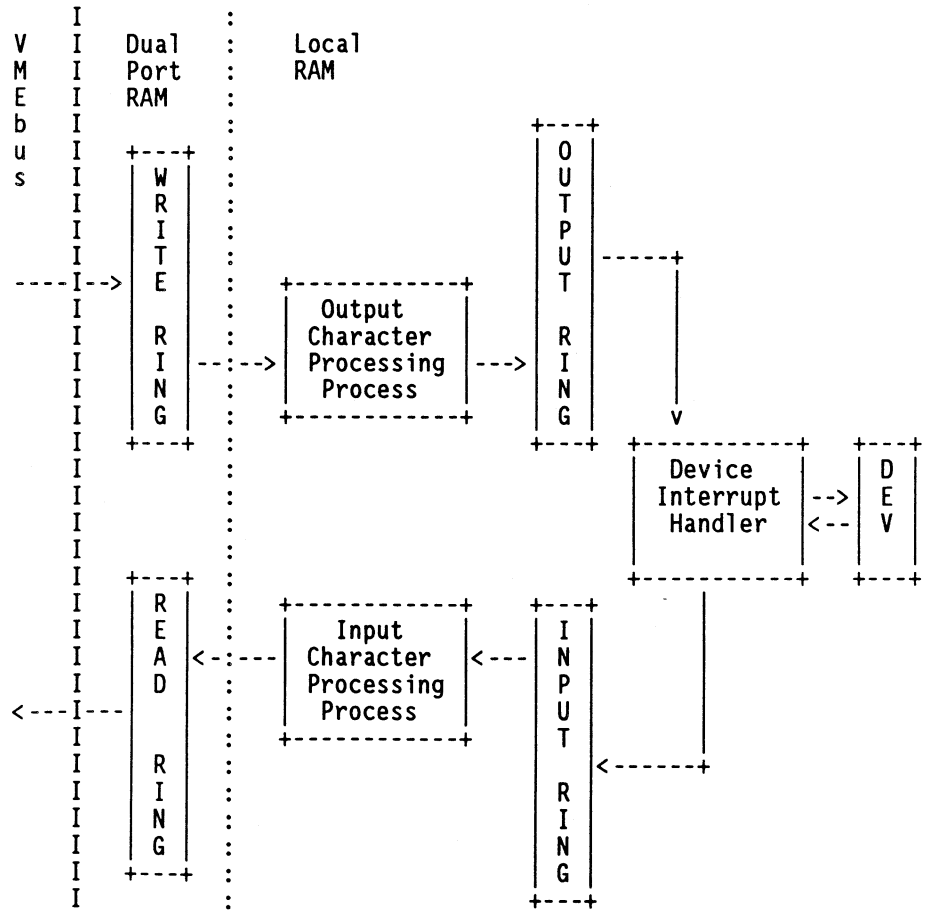


FIGURE 5-2. WRITE/READ OPERATION

5.3.1 Write Operation

In the Write Operation, the host writes all characters into the WRITE ring-buffer residing in the board's dual-port RAM. The Timer detects this situation and wakes up the OCP process which in turn moves those characters to the OUTPUT ring ready for the Device Interrupt Handler to output to the device.

If the WRITE ring-buffer is full because the firmware does not consume fast enough or the output is suspended, the host should wait for spaces on the WRITE ring by sending a packet to the firmware and then waiting for the packet to be returned. This packet will be returned to the host by the OCP process when the number of characters in the WRITE ring drops below a threshold level, referred to as the Low Water Mark.

The Low Water Mark detection will reduce a significant amount of packets traveling between the host and the firmware when the firmware is heavily loaded or the the OUTPUT ring reaches the High Water Mark.

5.3.2 Read Operation

In this operation, the Device Interrupt Handler gets a character from the device and puts it into the INPUT ring (or RAW queue). The Timer detects that there is a character in the INPUT ring and then wakes the ICP process up.

Upon awakening, the ICP process gets a character from the INPUT ring, performs the necessary translation or action, then puts it into the READ ring in the form of a frame. Each frame can be viewed as a part of a complete line terminated by a delimiter. If a line is too long (longer than a frame can hold, 255 characters), it will be broken down into many frames.

By using the frame format, the firmware can process the remaining characters in the ring before the host needs it. This also reduces the number of packets and increases the overlap operation. The detail of the frame will be discussed in the next sections.

If the READ ring is empty because the host consumes faster than the firmware can produce, the host should wait for an available frame by sending a packet to the firmware, and then waiting for it to be returned. This packet will be returned by the ICP process, if there is at least a complete line in the READ ring. A complete line is terminated by a delimiter or timeout depending on what mode is set.

5.4 CONTROL OPERATION

As soon as the host sets the attention bit in the IPC_CSR, the BPP_receiver is activated upon the local attention interrupt. The BPP_receiver dequeues the packet from the channel command pipe and hands it to the CTL process. It then notifies the CTL process by calling a kernel's signal function to make it runable in the firmware.

When the CTL process has a chance to run, it further passes its sub-command to perform a specific function such as setting or getting current configuration, changing baud rate, downloading code, or sending a break sequence. Some sub-commands require parameters or more data to work with. This data can be placed in the packet or somewhere in the dual-port RAM.

Finally, the CTL process returns the packet to the host and then looks at its command queue for any pending command. If the queue is empty, it goes to sleep waiting for the next request.

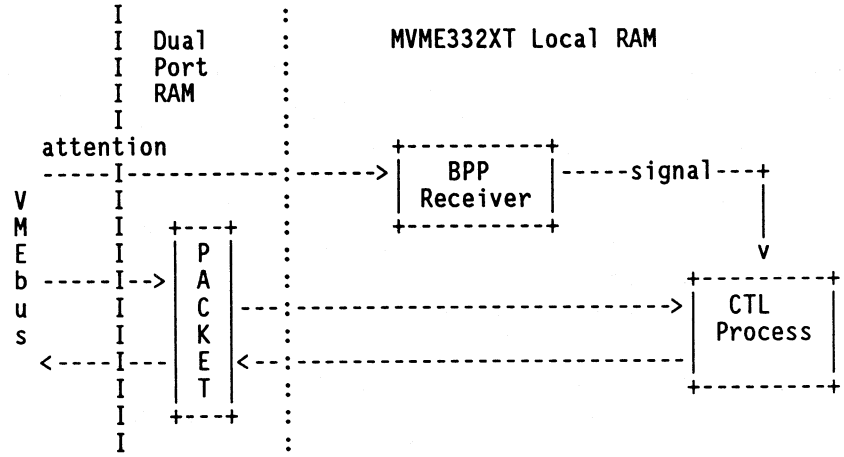


FIGURE 5-3. CONTROL OPERATION

5.5 GATE OPERATION

This operation is much the same as the control operation mentioned earlier, excepts that the GATE process is awakened to run if the packet is either a OPEN or a CLOSE packet.

Upon receiving the OPEN packet, the GATE process configures the device, asserts all the device's modem signals such as DTR and RTS,

and then returns the packet back to the host with the current status of the modem input signal DCD (Data Carrier Detect). The host must decide whether to wait for DCD or not, depending on the process mode.

When the GATE process receives a CLOSE packet, it will negate all modem control signals, disable the device to prevent further access, and then return the packet back to the host.

5.6 DUAL-PORT MEMORY

The MVME332XT's dual-port memory provides a communication mechanism between the firmware and the host. This is intented to reduce the number of bus accesses which may sacrifice the firmware performance, since the local CPU has to wait for the bus arbitration to complete before acknowledging any device interrupts.

The current implementation of the firmware requires that all communication data structures (including channels, envelopes, packets, and buffers) reside in this space, otherwise a fatal condition may occur because of misinterpretation. Since the local address of the dual-port RAM and the address viewed from the bus are different, the firmware must translate a pointer specified in a channel link list to the local accessible address. This is done by adding its lower word (bits Ø to 15) to the base address of the memory and ignoring the upper word (bits 16 to 31). Therefore, a pointer pointing to a location outside of the dual-port memory will be treated as the one that points to the inside. The firmware makes no attempt to check whether a pointer is inbound or not, since that will degrade its performance significantly.

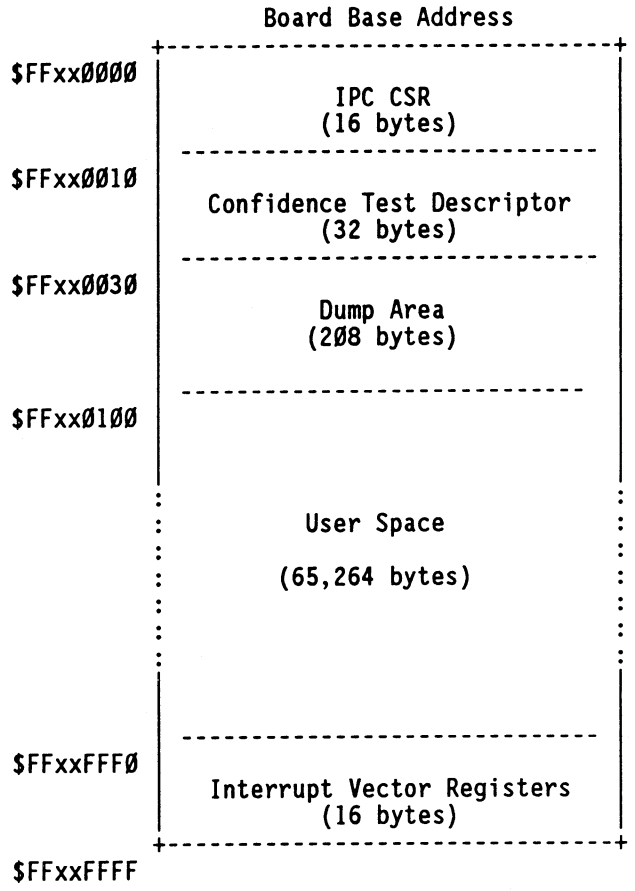
Even though the size of the dual-port RAM is 64K bytes, a portion of it is reserved for the hardware registers (IPC and Vector registers). Another portion is used by the firmware to display the information about the confidence test and crash condition (confidence test descriptor and dump area).

Figure 5-4 illustrates the spaces reserved for the firmware and the hardware. The space starting at offset $Ø1ØØ to the offset $FFFØ is available to the host for its data structures.

The memory map spaces for the firmware and hardware are described as follows.

IPC_CSR

This space is reserved for the IPC control/status registers as described in the previous section.



Where: xx is configurable in 8-position switch.

FIGURE 5-4. DUAL-PORT MEMORY MAP

Confidence Test Descriptor

On the power-up reset, the firmware executes a series of tests, so called confidence test, to make sure the hardware is in a good condition to operate. The information about each individual test will be placed in this block so that the host can obtain the detail of the failure when the test failed. The next section will discuss the detail of this field.

Dump Area

This area is implemented as a circular buffer. It is used to dump a message, in English text, so that a user can use an off-board debugger's memory display command to examine the reason why a confidence test failed or a crash in the firmware. The at sign (@) in the buffer indicates a starting point to read and then wraps around to the top if necessary.

User Area

This space is free for the host to use for any purpose. It is initialized to zero by the firmware upon a system reset. Typical use of this area is for character buffers, IPC structures, and download buffers.

Interrupt Vector Registers

This space is used as a set of eight hardware registers. Each register is a word in length and contains a vector for an interrupt level with the level Ø at the offset $FFFØ and the level 7 at the offset $FFFE. Only the lower byte (bits Ø to 7) is significant, the upper byte is ignored. The contents of these registers are initialized to $ØF, which is a standard uninitialized vector until it is reconfigured by the INIT packet.

CHAPTER 6
HOST/IPC INTERFACE

6.1 GENERAL

The interface protocol used by a host CPU to communicate with an IPC includes a set of basic commands sent through registers in the IPC_CSR (IPC command/status register) map and a set of commands sent in packets via virtual channels consisting of a command pipe and a status pipe. The host CPU uses a CSR command to establish the virtual channels for host/IPC communications. After the host creates one or more virtual channels, all other commands are transmitted between the host CPU and the IPC on these channels.

A virtual channel consists of a channel header structure which defines the attributes of the channel and two pipes. On one pipe, the host CPU enqueues command packets for the IPC. On the other pipe, the IPC enqueues status packets for the host. A pipe is simply a queue or singly linked list with one CPU manipulating the head pointer and another CPU manipulating the tail pointer.

This interface is called a non-busy interface because, once the channels are established, the host CPU never finds the interface to be in a busy state. This means that the host never has to wait to send a command. It also means that the IPC never has to wait to return status. The only limiting factor is the amount of memory available to the host for queueing command packets. This generic interface was created with the intent that it should be identical for all VME IPCs developed by the Austin Design Center of Motorola's Microcomputer Division.

In the chapters that follow, all commands supported by all devices attached to the MVME332XT are described. In some cases the packets that are shown have been condensed to avoid unnecessary duplication. In these cases some fields of the packet are not shown. The numbers at the left of each line of a packet are byte offsets to the fields on that line. These numbers increment by 2, since each line in a packet is two bytes long. Where fields are missing, the offsets jump by more than two bytes. To see a complete packet with all fields, refer to the appendix. Fields are in the same relative location in the packet for all packets for all devices. Where noted, certain fields are used for different purposes by different devices (primarily in the opcode dependent command fields and the device dependent status fields).

Status bits or fields that are unique to one device or one command are described in the appropriate section on that device or command. Where the status fields are the same for several commands for the same device, one section describes those status fields in detail.

6.2 MVME332XT HOST INTERFACE

The host/IPC interface is actually separated into two distinct interface protocols. The first protocol utilizes the IPC TAS Register in the IPC Control/Status Register (IPC_CSR) to send commands from the host to the IPC. Commands sent through the IPC TAS Register are referred to in this document as CSR Commands. This is a Busy interface in the sense that only one I/O driver may send a command at any given time. The driver must be granted access to the CSR space by successfully setting the IPC TAS bit in the IPC TAS Register. If one driver already has ownership of the CSR space, a second driver trying to access the IPC must wait until the first driver relinquishes its ownership.

The second protocol utilizes dual-port memory space and the Attention Bit in the CSR IPC Control Register to send command and status packets between the host CPU and the IPC. This is referred to as the Buffered Pipe Protocol. This interface uses virtual channels consisting of singly linked lists to enqueue command packets and dequeue status packets. This protocol represents a Non-Busy interface. Three things are provided to ensure that this interface is never busy.

1. First, each driver has its own channel on which to send commands so that it never has to wait for another driver to release the channel.
2. Second, the driver can queue up additional commands without having to wait for the IPC to complete a previous command.
3. Third, the driver and the IPC may both access the channel simultaneously with no interlock required.

Each channel has one command pipe and one status pipe. Commands may be sent at any time by enqueueing a packet on the command pipe and setting the Attention Bit in the IPC Control Register. Several drivers may set the Attention Bit at the same time without first having to gain ownership of the CSR space. The IPC scans all channels whenever the Attention Bit is set so it is not important which driver sets the Attention Bit or how many drivers set it.

6.3 BASIC CSR COMMANDS

Initial communications between a host CPU driver and the IPC Control Software are performed using the IPC_CSR space registers. The "create_channel" CSR command is used to establish channels between the host CPU device driver and the IPC. After these channels are create, they are used by a driver to pass channel command packets to the IPC. Similarly, the IPC uses channels to return executed command packets (status packets) to the host CPU.

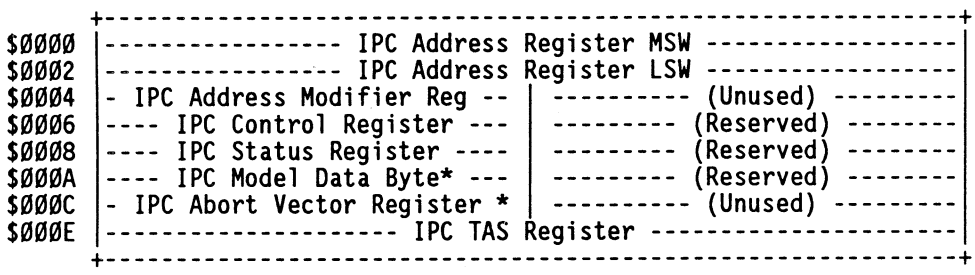
It is important to note the reset sequence for the MVME332XT. Before any communication to the MVME332XT can be initiated two conditions must be met. First, the system failure signal must be cleared by the MVME332XT. This signal is the VMEbus signal SYSFAIL*. This signal is asserted by the hardware at reset. This signal is cleared by the MVME332XT firmware once the on-board confidence test has been successfully completed. A failure in the confidence test causes the SYSFAIL* signal to remain asserted. To locate failed modules, the MVME332XT contains an inhibit SYSFAIL* bit in the CSR Control Register. The host CPU sets this bit to disable the MVME332XT from driving SYSFAIL*. This bit should be in the cleared state and no SYSFAIL* detected by the host CPU for normal operation. After the SYSFAIL* signal is no longer asserted, the busy bit (BSY) in the CSR Control Register must be cleared. The MVME332XT firmware clears this bit after it completes the necessary initialization sequence. Once these two conditions are met, the host CPU may initiate CSR commands and start normal operations with the MVME332XT.

Manual resets of the MVME332XT may be accomplished by using the IPC reset bit in the CSR Control Register. The host CPU should set this bit, wait for 500 milliseconds, and then clear the bit.

The following sub-sections deal with the protocol followed when issuing a CSR command and the specifics of the "create_channel" CSR command.

6.3.1 CSR Command Protocol

This section contains a discussion of the protocol followed by a driver when issuing a CSR command to the IPC. Before discussing the CSR command protocol, a basic familiarity with the hardware interface used to implement CSR commands is needed.



* Unused on VMEbus Designs.

FIGURE 6-1. IPC_CSR REGISTERS

This CSR space is accessible from the VMEbus as well as from local RAM space. This permits both the host CPU and the IPC CPU to read and write to any location in this CSR space. Figure 6-1 illustrates the IPC_CSR space as viewed by the IPC Control Software and the host CPU. The IPC TAS Register is a register in this space which is used when CSR commands are issued to the IPC. To notify the IPC of CSR commands the Attention Bit is set in the CSR Control Register as illustrated in Figure 6-2.

CSR Address Offset: $Ø6

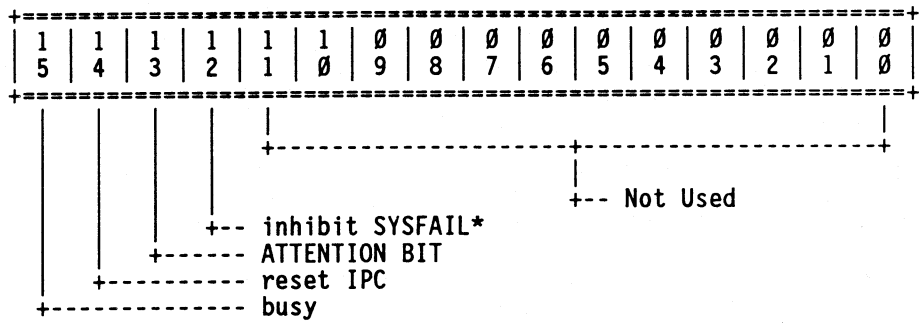


FIGURE 6-2. CSR CONTROL REGISTER BIT ASSIGNMENTS

The bit definitions for the IPC TAS Register are illustrated in Figure 6-3.

CSR Address Offset: $ØE

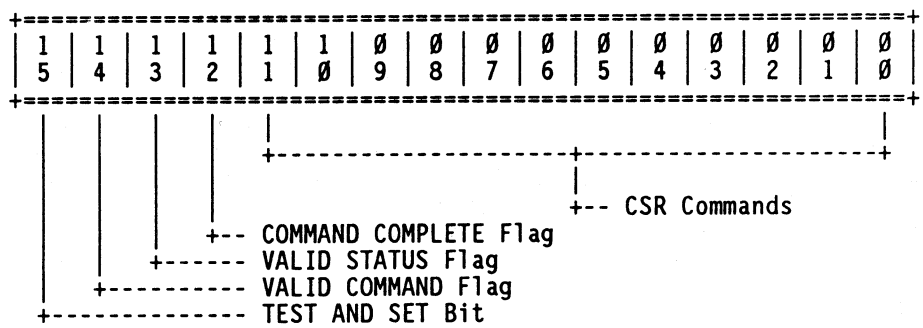


FIGURE 6-3. IPC_CSR TAS REGISTER

Table 6-1 lists the commands that are issued through the CSR space.

TABLE 6-1. CSR COMMANDS

Command Field	CSR Command
Ø	reserved
1	Create Channel
2	Delete Channel
3	unassigned
:	:
$FFF	unassigned

The protocol to be followed when issuing a CSR command to the IPC is as follows:

1. The host driver gains access to the IPC_CSR space by using the Test and Set instruction on the IPC TAS Register's TAS bit (bit 15). When TASing of this bit indicates that the bit was clear prior to the TAS operation, the driver has "possession" of the CSR space.
2. The driver writes the CSR command opcode in the IPC TAS Register (bits Ø through 11).
3. The driver writes associated parameters in other locations in the IPC_CSR space (e.g., IPC Address Register, IPC Address Modifier Register).
4. The driver sets the VALID COMMAND Flag in the IPC TAS Register (bit 14).
5. The driver generates an Attention Bit interrupt to the IPC by setting bit 13 of the IPC Control Register.

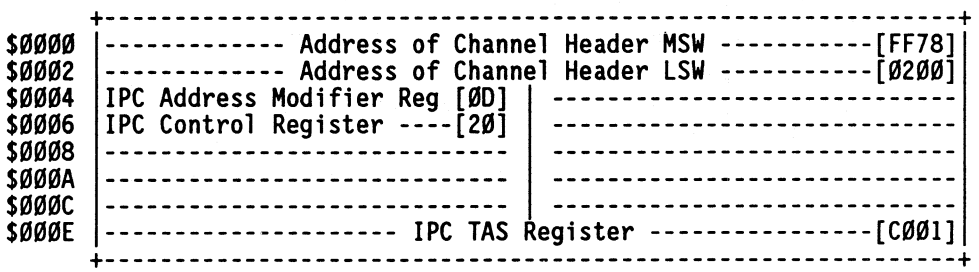


FIGURE 6-4. IPC_CSR REGISTERS WITH THE VALID COMMAND BIT SET

Figure 6-4 illustrates the contents of the CSR registers after the host CPU has set the Attention Bit. At this point the IPC Control Software does the following:

1. If the TAS bit is set, the VALID COMMAND flag is set, and the COMMAND COMPLETE flag (bit 12) is NOT set:
   a. IPC executes the CSR command.
   b. IPC places status concerning execution of the command into the IPC Status Register and then sets the VALID STATUS flag (bit 13 of the IPC Flag Word).
2. The IPC then also unconditionally polls any outstanding command pipes to see if there are any channel command packets to execute.

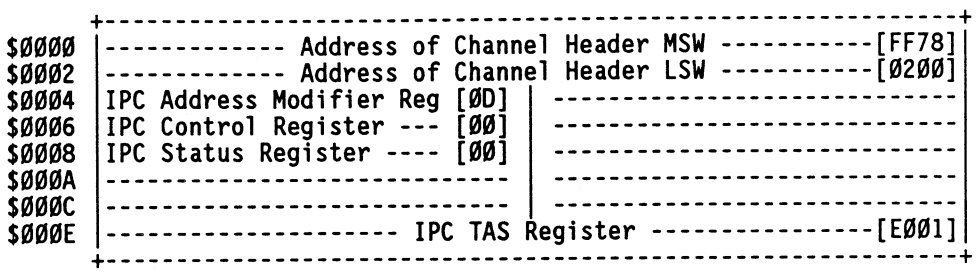


FIGURE 6-5. IPC_CSR REGISTERS WITH THE VALID STATUS BIT SET

Figure 6-5 illustrates the contents of the CSR registers after the IPC has set the Valid Status Bit in the CSR TAS Register. Now the driver does the following:

1. The driver polls the VALID STATUS Flag until set.
2. The driver then reads the IPC Status Register.
3. The driver then sets the COMMAND COMPLETE Flag (bit 12 of the IPC TAS Register). The driver must set the COMMAND COMPLETE bit even if status indicates an error.
4. The driver then generates another Attention Bit interrupt to the IPC.

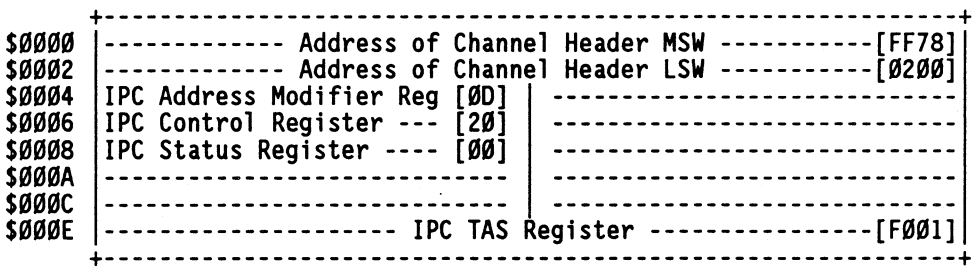


FIGURE 6-6. IPC_CSR REGISTERS WITH THE COMMAND COMPLETE BIT SET

Figure 6-6 illustrates the contents of the CSR registers after the host CPU has set the Attention Bit. The IPC responds to the interrupt as follows:

6

1. If the TAS bit set, the VALID COMMAND flag set, and the COMMAND COMPLETE flag set:

   a. IPC clears VALID COMMAND bit.

   b. IPC clears VALID STATUS bit.

   c. IPC clears COMMAND COMPLETE bit.

   d. IPC clears TAS bit (thus readying itself for another CSR command).

2. The IPC unconditionally polls any existing command pipes for any outstanding channel command packets. This is done every time the Attention Bit is set, since the IPC has no way to determine which CPU or which process set the Attention Bit.

6.3.2 Create Channel CSR Command Description

A channel consists of a command pipe and a status pipe. Channel command packets are shipped by a driver to the IPC via the command pipe. Channel status packets are returned to the driver via the status pipe. To send commands to an IPC, the host driver must create at least one channel. To create such a channel, a driver performs the following activities.

First, it builds the channel header structure in the MVME332XT dual-port memory. This process is described in more detail in the following section on creating a channel.

Next, the driver informs the IPC Control Software of the structure's existence. It does this by issuing a "create channel" CSR command to the IPC.

The parameters passed with this CSR command are the address of the channel header structure and the address modifier associated with the channel header structure. These are placed in the IPC Address Register and the IPC Address Modifier Register, respectively.

Upon successful execution of the CSR command, the IPC Control Software assigns a unique channel number to the newly created channel, writes the channel number to the channel header structure in dual-port memory, sets the valid channel flag in the channel header structure in dual-port memory, and adds the channel to its internally kept list of existing channels. If a channel is successfully created, a status of $0000 is returned in the CSR IPC Status Register.

If a channel cannot be created because there are no more free channels in the IPC, a status of $FFFF is returned in the CSR IPC Status Register. A status of $FFFF is also returned if the IPC receives an invalid CSR command.

If the IPC could not read the channel header structure in the dual-port memory, a status of $C000 is returned in the CSR Status Register. This can be caused by a bad channel header address, a bad address modifier, or some other problem with data transfers on the system bus. The IPC does not create a channel if it cannot successfully read the channel header structure in the host memory.

If the IPC could not write the channel number or the valid flag in the channel header structure in dual-port memory, a status of $8000 is returned in the CSR IPC Status Register. This indicates a problem with data transfers on the system bus. The IPC does not create a channel if it cannot successfully write to the channel header structure in the dual-port memory.

The command pipes of these channels are polled whenever an attention bit interrupt is detected by the IPC. A copy of the channel header structure including the channel's address modifier is kept in the IPC's local RAM space.

6.3.3 Delete Channel CSR Command Description

The "delete channel" command is provided to allow dynamic creation and deletion of channels if desired. The Buffered Pipe Protocol requires that the IPC scan all existing channels each time the Attention Bit is set. The IPC operates more efficiently in handling packets if the number of channels to be scanned is kept to a minimum.

To delete a channel, the driver informs the IPC Control Software of its intent to delete the channel. It does this by issuing a "delete channel" CSR command to the IPC.

The parameters passed with this CSR command are the address of the channel header structure and the address modifier associated with the channel header structure. These are placed in the IPC Address Register and the IPC Address Modifier Register, respectively.

Upon successful execution of the CSR command, the IPC Control Software removes the channel from its internally kept list of existing channels and returns a status of $ØØØØ in the CSR IPC Status Register, indicating that the command completed successfully.

If the channel being deleted is not a valid channel, a status of $FFFF is returned in the CSR IPC Status Register, indicating that the channel has already been deleted. A status of $FFFF is also returned if the IPC receives an invalid CSR command.

If the IPC could not read the channel header structure in the dual-port memory, a status of $CØØØ is returned in the CSR Status Register. In this case, the IPC cannot verify the existence of a valid channel. The channel is not deleted.

If the "delete channel" command completes successfully, the host driver is free to dispose of the channel header structure and the associated envelopes and packets and deallocate the memory used by these structures.

6.4 CREATING CHANNELS FOR HOST/IPC COMMUNICATIONS

The following section describes the process used by the host driver to create a BPP channel for communications with an IPC. It is possible for multiple host CPUs to create channels to the same IPC. The IPC_CSR registers are used for this operation so an interlock mechanism is required. Only one CPU at a time may perform the CSR command to create a channel.

6.4.1 Setting Up Queue Structures For 'create channel' Command

To begin communicating with the IPC, the device driver must set up a free packet queue and a free envelope queue. These are illustrated in Figure 6-7 below.

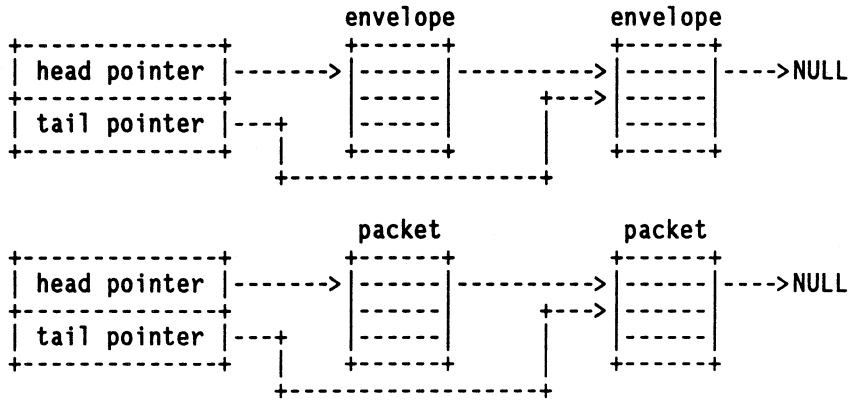


FIGURE 6-7. STRUCTURE FOR FREE ENVELOPE QUEUE AND FREE PACKET QUEUE

Both free packets and free envelopes use the structure illustrated above. A channel header structure must also be set up in memory. The channel header structure consists of a command pipe, a status pipe, and some additional parameters as represented in Figure 6-8.

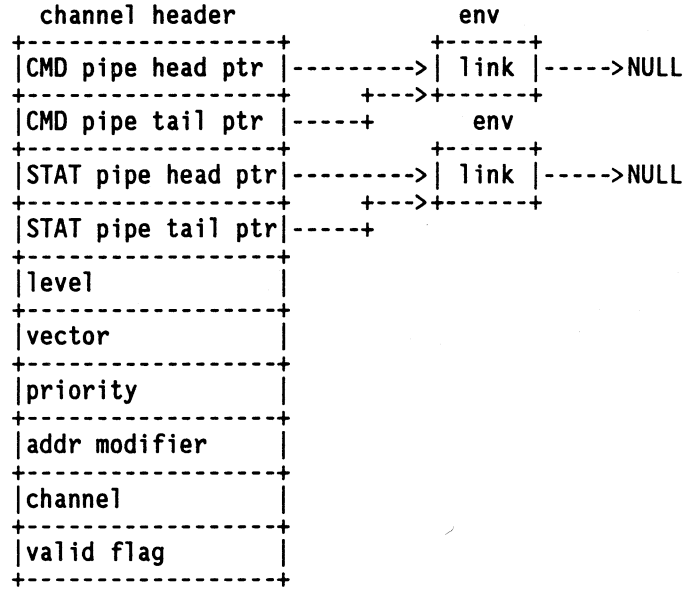


FIGURE 6-8. CHANNEL HEADER STRUCTURE WITH EMPTY CHANNELS

Figure 6-8 is a channel header structure with one envelope on the command pipe and one envelope on the status pipe. Both envelopes have their link fields pointing to null. The valid flag in the envelope has a value of zero, meaning that this is the last envelope in the pipe.

All of the fields in the channel header are filled in with the appropriate values except for the valid flag and the channel number. The channel number and valid flag are initialized by the IPC at the successful completion of a "create channel" command.

After assembling the channel header structure, the host CPU executes the "create channel" command. This is a CSR command. It requires that the host CPU set the TAS bit in the IPC TAS Register. Once the TAS bit is set, the host CPU proceeds with the CSR Command Protocol. This protocol is described in detail in the section entitled "CSR Command Protocol".

After checking the valid flag to ensure that no problem arose in creating the channel, the host CPU can begin sending commands to the IPC.

6.4.2 Enqueueing Packets On The Command Pipe

To send a packet the CPU enqueues another envelope and packet on the command pipe. New envelopes and packets are always enqueued at the tail of the command pipe. First, the CPU attaches a packet to the null envelope on the command pipe. The field labeled p.p. means packet pointer.

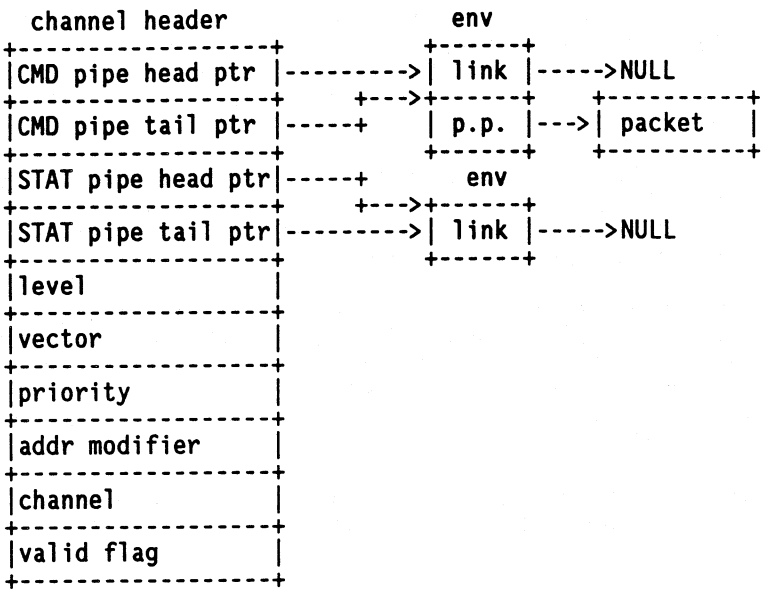


FIGURE 6-9. CHANNEL HEADER STRUCTURE WITH PACKET BEING ADDED TO THE COMMAND CHANNEL

Next, a new null envelope is enqueued on the command pipe. This is done by setting the address of the link field of the old null envelope to the address of a new envelope and then setting the link field of the new envelope to point to null. The valid flag of the new envelope must be set to zero to show that this is a null envelope. Then, the CMD pipe tail pointer is changed to point to the new null envelope.

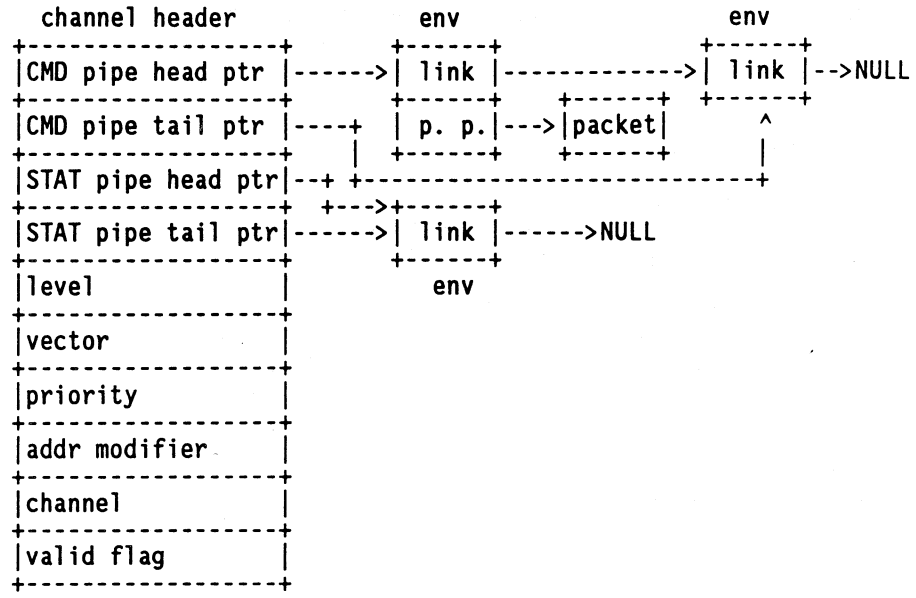


FIGURE 6-10. CHANNEL HEADER STRUCTURE WITH ONE PACKET ON THE COMMAND PIPE

All the applicable fields must be filled in either now or before the envelope and packet are enqueued on the command pipe with the exception of the valid flag field in the envelope. Setting the valid flag must be the last operation. After setting the valid flag, the host CPU sets the attention bit to notify the MVME332XT that a command is waiting for processing. Any number of commands may be enqueued on the command pipe, with either an Attention Bit interrupt to the MVME332XT after each command or one Attention Bit interrupt after a list of commands are enqueued. For each command queued on the command pipe, the MVME332XT notifies the host via a VMEbus interrupt when it completes a command.

6.4.3 Dequeueing Packets From The Status Pipe

When the IPC completes a command, it fills in all necessary fields to inform the host that the command has completed. It then enqueues the packet on the tail of the status pipe. Once the status packet is enqueued on the status pipe, the IPC generates an interrupt to the host CPU.

Illustrated below in Figure 6-11 is what the channel header and command/status pipes look like after a command completes and an interrupt to the host CPU is pending. When the host CPU receives the

interrupt it examines the status queue to see what happened. To do this the host CPU dequeues envelopes and packets from the head of the status pipe.

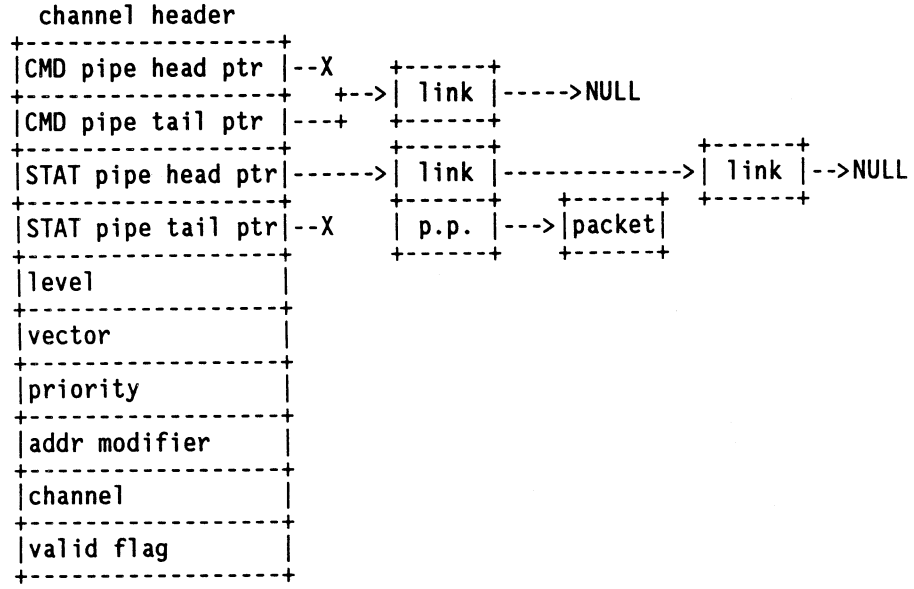


FIGURE 6-11. CHANNEL HEADER STRUCTURE WITH ONE PACKET ON THE STATUS PIPE

The command pipe head pointer and the status pipe tail pointer point to an X to indicate that they are stale and are not updated by the IPC. The command pipe head pointer and the status pipe tail pointer in system memory are not updated by the IPC and are never used once the "create channel" command is finished.

6.4.4 Buffered Pipe Protocol Summary

The IPC keeps its own copy of the command header and pipe structures in its local memory. After any commands are executed, the IPC copies of the command pipe head pointer and the status pipe tail pointer are different from what the host CPU has in host memory. The host CPU only updates the command pipe tail pointer and the status pipe head pointer as it enqueues and dequeues envelopes to and from the pipes. Also, the host CPU always enqueues envelopes/packets at the tail of the command pipe and dequeues envelopes/packets from the head of the status pipe.

6.5 IPC CHANNEL COMMUNICATIONS

This sub-section deals with driver/IPC communications via established channels. Communications between drivers and each IPC in a system are handled almost exclusively via command/status packets which are exchanged along these channels.

6.5.1 Establishing Driver/IPC Channel Communications

A driver begins channel communications by setting up at least three data structures in the MVME332XT's dual-port memory. These include a command pipe, status pipe, and a collection of free command/status packets. Command pipe and status pipe pairs are referred to as command channels.

Drivers pass command packets to the IPC via command pipes. The IPC returns processed command packets as status packets via status pipes.

The driver obtains command packets for transmission to the IPC from a collection of free command/status packets maintained by the driver. This pool of free packets is allocated by the driver in the dual-port memory.

6.5.2 Command Packet Queueing And Notification Procedure

When a driver has prepared a command/status packet for the IPC, i.e., has filled it with necessary command information, the driver enqueues that packet on a previously established command pipe. The driver then informs the IPC of the packet by setting the Attention Bit in the IPC Control Register. This causes a local interrupt to the IPC processor. The IPC then examines and processes the command packet.

6.5.3 Status Packet Queueing And Notification Procedure

When the IPC Control Software has finished processing a command/status packet and has put status into the packet, it enqueues the packet on a previously established status pipe. Once the packet is queued on the status pipe, the valid flag is set and the IPC then notifies the driver via a VMEbus interrupt. Upon notification (via the VMEbus interrupt), a driver reads all valid status packets queued on the status pipe. The host enqueues used command/status packets on its free command/status packet queue for later use.

If the interrupt level associated with the channel is zero (the vector and interrupt level fields are shown in the command channel structure), no interrupt is generated and the driver must poll the status channel while waiting for a packet to complete.

CHAPTER 7
DATA STRUCTURES

7.1 INTRODUCTION

This section describes and illustrates the necessary data structures, that drivers must establish in order to communicate (via the channel command packets) with the IPC. Most of these structures relate to IOCTL commands since such commands provide many various functions.

The MVME332XT firmware requires that all data (shared) structures must reside on its dual-port memory, otherwise, a catastrophic condition may occur as a result of misinterpretation in the firmware. For the sake of speed, the firmware ignores all checking on every access, but assumes that all communications are done on the dual-port memory.

Some structures are derived from the SYSTEM V/68 manual such as termio, termcb, and sgttyb, but some are specific for the MVME332XT only.

7.2 CHANNEL HEADER STRUCTURE

A single channel is composed of a command pipe and a status pipe. Each of these pipes is composed of a head pointer and a tail pointer that point to envelopes. Pipes must always contain at least one envelope (called the NULL envelope) in which case both the head and tail pointer point to the same envelope. Figure 7-1 illustrates the channel structure expected by the IPC.

The fields in the IPC channel header structure are described below.

Command Pipe

The Command Pipe structure consists of two 32-bit pointers. The first pointer points to the envelope at the head of the command pipe. This pointer is read once by the IPC when the channel is created. The IPC then updates a local copy of the head pointer as it removes packets from the command pipe. The second pointer points to the envelope at the tail of the command pipe which is ALWAYS a null envelope. This pointer is updated by the host CPU as it adds new packets to the tail of the command pipe. It is never read by the IPC. The IPC depends on ALWAYS finding a null envelope at the tail of the command pipe. Because neither processor accesses the pointer maintained by the other processor, a non-busy interface is achieved that requires no interlocking handshakes.

```
      +-------------------------------------------------------------+
$00   | ------------ command pipe head pointer msw ---------------  |
$02   | ------------ command pipe head pointer lsw ---------------  |
$04   | ------------ command pipe tail pointer msw ---------------  |
$06   | ------------ command pipe tail pointer lsw ---------------  |
$08   | ------------ status pipe head pointer msw ----------------  |
$0A   | ------------ status pipe head pointer lsw ----------------  |
$0C   | ------------ status pipe tail pointer msw ----------------  |
$0E   | ------------ status pipe tail pointer lsw ----------------  |
$10   | ----- interrupt level* -----  | ----- interrupt vector* ----  |
$12   | ----- channel priority* ----  | ----- address modifier -----  |
$14   | ----- channel number -------  | ---------- valid* ----------  |
$16   | --------- datasize* --------  | --------- reserved ---------  |
      +-------------------------------------------------------------+
       * Not used by the MVME332xt.

      struct channel {              /* C declaration */
            struct envelope  *command_pipe_head_ptr;
            struct envelope  *command_pipe_tail_ptr;
            struct envelope  *status_pipe_head_ptr;
            struct envelope  *status_pipe_tail_ptr;
            char              interrupt_level;
            char              interrupt_vector;
            char              channel_priority;
            char              address_modifier;
            char              channel_number;
            char              valid;
            char              datasize;
            char              reserved;
      } channel;
```

FIGURE 7-1. CHANNEL FORMAT

Status Pipe

The Status Pipe structure consists of two 32-bit pointers. The first pointer points to the envelope at the head of the status pipe. This pointer is updated by the host CPU as it removes packets from the status pipe. The second pointer points to the envelope at the tail of the status pipe which is ALWAYS a null envelope. This pointer is read once by the IPC when the channel is created. A local copy of this pointer is updated by the IPC as it adds new packets to the tail of the status pipe. It is never read by the host CPU. The host CPU depends on ALWAYS finding a null envelope at the tail of the status pipe. Because neither processor accesses the pointer maintained by the other processor, a non-busy interface is achieved that requires no interlocking handshakes.

Interrupt Level

This interrupt level is the level at which the IPC interrupts the host CPU when execution of a command packet is complete. If the interrupt level is zero, no interrupts are issued by the IPC for this channel and the host CPU must poll the status pipe to determine when a status packet is returned.

Interrupt Vector

This is the interrupt vector used by the IPC for all interrupts issued by the IPC to the host CPU for this channel.

Channel Priority

This priority is used by the IPC when it polls all active channels looking for valid command packets. The IPC polls all active channels each time it receives an Attention Bit Interrupt from any host CPU.

Address Modifier

This address modifier is used by the IPC to transfer all packets between host memory and IPC memory.

Channel Number

This is a unique channel number assigned by the IPC at the time a channel is created. The IPC writes this number to the channel header structure in host memory. The host CPU places this number in each packet it sends to the IPC to indicate the channel the command packet was sent on and the channel the status packet should be returned on.

Valid Flag

This flag is set by the IPC when the channel is created to indicate to the host that the IPC now recognizes this channel and is ready to accept command packets on it.

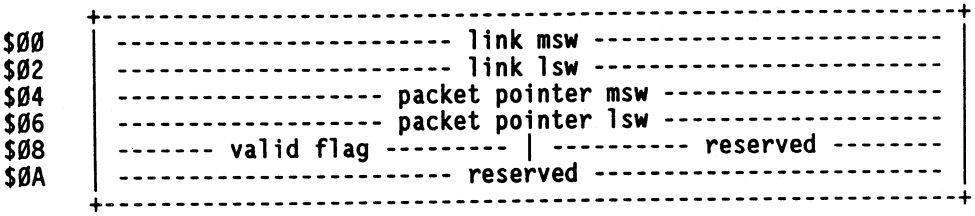
Datasize

Not currently implemented in the firmware, this field specifies the data bus width for transfers of BPP envelopes and packets.

Value	Data Bus Width
Ø	32-bit (default)
1	16-bit
2	32-bit

7.3 ENVELOPE STRUCTURE

System envelopes are kept in the MVME332XT's dual-port memory and have the structure illustrated in Figure 7-2.




```
struct envelope {                   /* C declaration */
        struct envelope  *link;     /* pointer to next envelope */
        struct packet    *packet_ptr;
        char              valid_flag;
        char              reserved[2];
} envelope;
```

FIGURE 7-2. ENVELOPE FORMAT

Each envelope contains the following fields:

Link

The link field points to the next envelope in the pipe.

Packet Pointer

This 32-bit field points to the body of the packet which contains the information specific to the command being issued. The packet may immediately follow the valid flag but is not required to. Envelopes and packets do not have to be allocated by the host as one contiguous block of memory since an envelope does not always point to the same packet. For efficiency in passing status packets back to the host, the packet pointer in the envelope is modified to point to the status packet being returned. This eliminates the need to move the entire contents of a packet to another place in global RAM.

Valid Flag

This 8-bit field is the last field in a packet to be modified. This field is eight bits long so the flag can be set in one indivisible operation. The function of the flag is to guarantee that the rest of the packet is valid before the IPC dequeues it. The IPC treats a packet as null while this field contains zero.

The reserved byte following the valid flag field should not be used for normal packet data.

The valid flag should only be set when a packet contains a valid command to be processed by an IPC. At all other times the valid flag is cleared. The host CPU should never enqueue a packet on a command pipe with the valid flag set unless ALL other data in the envelope and packet is valid.

Reserved

These reserved fields force the length of an envelope to a longword boundary. They are not used by the IPC, but simply guarantee that an envelope can be transferred as three longwords.

7.4 PACKET STRUCTURE

The IPC expects packet format to appear as illustrated in Figure 7-3. Blank fields are not used by the firmware, driver can use them for any purpose.

The following are descriptions of the packet fields.

Eyecatcher

This field consists of four "eyecatching" ASCII characters. Its purpose is to aid in the tracking of command/status packets in the system memory. It is used primarily during debugging.

Command Pipe Number and Status Pipe Number

The Command Pipe Number field contains the number of the particular command pipe which a command/status packet is sent on by the host. Similarly, the Status Pipe Number field contains the number of the particular status pipe which a command/status packet is sent on by the IPC.

The host CPU places $FF in the status pipe number field if the status packet is to be returned on the same channel used to send the command packet. The host CPU fills in the status pipe number if it wants status returned on a channel other than the one the command was sent over.

Command

The Command field specifies the command the MVME332XT is to execute. Commands are operating system dependent but tend to follow predictable patterns.

Command Dependent

This field is used in some commands but not all, refer to the specific command section for more detail.

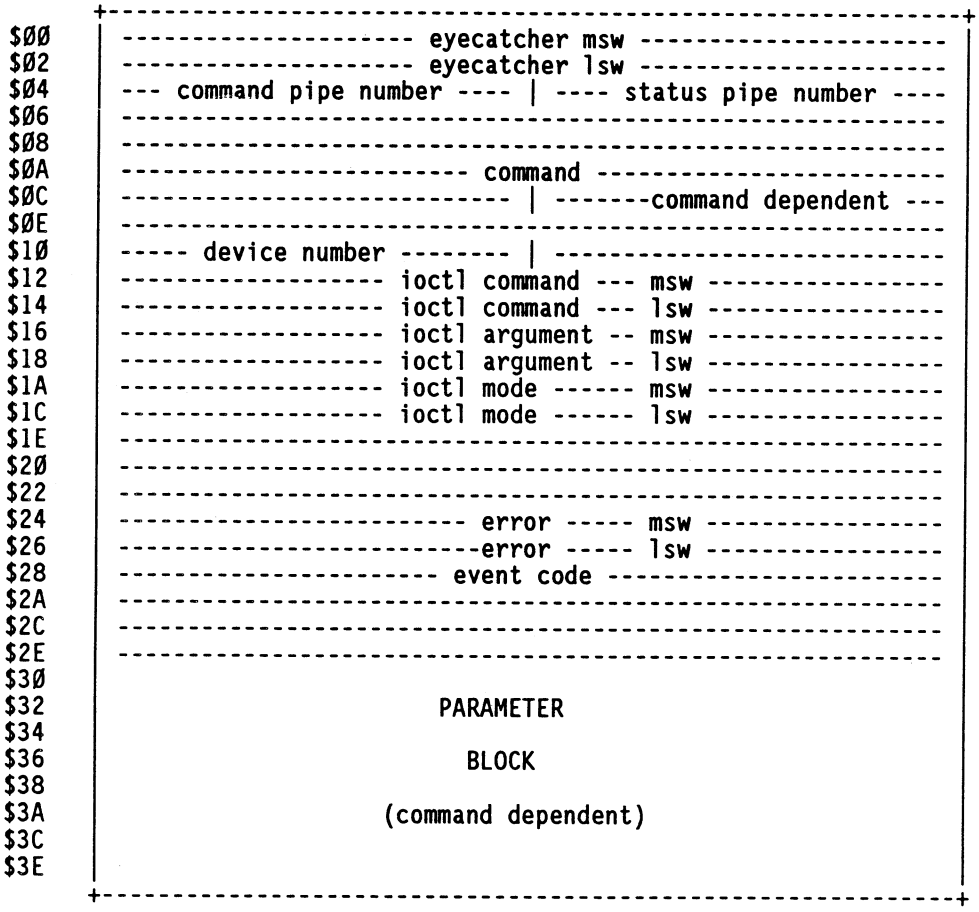
Device Number

This is the unit number for the peripheral device to be accessed by the host. Note that the firmware treats the printer device as an output only serial device, so there is no special interface for the printer device.

Ioctl Command, Argument, Mode

These fields are sub-commands and parameters for the Ioctl command. More detail will be discussed in the device specific section.

REQUIRED PARAMETERS:




```
struct packet {   /* C style declaration */
    char          eye_catcher[4];
    unsigned char command_pipe_number;
    unsigned char status_pipe_number;
    char          filler_0[4];
    short         command;
    char          filler_10[1];
    char          command_dependent;
    char          filler_11[2];
    unsigned char device_number;
```

FIGURE 7-3. PACKET FORMAT

```
        char            filler_2[1];
        long            ioctl_command;
        long            ioctl_argument;
        long            ioctl_mode;
        char            filler_3[6];
        long            error;
        short           event_code;
        char            filler_4[6];

        union {
                struct termio  tio;  /* termio structure */
                struct termcb  tcb;  /* termcb structure */
                struct sgttyb  sgt;  /* sgttyb structure */
                struct dl_info dl;   /* download info structure */
                long           param;/* general purpose parameter */
        } parameter_block;           /* ioctl_command dependent */

} packet;
```

FIGURE 7-3. PACKET FORMAT (cont.)

Table 7-1 provides a generic list of the commands supported by IPCs.

TABLE 7-1. IPC COMMAND SUMMARY

Command Code	Operation
$Ø	Init
$1	Read_Wakeup
$2	Write_Wakeup
$3	Open
$4	Ioctl
$5	Close
$6	Event

Error

This field contains an error code returned from the IPC upon completion of a command. A non-zero in this field indicates an error condition occurs during execution of the command. This field is cleared when the firmware receives a packet.

Table 7-2 provides a list of all the devices supported by the MVME332XT.

TABLE 7-2. DEVICE NUMBER ASSIGNMENT

Device Number	Physical Device
$Ø	Serial Port Ø
$1	Serial Port 1
$2	Serial Port 2
$3	Serial Port 3
$4	Serial Port 4
$5	Serial Port 5
$6	Serial Port 6
$7	Serial Port 7
$8	Printer Port

Event Code

This field is returned by the firmware and contains bits corresponding to events that occur on the firmware side. Refer to the EVENT packet format for more information about event code.

Parameter Block

The following fields contain parameters specific to the command being sent by the host to the IPC. These parameters may include a configuration data structure required by a command. These fields are not required for all commands. See the documentation on each specific command for information on the fields required by that command.

7.5 TERMIO STRUCTURE

This structure is used extensively in the firmware as well as the IOCTL commands (TCSETA, TCGETA, TCSETDF, TCGETDF) and the INIT command. It contains all the information necessary to configure a device such as baud rate, character size, parity, and translation characters. Figure 7-4 illustrates a format of a termio structure and a typical declaration of the termio structure in the "C" language style.

```
      +-------------------------------------------------------------+
$00   | ------------------- input option flags ------------------- |
$02   | -------------------output option flags -------------------- |
$04   | ------------------ control option flags ------------------- |
$06   | ------------------- local option flags ------------------- |
$08   | -- line discipline number -- | ---- interrupt character --- |
$0A   | ------ quit character ------ | ------- erase character ---- |
$0C   | ------ kill character ------ | -------- eof character ----- |
$0E   | ------ eol  character ------ | ---------- reserved -------- |
$10   | ----- switch character ----- | ---------- reserved -------- |
$12   +-------------------------------------------------------------+

      struct termio {
              unsigned short c_iflag;   /* input option flags */
              unsigned short c_oflag;   /* output option flags */
              unsigned short c_cflag;   /* control option flags */
              unsigned short c_lflag;   /* local option flags */
              char           c_line;    /* line discipline number */
              unsigned char  c_cc[NCC]; /* control characters */
      };
```

FIGURE 7-4. TERMIO STRUCTURE FORMAT

The following is a brief description of each field. Refer to the SYSTEM V/68 Programmer's Reference Manual or Appendix J for more details.

Input Option Flags (c_iflag)

This field consists of all options that affect the input characters. The firmware ICP process uses these flags to perform a specific action for every input character, such as map a CR (carriage-return) to a LF (line feed), strip a character to a 7-bit character, enable software handshaking (XON/XOFF), or send an event packet back to the host if a Break character is received. This field is ignored for the printer device since it is an output only device.

Output Option Flags (c_oflag)

This field is used by the firmware OCP process to perform a specific action for every output character, such as map a CR to LF or vice versa, and expand a tab character to a number of spaces. Note that the current implementation of the OCP does not support character delay, therefore, all output delay options specified in the Termio are ignored by the firmware. Only the TAB3 option is used to expand a tab character to eight blank spaces.

Control Option Flags (c_cflag)

This field contains all hardware options such as baud-rate, character size, number of stop bits, parity control, and modem control. The firmware CTL process uses this field to configure a device upon the host request.

Local Option Flags (c_lflag)

This field contains various options that affect input characters. It is used by the firmware ICP process to enable an echo mode, a signal mode, a translation mode, and a timeout mode for the ICP.

Line Discipline Number (c_line)

The Line Discipline Number field is used by the CTL process to switch all firmware processes (GATE, OCP, ICP, and CTL itself) to a different set of routines specified in the line discipline table. This is done so that they can perform a new functionality such as international character translation in the ISP (International Software Package).

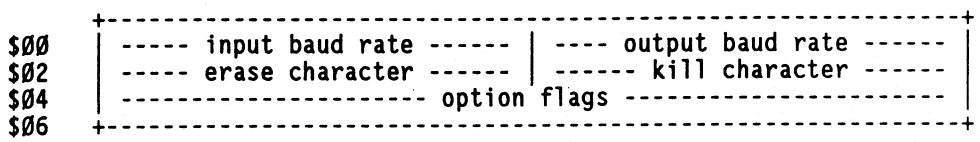
Control Character (c_cc[NCC])

These fields allows the host to change some control characters to be any character for different taste. Upon receiving an interrupt character or a quit character, the ICP process sends an event packet to the host to indicate that an special event has occurred. This is done so that the driver can notify the processes associated to a device.

The erase, kill, eof, and eol characters are used by the ICP process to erase or delete the current input line, or to build a complete line on the READ ring.

7.6 SGTTYB STRUCTURE

The Sgttyb structure is an earlier version of the termio structure existing in version 6 and 7 of UNIX or BSD. It is included in the firmware for data structure conversion and for backward compatibility only. Two ioctl commands (TIOGETP and TIOSETP) use this structure as a parameter block to instruct the firmware to converts it to termio internally.




```
struct sgttyb {
        char    sg_ispeed;      /* input baud rate */
        char    sg_ospeed;      /* output baud rate */
        char    sg_erase;       /* erase character */
        char    sg_kill;        /* kill character */
        short   sg_flags;       /* control option flags */
};
```

FIGURE 7-5. SGTTYB STRUCTURE FORMAT

Figure 7-5 illustrates the format of the sgttyb structure. For more information, refer to the appendix or Section 4 of the BSD UNIX Manual. The following is a brief description of each field.

Input And Output Baud-Rate (sg_ispeed, sg_ospeed)

These fields specify the baud rate separately for input and output, but the firmware ignores the output field since all devices support only one baud rate generator.

Erase/Kill Character (sg_erase, sg_kill)

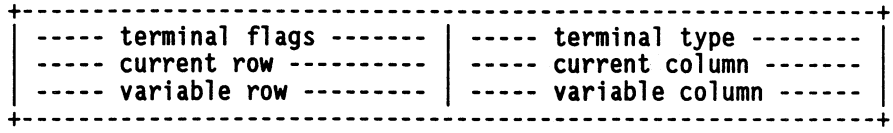
The erase and kill characters are used by the ICP process to erase or delete the current input line.

Control Option Flags (sg_flags)

This field contains some control options, such as delay on some special characters, expand tab to spaces, select parity mode, map CR to LF or vice versa, map upper case to lower case on input, and automatic flow control.

7.7 TERMCB STRUCTURE

This structure is used only in a few ioctl commands to get or set the current cursor address for virtual terminal support under the old version of UNIX. It is supported only for backward compatibility. Figure 7-6 illustrates the format of termcb structure and a typical declaration in the "C" language style. It contains row and column addresses of the terminal cursor.




```
struct termcb {
        char        st_flgs;      /* terminal flags */
        char        st_termt;     /* terminal type */
        char        st_crow;      /* current row */
        char        st_ccol;      /* current column */
        char        st_vrow;      /* variable row */
        char        st_lrow;      /* last row */
};
```

FIGURE 7-6. TERMCB STRUCTURE FORMAT

The following is a brief description of each field. At present, there is no reference available on its use, except in the "getty" utility in the SYSTEM V/68 Administrator's Reference Manual.

Terminal Flags (st_flgs)

This field specifies options for a specific terminal type, such as perform special new line, auto new line at column 80, perform special action on the last column of last row, echo terminal cursor control, and suppress sending escape sequence to the host. A zero in this field instructs the firmware to use the default configuration, which is set up in the terminal specific driver in the firmware.

Terminal Type (st_termt)

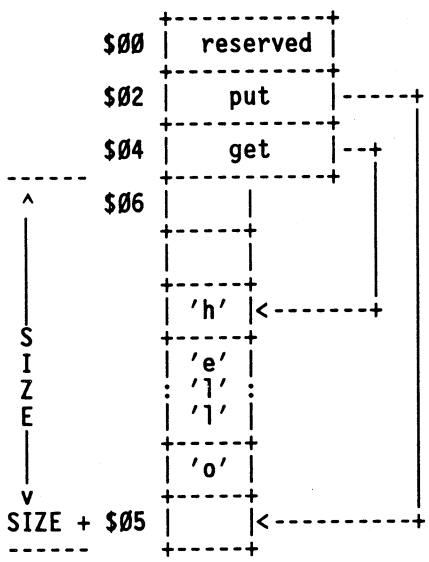
This field specifies what type of terminal the data structure is for. These are TEC Scope, DEC vt61, DEC vt100, TEK 4023, TTY Mod 40/1, HP 45, and TTY Mod 40/2B.

Rows And Columns

These fields specify the terminal cursor address for the row and column number.

7.8 RING STRUCTURE

There are four ring structures for each logical device, READ, WRITE, OUTPUT, and INPUT ring. The first two are shared between the host and the firmware and the second two are locally used by the firmware. The READ/WRITE rings are established in the dual-port memory by the driver at initialization time. The OUTPUT/INPUT ring structures are statically allocated on the local memory in the firmware.




```
struct ring {                   /* declaration in "C" */
        unsigned short reserved;  /* reserved for future */
        unsigned short put;       /* put index */
        unsigned short get;       /* get index */
        char           data[SIZE];/* data buffer */
};
```

FIGURE 7-7. RING STRUCTURE FORMAT

Figure 7-7 represents the ring structure format. Each ring can be configured with a different size up to 64K bytes, but it must be a power of two and must start on an EVEN word boundary. The firmware will return an error if it does not meet their requirements.

A ring consists of a "put" index and a "get" index into the ring data array. Both are a short type (two bytes), therefore, the maximum

size of a ring is 64K bytes. A ring is considered as an EMPTY ring when the "get" index is equal to the "put" index, and a FULL ring when the "put" is less than the "get" by one modulo the ring size.

In the case of write, the host uses the "put" index to put its data into the ring and the firmware uses the "get" index to get that data out. In the case of read, the firmware should use the "put" index and the host should use the "get" index. Such a protocol does not require any interlock mechanism since variables are not shared. The "put" and "get" indexes should be ANDed with (SIZE-1) prior to accessing the ring and incrementing the index.

Figure 7-7 illustrates a "C" style declaration of a ring structure. Note that the SIZE is configurable in the init command.

CAUTION: Before updating any index, one side should make a local copy, change it, then write it back to the ring.

Once again, the firmware requires all rings to reside on its dual-port memory.

Examples: 1. To put a character into the ring buffer:

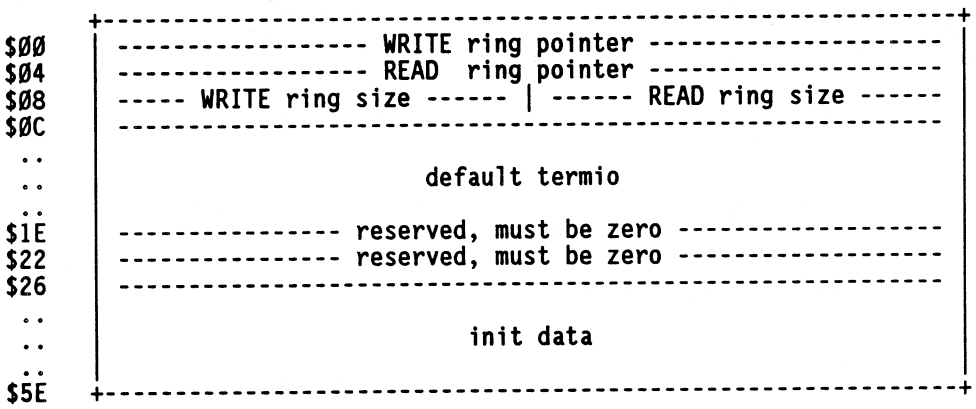
```
struct ring * ring; char c;
ring->data[ring->put++&(SIZE-1)] = c;
```

2. To get a character from the ring buffer:

```
c = ring->data[ring->get++&(SIZE-1)];
```

7.9 INIT_INFO STRUCTURE

This structure is used only one time in the driver initialization routine to establish a device's global parameters for the firmware. It contains information about where the rings are, how big for each ring, what device default configuration, etc. The INIT packet requires nine of these structures, one per logical device.




```
struct init_info {
        struct ring     *write_ring_ptr; /* WRITE ring pointer */
        struct ring     *read_ring_ptr;  /* READ  ring pointer */
        unsigned short write_ring_size;  /* WRITE ring size */
        unsigned short read_ring_size;   /* READ  ring size */
        struct termio  default_termio;   /* default termio */
        unsigned long  reserved[2];      /* reserved, MBZ */
        char           init_data[56];    /* init data */
};
```

FIGURE 7-8. INIT_INFO STRUCTURE FORMAT

Figure 7-8 illustrates the format of init_info structure and a typical declaration of the init_info structure in the "C" language style. The following is a brief description of each field. Refer to the section that describes the init packet for more information.

WRITE Ring Pointer

This field contains a pointer to a device's WRITE ring structure residing in the dual-port memory. The firmware's OCP process will use it to know where to get the data from.

READ Ring Pointer

Same as the previous item, this field contains a pointer to a READ ring structure so that the firmware's ICP process knows where to put its data.

WRITE Ring Size

The size of the WRITE ring can be specified here. It must be non-zero and a power of 2 value.

READ Ring Size

Same as the previous field but for the READ ring.

Default Termio

This field sets up a default configuration for a device such as baud rate, character size, or parity. The firmware's GATE process will use it to initialize the device when it receives an OPEN packet.

Reserved

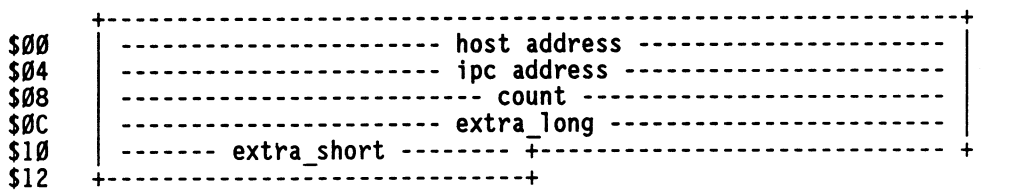
These fields must be reserved for Motorola and cleared to zero, since future downloadable line disciplines may use them to perform some special actions such as ISP dual-language translation.

Init Data

These fields are currently not used by the firmware, but they are available for future expansion. The firmware will copy them to the associated device's internal structure so that a downloadable line discipline is able to use them as default values. This allows a driver initialization routine to pass any information to any downloadable line discipline.

7.10 DL_INFO STRUCTURE

This structure is used to download a program, a data structure, a line discipline, or a new firmware to the MVME332XT local memory. It is also used to instruct the firmware's CTL process to execute a specific function address residing in the firmware side.




```
struct dl_info {
      unsigned long  host_addr;   /* address on the host side */
      unsigned long  ipc_addr;    /* address on the IPC side */
      unsigned long  count;       /* number of byte to transfer */
      unsigned long  extra_long;  /* general purpose use */
      unsigned short extra_short; /* general purpose use */
};
```

FIGURE 7-9. DL_INFO STRUCTURE FORMAT

Figure 7-9 illustrates the format of the dl_info structure and its typical declaration in the "C" language style. The following is a description of each field. Refer to the section that describes the IOCTL packet for more information.

Host Address (host_addr)

This field specifies the source or destination address on the host side. The firmware uses this information to get the host's data or to supply its data to the host. The current implementation of the firmware requires that the address in this field has to be on its dual-port memory.

IPC Address (ipc_addr)

This field specifies the source or destination address on the IPC side. The firmware uses this information to get its data for the host or to store the data supplied by host. The firmware will make sure this address is in the downloadable range, otherwise, it will ignore the request and return an error code in the packet.

Count (count)

This field specifies the number of bytes to transfer. The

firmware will make sure that it is fitted into the downloadable area, otherwise, the request will be ignored.

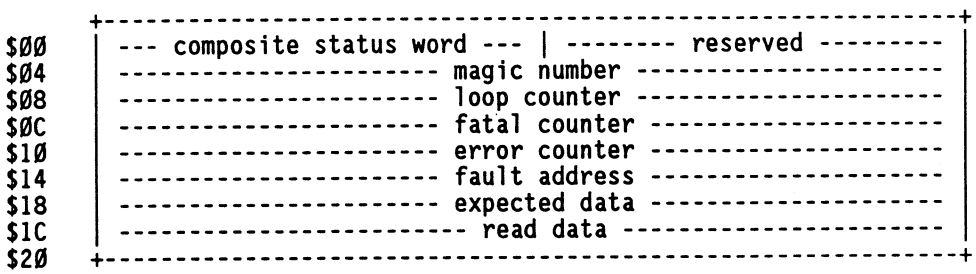
Extra_Long And Extra_Short

These two fields are available for general purpose use. They may be needed in few ioctl commands, but not all.

7.11 CONFIDENCE TEST DESCRIPTOR

The confidence test descriptor is a data structure that is used by the firmware to present the information about the on-board confidence test sequence. If any error occurs during the test sequence, this structure will present pertinent test information such as point of failure and failing data.

This structure is located in the dual-port memory just after the IPC CSR space and before the dump area. A non-zero value in the fatal error counter field indicates that the test has failed, the composite status word indicates what sub-test is executing, and other fields tell more about the fault address and pattern used in the test.




```
struct ctdesc {
        unsigned short  csw,     /* composite status word */
                        resv;
        unsigned long   magic,   /* magic number */
                        lcnt,    /* loop counter */
                        fatal,   /* fatal error counter */
                        error,   /* non-critical error counter */
                        faddr,   /* fault address */
                        expdata, /* expect data */
                        readata; /* read data */
};
```

FIGURE 7-10. CONFIDENCE TEST DESCRIPTOR FORMAT

Figure 7-10 illustrates the format of the Confidence Test Descriptor structure and its "C" style declaration. It consists of a composite status word (CSW), a loop counter (LC), a fault address (FA), an expected data, and a read data field. The details of each field is described below.

Composite Status Word (csw)

The composite status word is updated by each subtest of the confidence test to indicate a subtest has started, successfully completed or failed. It provides information such as subtest number, submodule number, and status number. Refer to Appendix C for more details.

Magic Number (magic)

The firmware startup routine compares this field against a predefined pattern to decide whether to run the confidence test or not. If there is a match indicating a warm start condition, it bypasses the whole test sequence. Otherwise, the cold start condition is detected and the tests are invoked. At the completion of the test, this field is initialized to the warm start condition preventing the tests from running in the next subsequent reset until power down.

The current implementation of the MVME332XT confidence test uses the string of "W332" for such a pattern.

Loop Counter (lcnt)

The loop counter field is used to count number of loops that the confidence test has passed during the burn-in test sequence.

Fatal Counter (fatal)

This field records the number of fatal errors occurring in the test sequence. A zero indicates the test has passed.

Error Counter (error)

This field records the number of non-fatal or soft errors occurring in the test sequence. A non-zero value in this field does not prevent the test from continuing, but indicates that such an error can be tolerated since it is correctable.

Fault Address (faddr)

The address of the failure is saved in this field when the test fails.

Expect Data (expdata)

This field indicates to the data that the test is expecting, as a result of the test.

Read Data (readata)

This field indicates the data is obtained from the test. If it is different from the "Expect Data" field, the test fails.

7.12 FRAME FORMAT

In WRITE operation, data can be directly placed into the WRITE ring without any extra effort or frame work. But in READ operation, to support a read-ahead scheme, data must be returned in a form of frame. Multiple frames can be processed and returned to the host at one burst. This will reduce the number of packets traveling back and forth between the host and the firmware when the READ ring is empty.

As shown in Figure 7-11, a frame starts out with a "count" field, then data, and ends with end-of-frame "eof" byte.

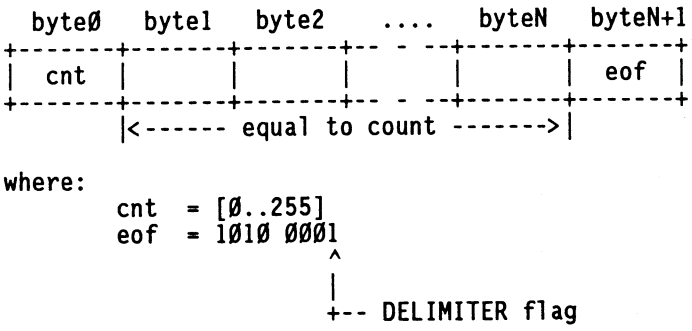


FIGURE 7-11. FRAME FORMAT

The "cnt" field (1 byte) indicates actual number of characters in a frame, the maximum is 255 bytes.

The "of" field (1 byte) has its first nibble pattern is ØxA, and bit Ø is the DELIMITER flag which indicates a complete input line. An input line is terminated by any following characters or conditions: a carriage-return (CR), a line-feed (LF), an end-of-line (EOL), an end-of-line 2 (EOL2), an end-of-file (EOF), and in RAW mode when VMIN is satisfied, or VTIME has expired.

In most case, an input line will fit into a frame, but in the case where the line is too long (longer than 255 characters) or the READ ring is full, it will be broken down into many frames with only the last frame containing a DELIMITER bit set in "eof" field.

Note that a frame is not a data structure, since its body may be wrapped around in the READ ring when the "put" index is less than the "get" index as demonstrated in Figure 7-12.

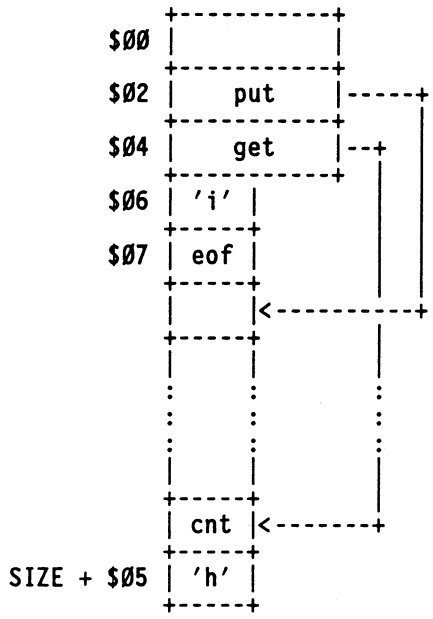


FIGURE 7-12. FRAME IN A READ RING

CHAPTER 8
COMMAND PACKETS

8.1 INTRODUCTION

This section describes the basic operation of the firmware and the details of the function of each command, the required packet format, and expected status results.

After system reset, the firmware clears the BUSY bit in the IPC_CSR when the confidence test completes. It then enters an idle loop, waiting for a "create channel" command to be issued by the host to establish a communication protocol. The host checks for the BUSY bit going away, allocates channel data structures including packets and envelopes on the dual-port memory, and then sends the "create channel" command to the firmware. The firmware is now able to accept some basic command packets such as OPEN, IOCTL, WRITE_WAKEUP, READ_WAKEUP, and CLOSE packet to assert a device's modem control lines, to change a device configuration, to notify the host if a WRITE ring is empty or a READ ring is full, and to negate a device's modem control lines, respectively. Figure 8-1 demonstrates a general flow of the board initialization and the use of commands.

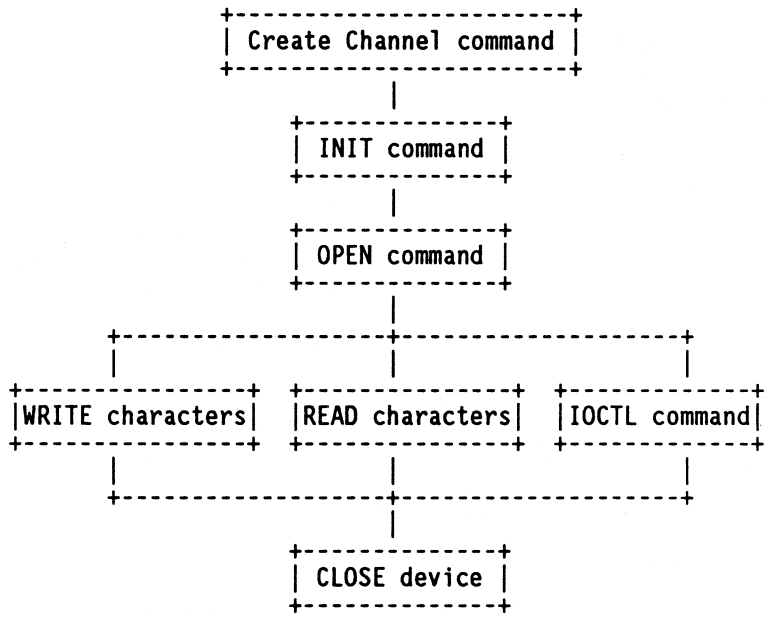


FIGURE 8-1. GENERAL COMMAND FLOW

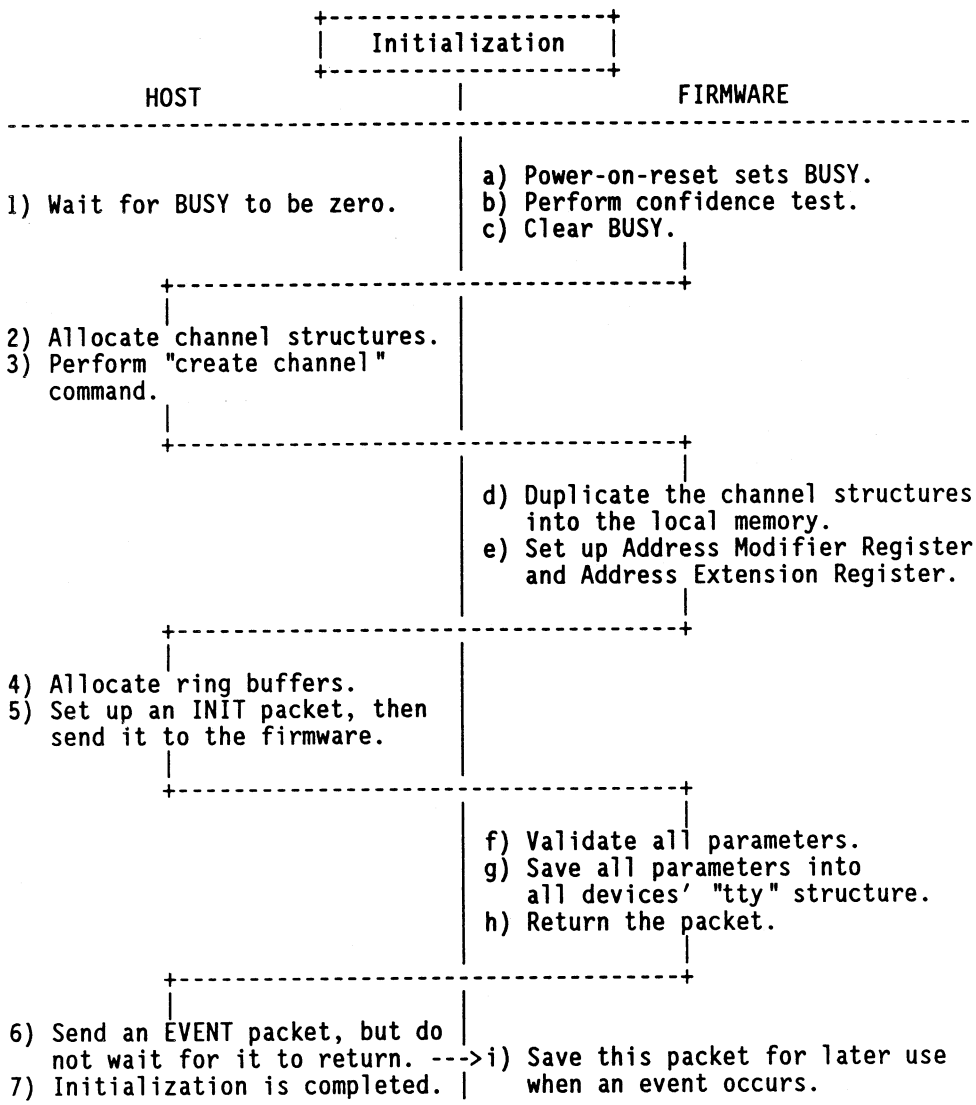


FIGURE 8-2. BOARD INITIALIZATION SEQUENCE

8.2 INITIALIZATION

Figure 8-2 presents the flow of the board initialization. After a channel is established as a result of the create_channel command (CSR command), the host sets up an INIT packet with all required parameters such as ring buffer pointers, ring buffer size, default

termio, interrupt vector, and interrupt level, and then sends it to the firmware. Upon receiving this packet, the firmware saves all the information into an internal "tty" structure associated to each device, and returns the packet back to the host with an error status if it detects any invalid parameters. In order to receive an event taking place on an abnormal situation such as break, quit character, interrupt character, or lost of DCD, the host sends an EVENT packet to the firmware so that the firmware may use it to return an event code in the packet. Note that the host does not have to wait for the packet to return since the firmware may not have any events pending in its data structure yet.

8.2.1 INIT Packet

The required parameters for this packet are illustrated in Figure 8-3. The command pipe number is obtained from the create_channel command. The status pipe number is equal to either the command pipe number or $FF if the host desires the packet to be returned in the same channel. Interrupt level and vector, in this example, are set to level 4 and vector $7F, respectively. The firmware uses these to interrupt the host when it returns a packet. The following is a brief description of each field.

Pointer to init_info array

This field points to the array of nine init_info structures, one for each device. The init_info is described in the "Data Structure" section. It contains the information about ring buffers and default termio.

Error

Error code ERR_PARM is returned to indicate that there is an invalid parameter specified in the packet or init_info. The following error codes are defined.

1. ERR_BUSY (251): The previous initialization is in progress.
2. ERR_PARM (252): Possible invalid parameters are listed below:
   a. Any ring size is smaller than 2 or not a power of two.
   b. Any ring address is located on an odd address.
   c. Any default line discipline number specified in a default termio is greater than the maximum number of lines (greater than 2) available in the ROM.
3. ERR_UNIT (253): Invalid device specified in the packet.
4. ERR_CHAN_NO (255): Invalid BBP communication channel number.

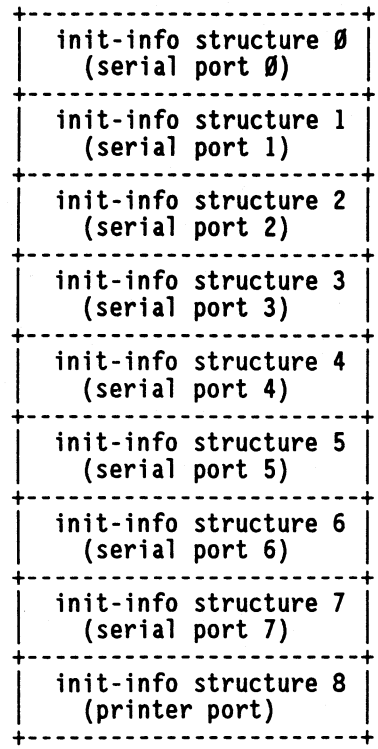
```
      +-------------------------------------------------------------+
$00   |  ---------------------------------------------------------  |
$02   |  ---------------------------------------------------------  |
$04   |   command pipe number --[01] | -- status pipe number ---[FF]|
$06   |  ---------------------------------------------------------  |
$08   |  ---------------------------------------------------------  |
$0A   |  ---------------------- command -------------------[0000]   |
$0C   |  ------------------------- | ----------------------------   |
$0E   |  ------------------------- | ---- interrupt level ----[04]  |
$10   |  ------------------------- | ----------------------------   |
$12   |  ------------- pointer to init_info array --msw------[FF78] |
$14   |  ------------- pointer to init_info array --lsw------[F000] |
$16   |  ---------------------------------------------------------  |
$18   |  ---------------------------------------------------------  |
$1A   |  ---------------------------------------------------------  |
$1C   |  ------------------------- | ---- interrupt vector ---[7F]  |
$1E   |  ---------------------------------------------------------  |
$20   |  ---------------------------------------------------------  |
$22   |  ---------------------------------------------------------  |
$24   |  ---------------------- error ----- msw -----------------   |
$26   |  ----------------------error ----- lsw -----------------   |
$3E   +-------------------------------------------------------------+
```

```
struct init_packet {      /* C style declaration */
        char              eye_catcher[4];
        unsigned char     command_pipe_number;
        unsigned char     status_pipe_number;
        char              filler_0[4];
        short             command;
        char              filler_1[3];
        char              interrupt_level;
        char              filler_2[2];
        struct init_info *init_info_ptr;
        char              filler_3[7];
        char              interrupt_vector;
        char              filler_4[6];
        long              error;
} init_packet;
```

FIGURE 8-3. INIT PACKET FORMAT

8.2.2 Init_info Array

Init_info array contains nine init_info structures, one per device with the element 0 for the serial device 0, and the element 8 for the printer device as illustrated in Figure 8-4. It allows the host to configure each ring buffer size or default baud rate for each device independently. For example, the default baud rate and READ ring size for port 0 can be set to 38.4K baud and 8K bytes, respectively.




```
/* C style declaration */
struct init_info init_info_array[9];
```

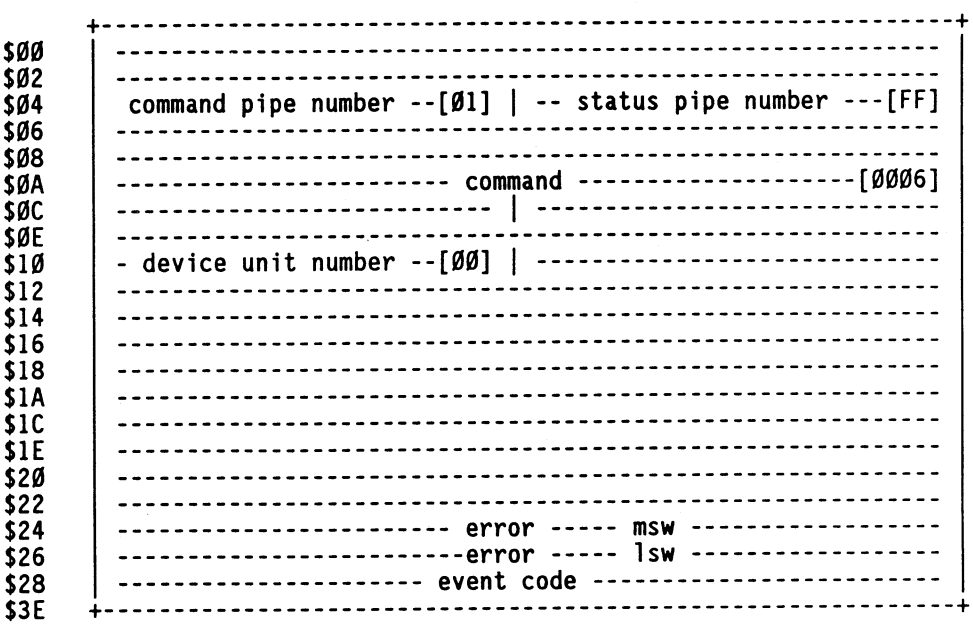
FIGURE 8-4. INIT_INFO ARRAY

8.2.3 EVENT Packet

The EVENT packet is used by the firmware to return an event code to the host to indicate that an abnormal condition such as lost of DCD, Break, or Interrupt character has occurred. The host may use this for any purpose such as wake up a process, kill a process, or just ignore it. An event taking place before the packet arrived is latched into the internal "tty" structure associated to each device, and is returned as soon as the packet is received.

To receive events of all devices, the host must send one EVENT packet to each device. A port that does not have a EVENT packet never returns any event. An event can only occur when a device is enabled by a OPEN packet. A CLOSE packet prevents any event from happening.

Figure 8-5 illustrates a format of EVENT packet, the device_unit_number field specifies what device will receive the packet.




```
struct event_packet {    /* C style declaration */
      char              eye_catcher[4];
      unsigned char     command_pipe_number;
      unsigned char     status_pipe_number;
      char              filler_0[4];
      short             command;
      char              filler_1[4];
      char              device_number;
      char              filler_2[19];
      long              error;
      unsigned short    event_code; /* returned from IPC */
} event_packet;
```

FIGURE 8-5. EVENT PACKET FORMAT

The following is a brief description of each field.

Error

Only one error code (EINVAL) is returned to indicate that the device unit number is out of range.

Event Code

The event code is returned by the firmware to indicate what event has occurred on the firmware side. Each bit in this field represents an event. Multiple events can be returned to the host at one time. The host can configure the firmware to accept or mask some events per port basis; for instance, break can be ignored if the IGNBRK bit is set in the device's termio structure.

Table 8-1 shows all possible events returned by the firmware. The interrupt character, quit character, and switch character are configurable in a device's termio structure. The hang-up condition is usually a lost of DCD, if E_LOST_DCD bit sets, baud rate is changed to zero baud rate, or a CLOSE packet is successfully completed. Any transition of a hardware control line such as DCD, DSR, CTS, PRINTER_FAULT, PRINTER_POUT, or PRINTER_SELECT will cause an EVENT.

TABLE 8-1. EVENT CODE TABLE

Event Code	Event Value	Description
E_INTR	$0001	An interrupt character has been received.
E_QUIT	$0002	A quit character has been received.
E_HUP	$0004	Hang-up condition has occurred.
E_DCD	$0008	DCD has become asserted.
E_DSR	$0010	DSR has become asserted.
E_CTS	$0020	CTS has become asserted.
E_LOST_DCD	$0040	DCD has become negated.
E_LOST_DSR	$0080	DSR has become negated.
E_LOST_CTS	$0100	CTS has become negated.
E_PR_FAULT	$0200	Printer FAULT has become asserted.
E_PR_POUT	$0400	Printer PAPER_OUT has become asserted.
E_PR_SELECT	$0800	Printer SELECT has become asserted.
E_SWTCH	$4000	A switch character has been received.
E_BREAK	$8000	A break sequence has occurred.

8

8.2.4 Initialization Example

Figure 8-6 illustrates an example of an initialization routine. It assumes all data structures are declared on the dual-port RAM. The details of the structures can be found in the DATA STRUCTURE section.

```
Initialization()
{
        /* device default configuration, baud rate, parity  */
        static struct termio def_termio = { Ø ... };

        /* set up init_info_array */
        for (i = Ø; i < 9, i++)
        {
            init_info_array[i].write_ring_size = 4Ø96;
            init_info_array[i].read_ring_size  = 2Ø48;
            init_info_array[i].default_termio = def_termio;
        }
        /* set up an init_packet */
        init_packet.command = INIT;
        init_packet.command_pipe_number = channel.chan_number;
        init_packet.status_pipe_number = ØxFF;
        init_packet.interrupt_level = 4;
        init_packet.interrupt_vector = Øx7f;
        init_packet.init_info_ptr = init_info_array;

        /* send the packet to the firmware, then wait for
           completion */
        bpp_send(&channel, &init_packet, WAIT);

        /* check for error */
        if (init_packet.error != Ø) {
             bad_parameter()
        }

        /* send one event packet to each device */
        for (i = Ø; i < 9, i++)
        {
            /* set up event packet */
            event_packet[i].command = EVENT;
            event_packet[i].device_number = i;
            event_packet[i].command_pipe_number =
                                    channel.chan_number;
            event_packet[i].status_pipe_number = ØxFF;

            /* send the packet to the firmware, but do not
               wait for completion */
            bpp_send(&channel, &event_packet[i], DO_NOT_WAIT);
        }
}
```

FIGURE 8-6. INITIALIZATION EXAMPLE

8.3 OPEN DEVICE

In order to enable a device for data transfer, the host sends an OPEN packet to the firmware. Upon receiving this packet, the firmware configures the device based on the information supplied by the INIT packet in the default termio, asserts all the device's modem control lines, and then enables the device interrupt. The device is now able to accept characters for receiving and transmitting. The OPEN packet is returned to the host with the current status of the device DCD. The host decides whether to wait for DCD to be asserted or not, if the DCD is negated. The firmware will notify the host when the DCD becomes asserted by returning an EVENT packet.

Once a device is opened, subsequent OPEN packets only return the information about the device DCD without actual device configuration.

Figure 8-7 illustrates the flow of the open sequence.

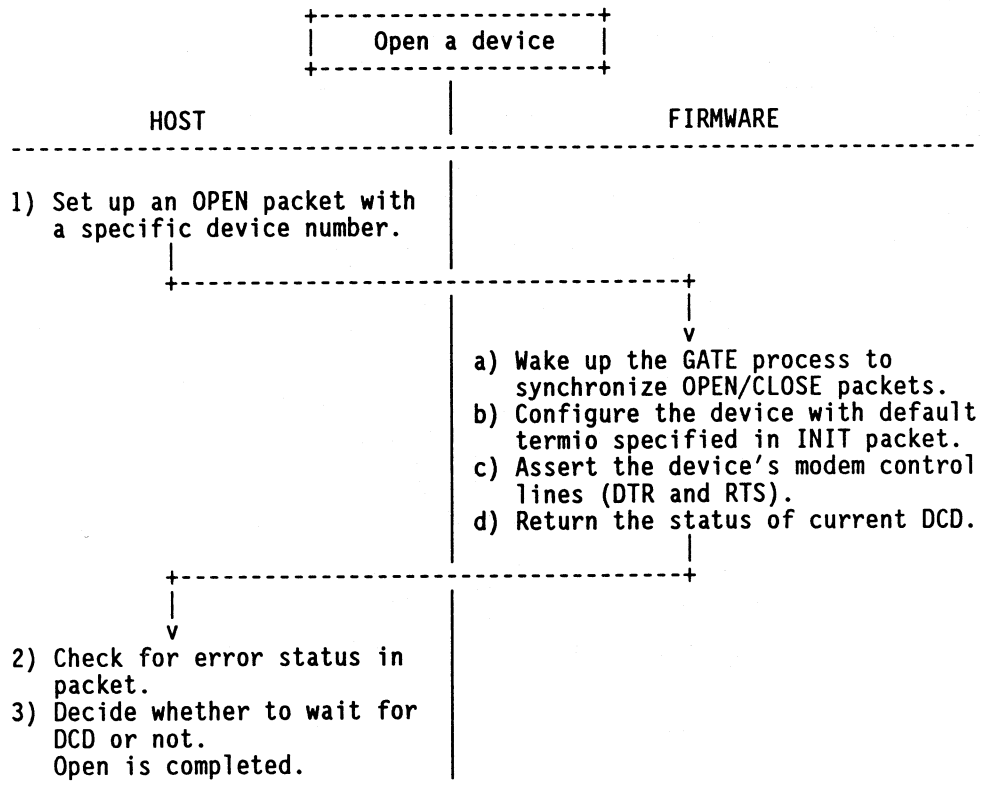
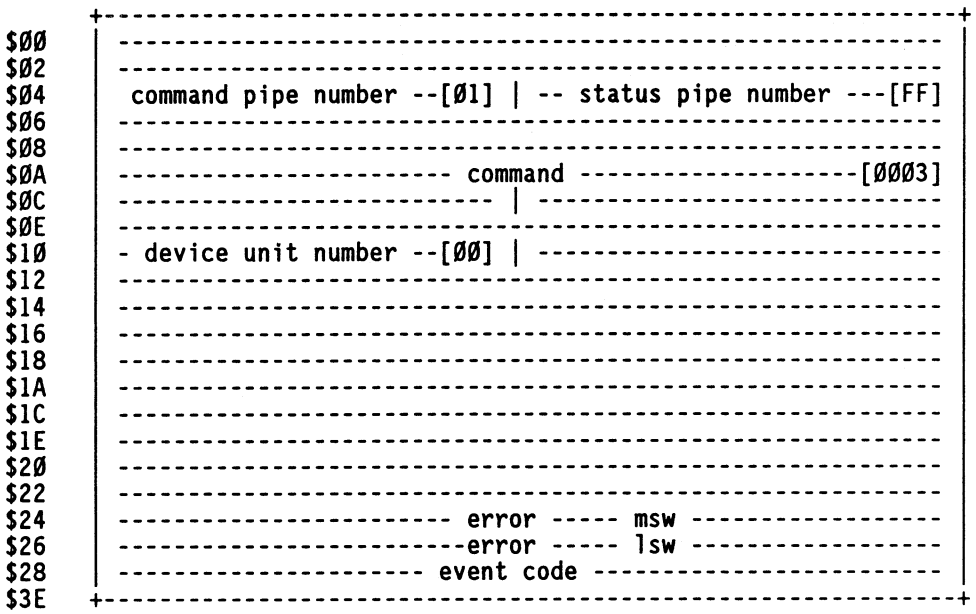


FIGURE 8-7. OPEN DEVICE SEQUENCE

8.3.1 OPEN Packet

The required parameters for this packet are illustrated in Figure 8-8. The device_unit_number field specifies what device will receive the packet. The command pipe number and status pipe number are obtained from the create channel command. This packet is almost identical to the EVENT packet except command is set to OPEN.




```
struct open_packet {        /* C style declaration */
          char              eye_catcher[4];
          unsigned char     command_pipe_number;
          unsigned char     status_pipe_number;
          char              filler_0[4];
          short             command;
          char              filler_1[4];
          char              device_number;
          char              filler_2[19];
          long              error;
          unsigned short    event_code; /* returned from IPC */
} open_packet;
```

FIGURE 8-8. OPEN PACKET FORMAT

The following is a brief description of each field.

Error

Only one error code (EINVAL) is returned to indicate that the device unit number is out of range.

Event Code

Only one event code returned in this field is E_DCD to indicate the current status of a device's DCD or PRINTER_SELECT line. A zero in this bit implies the signal is negated or inactive. Refer to EVENT packet format for more information about event code.

8.3.2 Open Example

The following code provides an example of an open routine. It assumes all data structures are declared on the dual-port RAM. The details of the structures can be found in the DATA STRUCTURE section.

```
open(dev)
{
        /* set up an open_packet */
        open_packet.command = OPEN;
        open_packet.device_number = dev;
        open_packet.command_pipe_number = channel.chan_number;
        open_packet.status_pipe_number = ØxFF;

        /* send the packet to the firmware, then wait for completion */
        bpp_send(&channel, &open_packet, WAIT);

        /* check for error */
        if (open_packet.error != Ø)
                invalid_device_number()
       if (open_packet.event_code & E_DCD)  /* DCD is asserted */
              return;
       else                          /* DCD is negated, wait for DCD */
              wait_event(DCD);
}
```

FIGURE 8-9. OPEN EXAMPLE

8.4 CLOSE DEVICE

To prevent any data transfer or any event from happening to a device, the host sends a CLOSE packet to the firmware. When the firmware receives this packet, it waits for all outstanding characters in the WRITE ring and the OUTPUT ring to be transmitted, then negates all the device's modem control lines and disables the device. Any data written to the ring while the device is closed will not be processed until an OPEN packet is received.

If a HUPCL (hang up on last close) bit is set in the device's termio structure, an EVENT packet will be returned to the host with an E_HUP event code so that the host driver can notify all tasks associated to the device.

Figure 8-1Ø illustrates the flow of a close sequence.

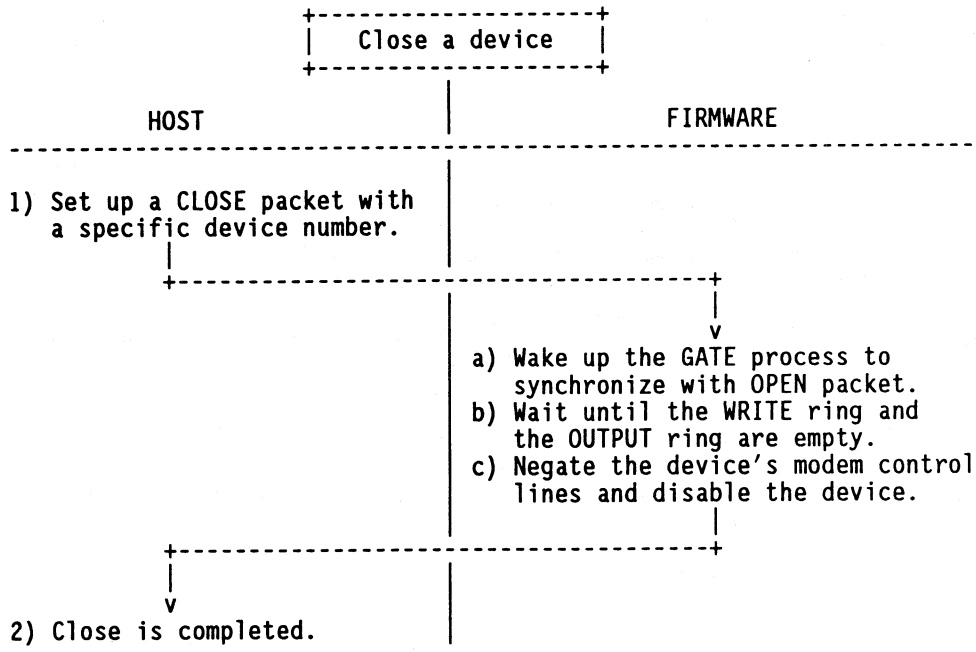
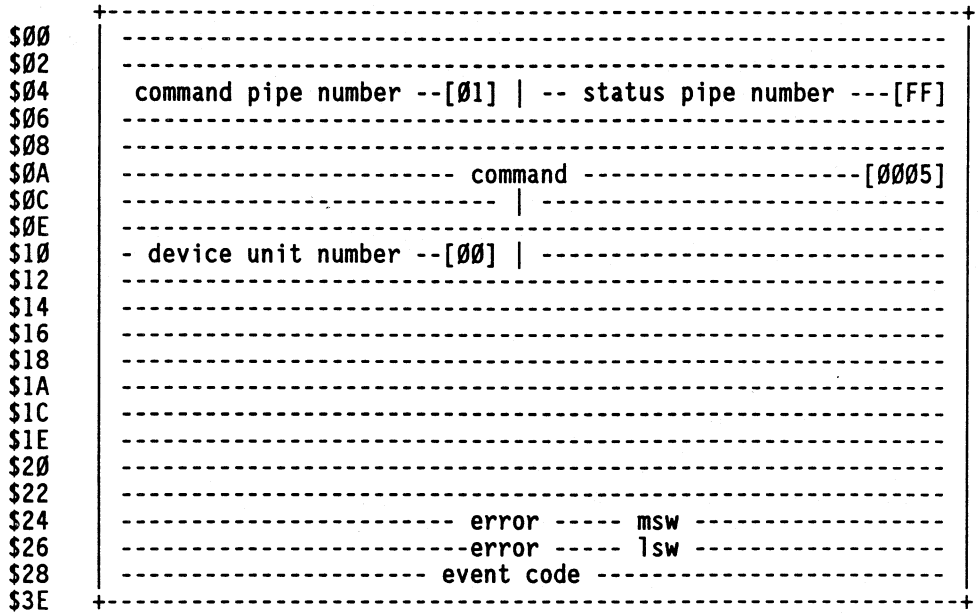


FIGURE 8-1Ø. CLOSE DEVICE SEQUENCE

8.4.1 CLOSE Packet

The required parameters for this packet are illustrated in Figure 8-11. The device_unit_number field specifies what device will receive the packet. The command pipe number and status pipe number are obtained from the create channel command. This packet is identical to the OPEN packet except command is set to CLOSE instead.




```
struct close_packet {      /* C style declaration */
          char             eye_catcher[4];
          unsigned char    command_pipe_number;
          unsigned char    status_pipe_number;
          char             filler_Ø[4];
          short            command;
          char             filler_1[4];
          char             device_number;
          char             filler_2[19];
          long             error;
          unsigned short   event_code; /* returned from IPC */
} close_packet;
```

FIGURE 8-11. CLOSE PACKET FORMAT

The following is a brief description of each field.

Error

Only one error code (EINVAL) is returned to indicate that the device unit number is out of range.

Event Code

Only one event code returned in this field is E_DCD to indicate the current status of a device's DCD or PRINTER_SELECT line. A zero in this bit implies the signal is negated or inactive. Refer to EVENT packet format for more information about event codes.

8.4.2 Close Example

The following code provides an example of a close routine. It assumes that all data structures are declared on the dual-port RAM. The details of the structures can be found in the DATA STRUCTURE section.


```
close(dev)
{
    /* set up an open_packet */
    close_packet.command = CLOSE;
    close_packet.device_number = dev;
    close_packet.command_pipe_number = channel.chan_number;
    close_packet.status_pipe_number = ØxFF;

    /* send the packet to the firmware, then wait for
       completion */
    bpp_send(&channel, &close_packet, WAIT);

    /* check for error */
    if (close_packet.error != Ø)
            invalid_device_number()
}
```

FIGURE 8-12. CLOSE EXAMPLE

8.5 READ CHARACTERS

To get characters from a device, the host simply checks the READ ring associated to the device to determine whether the ring is empty or not. If the ring is not empty, a simple copy is performed from the ring to a task's buffer. If the ring is empty and the host is willing to wait for characters to arrive, the READ_WAKEUP packet can be used to inform the firmware that the host is waiting for an input line to be available. The packet will be returned as soon as the firmware's ICP process completes at least one cooked line in the READ ring.

When the driver interrupt routine receives this type of packet, it simply wakes up whatever tasks are waiting on the packet address so that the task can continue to get characters out from the ring. The firmware is able to accept many READ_WAKEUP packets for each device as long as the host can supply, but it will return all of them at once when it has a cooked line (the line that ends with a delimiter). Figure 8-13 shows a flow of a read sequence.

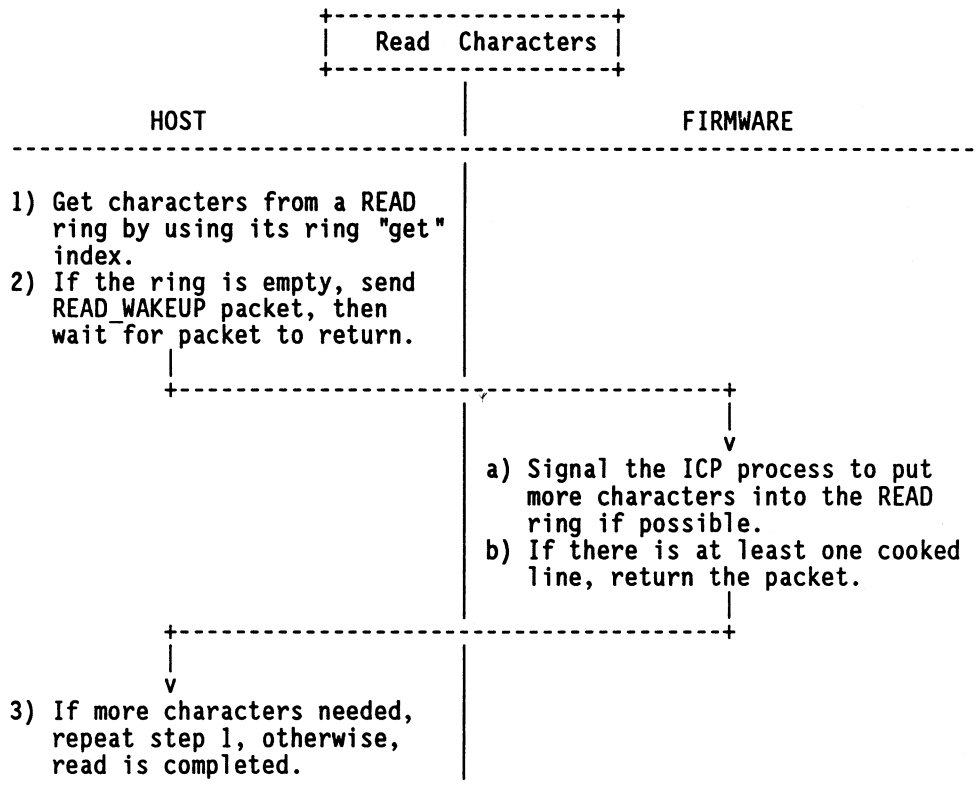
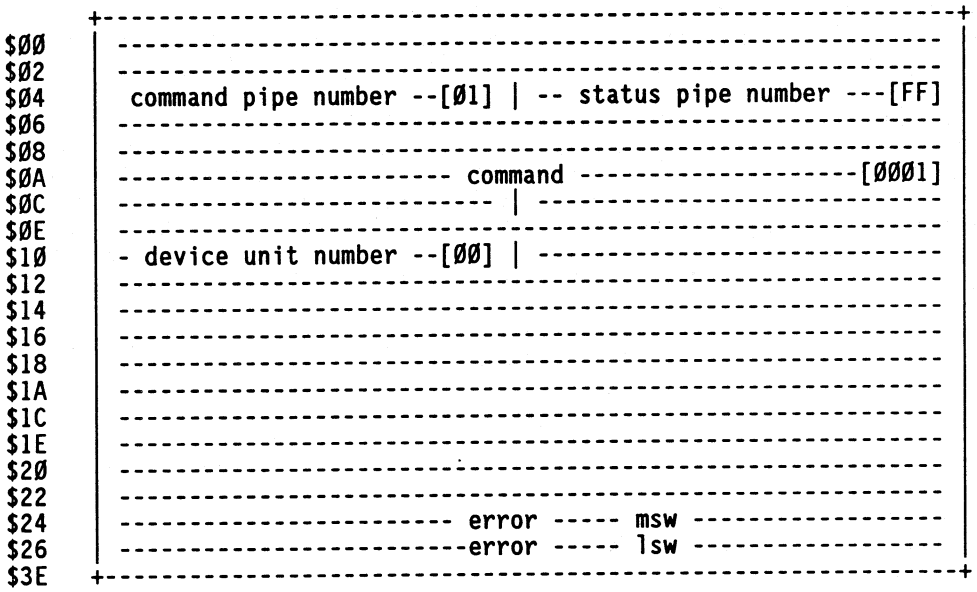


FIGURE 8-13. READ DEVICE SEQUENCE

8.5.1 READ_WAKEUP Packet

The required parameters for this packet are illustrated in Figure 8-14. The device_unit_number field specifies what device will receive the packet. The command pipe number and status pipe number are obtained from the create_channel command.




```
struct read_wakeup_packet {   /* C style declaration */
          char                eye_catcher[4];
          unsigned char       command_pipe_number;
          unsigned char       status_pipe_number;
          char                filler_0[4];
          short               command;
          char                filler_1[4];
          char                device_number;
          char                filler_2[19];
          long                error;
} read_wakeup_packet;
```

FIGURE 8-14. READ_WAKEUP PACKET FORMAT

The following is a brief description of each field.

Error

The EINVAL error code is returned to indicate that the device unit number is out of range.

The EIO error code is returned if a CLOCAL bit is not set in the device termio structure and the DCD becomes negated.

8.5.2 Read Characters Example

The following code provides an example of a read routine. The host should use the get index and never change the put index. It assumes that a frame does not cross the ring boundary. The READ ring is a circular buffer, therefore, the host must properly maintain pointers on frames that "wrap" around. The firmware guarantees that there must be a cooked line (with the delimiter flag in the end-of-frame field) if the ring is not empty.

```
read(dev, buffer)
{
    get = read_ring[dev].get;
    put = read_ring[dev].put;

    if (get != put) {              /* character available */
        frame_count = read_ring[dev].data[get];
        block_copy(&read_ring[dev].data[get + 1], buffer,
                                              frame_count);
        /* end-of-frame flag */
        eof = read_ring[dev].data[get + frame_count + 1];
        if (eof & DELIMITER)        /* a complete line */
              return;
    }
    else {
         /* set up an READ_WAKEUP packet */
         read_wakeup_packet.command = READ_WAKEUP;
         read_wakeup_packet.device_number = dev;
         read_wakeup_packet.command_pipe_number =
                                    channel.chan_number;
         read_wakeup_packet.status_pipe_number = ØxFF;

         /* send the packet to the firmware, then wait for
            the packet to be returned */
         bpp_send(&channel, &read_wakeup_packet, WAIT);

         /* check for error */
         if (read_wakeup_packet.error != Ø)
                lost_of_DCD()
    }
}
```

FIGURE 8-15. READ CHARACTERS EXAMPLE

8.6 WRITE CHARACTERS

Before writing characters to the device's WRITE ring buffer, the host checks the ring to determine whether the ring is full or not. If the ring is not full, the host simply copies characters from a task's buffer to the ring. Otherwise, if the ring is full and the task desires to wait, the host sends a WRITE_WAKEUP packet to the firmware so that the firmware will use it to inform the host when there are available spaces in the ring.

When the driver interrupt routine receives this type of packet, it simply wakes up whatever tasks are waiting on the packet address so that the tasks can continue to put more characters into the ring. The firmware is able to accept as many WRITE_WAKEUP packets for each device as the host can supply. But, it will return all of them at once, when the number of characters in the device's WRITE ring drops below a low water mark (quarter of ring size).

This can introduce a problem when multiple output strings are interleaved on a terminal display. It is the host's responsibility to implement a record locking mechanism in the driver write routine to prevent such conditions from occurring.

Figure 8-16 illustrates the flow of a write sequence.

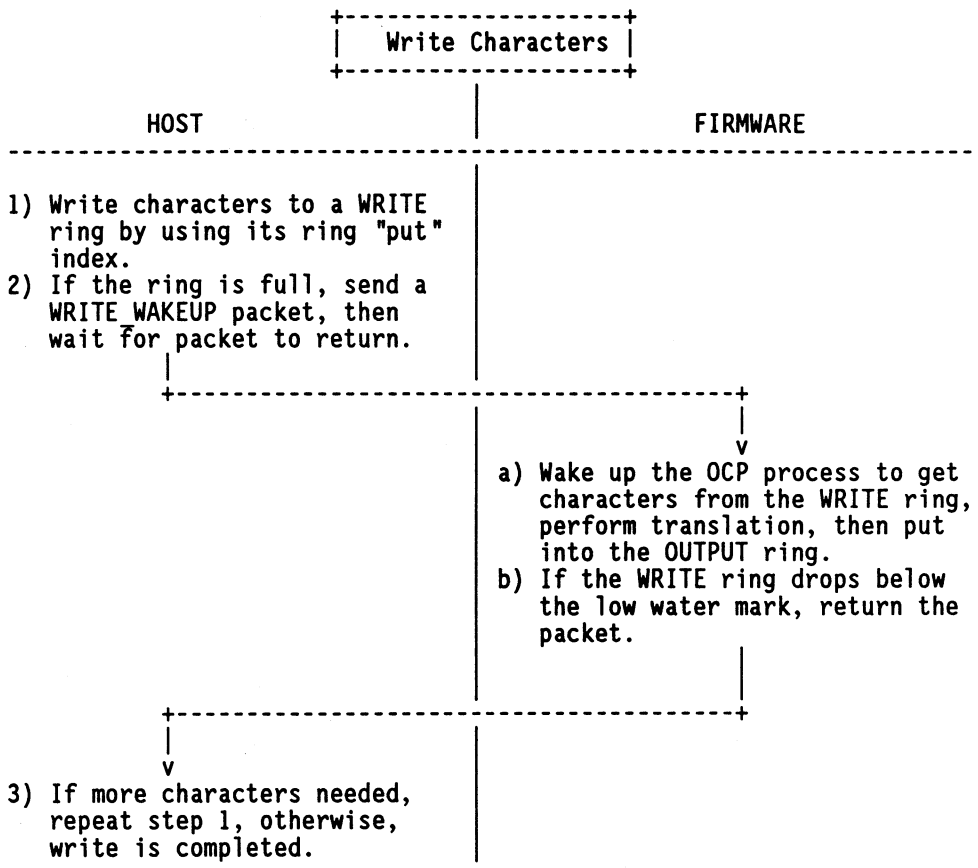
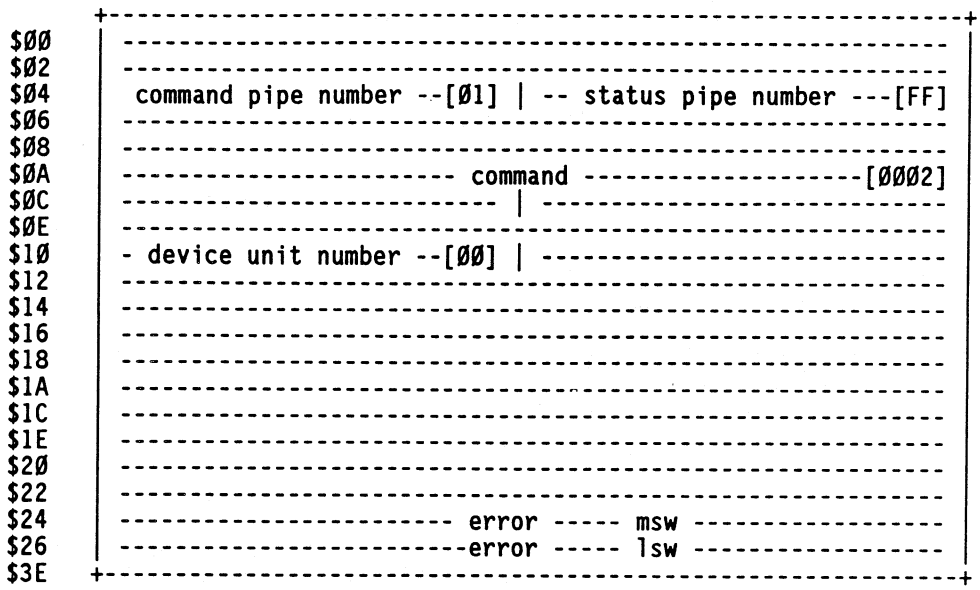


FIGURE 8-16. WRITE CHARACTERS SEQUENCE

8.6.1 WRITE_WAKEUP Packet

The required parameters for this packet are illustrated in Figure 8-17. The device_unit_number field specifies what device will receive the packet. The command pipe number and status pipe number are obtained from the create_channel command.



8

```
struct write_wakeup_packet {   /* C style declaration */
          char                 eye_catcher[4];
          unsigned char        command_pipe_number;
          unsigned char        status_pipe_number;
          char                 filler_0[4];
          short                command;
          char                 filler_1[4];
          char                 device_number;
          char                 filler_2[19];
          long                 error;
} write_wakeup_packet;
```

FIGURE 8-17. WRITE_WAKEUP PACKET FORMAT

The following is a brief description of each field.

Error

The EINVAL error code is returned to indicate that the device unit number is out of range.

The EIO error code is returned if a CLOCAL bit is not set in the device termio structure and the DCD becomes negated.

8.6.2 Write Character Example

The following code provides an example of a write routine. It is simpler than the read routine since there is no frame format on the ring at all. However, the host may have to perform a block_copy twice if the available space crosses the bottom of the ring. The host should use the put index and never change the get index.

```
write(dev, buffer)
{
    get = write_ring[dev].get;
    put = write_ring[dev].put;

    if ((put + 1) == get) {                /* the ring is FULL */
        /* set up an write_wakeup_packet */
        write_wakeup_packet.command = WRITE_WAKEUP;
        write_wakeup_packet.device_number = dev;
        write_wakeup_packet.command_pipe_number =
                                        channel.chan_number;
        write_wakeup_packet.status_pipe_number = ØxFF;

        /* send the packet to the firmware, then wait for the
           packet to be returned */
        bpp_send(&channel, &write_wakeup_packet, WAIT);

        /* check for error */
        if (write_wakeup_packet.error != Ø)
                lost_of_DCD()
    }
    else
        block_copy(buffer, &write_ring[dev].data[put], get - put);
}
```

FIGURE 8-18. WRITE CHARACTERS EXAMPLE

8.7 CONTROL A DEVICE

Beside open, close, read, and write, the firmware allows the host to perform some special commands that do not fit into the above categories via a IOCTL packet. In this packet a sub command field, called ioctl command, is required to distinguish such special commands such as configure a device, flush a buffer, suspend an output, or download a chunk of code. Some ioctl commands need more parameters attached to the packet. If the parameters are needed, the packet parameter block can be used for this purpose. In these circumstances, the parameter block can be a termio structure, a termcb structure, or a download info structure dependent on what ioctl command is being specified.

Upon receiving at this type of packet, the firmware hands it to the CTL process associated with the device which further decodes the ioctl command field to carry out the requested function. In some cases, the CTL process waits for all output characters to be transmitted before changing the device configuration as in the TCSETAW command.

Multiple IOCTL packets can be sent to a device's CTL process at once. They will be queued in an order of First-In-First-Out (FIFO). The CTL process has to complete one packet before processing the next one. However, if they are sent to all different CTL processes i.e., for different devices, they will be executed concurrently.

Figure 8-19 illustrates the flow of a control sequence.

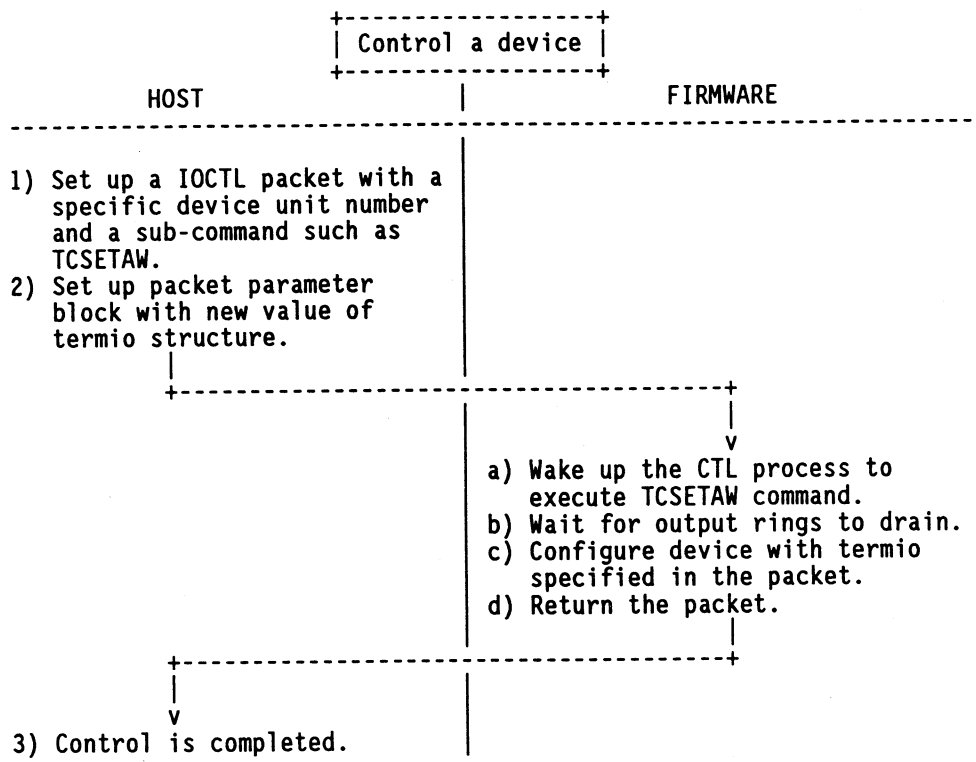


FIGURE 8-19. CONTROL A DEVICE SEQUENCE

8.7.1 IOCTL Packet

The required parameters for this packet are illustrated in Figure 8-2Ø. The command pipe number is obtained from the create_channel command. The status pipe number is equal to either the command pipe number or $FF, if the host desires the packet to be returned in the same channel. The device number field specifies what device this packet will be for.

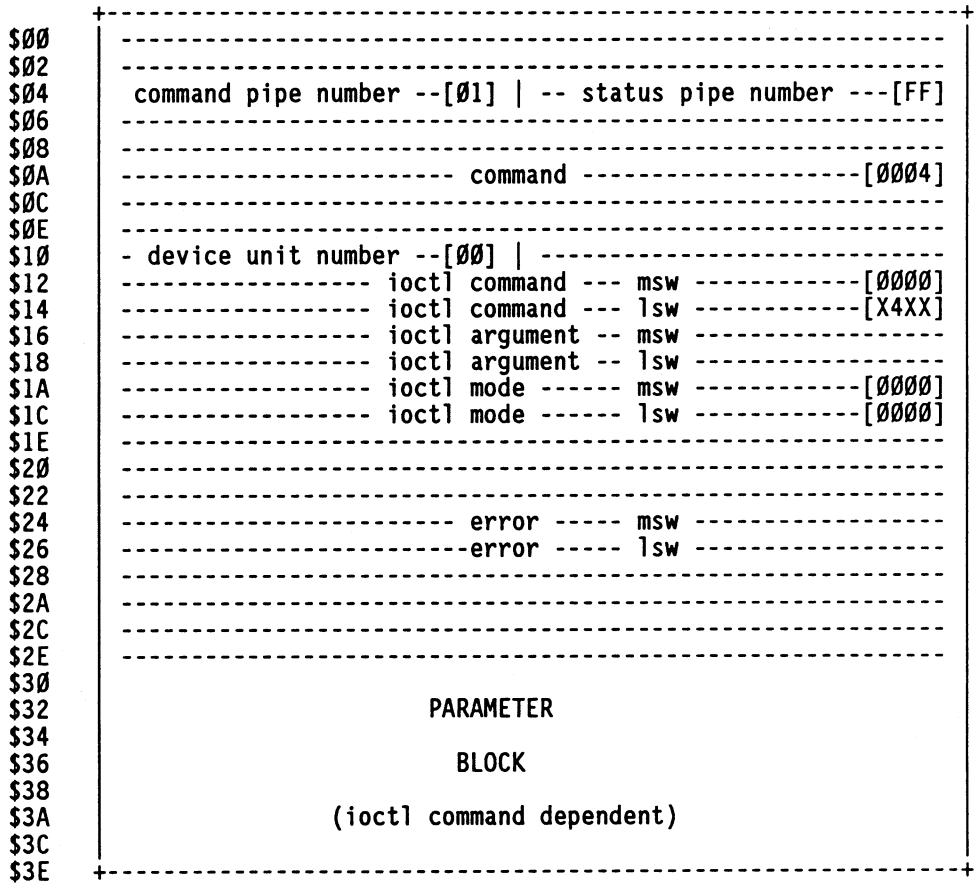


FIGURE 8-2Ø. IOCTL PACKET FORMAT

8

```
struct ioctl_packet {   /* C style declaration */
    char          eye_catcher[4];
    unsigned char command_pipe_number;
    unsigned char status_pipe_number;
    char          filler_0[4];
    short         command;
    char          filler_1[4];
    unsigned char device_number;
    char          filler_2[1];
    long          ioctl_command;
    long          ioctl_argument;
    long          ioctl_mode;
    char          filler_3[6];
    long          error;
    char          filler_4[8];

    union {
            struct termio  tio;  /* termio structure */
            struct termcb  tcb;  /* termcb structure */
            struct sgttyb  sgt;  /* sgttyb structure */
            struct dl_info dl;   /* download info structure */
            long           param;/* general purpose parameter */
    } parameter_block;           /* ioctl_command dependent */

} ioctl_packet;
```

FIGURE 8-20. IOCTL PACKET FORMAT (cont.)

The following is a brief description of each field.

Ioctl Command

As mentioned above, this field contains a sub command for a specific function. Some of them are derived from the UNIX ioctl commands while others are private for the MVME332XT. Table 8-2 provides a list of all ioctl commands. Details will be discussed in the next sections or in the appendix.

TABLE 8-2. IOCTL COMMANDS

Ioctl Commands	Value	Description
LDOPEN	$4400	Set device internal state to open.
LDCLOSE	$4401	Clear device internal state, flush all rings.
LDCHG	$4402	No operation, return no error.
LDGETT	$4408	Get current virtual terminal information into the termcb structure in the packet.
LDSETT	$4409	Set virtual terminal parameters to the new one.
TCGETA	$5401	Get a device's current termio structure.
TCSETA	$5402	Change a device's termio structure to the new one.
TCSETAW	$5403	Same as TCSETA but wait for device's WRITE ring and OUTPUT ring to drain.
TCSETAF	$5404	Same as TCSETAF but flush the input rings after waiting for the output rings to drain.
TCSBRK	$5405	Transmit a Break Sequence on output (as long as 250ms).
TCXONC	$5406	Suspend or resume the output, send XON or XOFF, assert or negate RTS or DTR depend on ioctl argument field.
TCFLSH	$5407	Flush WRITE ring and OUTPUT ring, READ ring and INPUT ring or both pairs depend on ioctl argument field.
TCSETHW	$5440	Enable or Disable hardware handshake feature.
TCGETHW	$5441	Get current information of hardware handshake.
TCGETDL	$5442	Get the downloadable address and size of MVME332XT local memory.
TCDLOAD	$5443	Download a block of data or code into MVME332XT local memory.
TCLINE	$5444	Copy a line discipline switch table previously downloaded into the internal data structure of MVME332XT.
TCEXEC	$5445	Instruct the firmware to execute a functional address previously downloaded into MVME332XT local RAM.
TCGETVR	$5446	Get the MVME332XT firmware version and revision number.
TCGETDF	$5447	Same as TCGETA command, but get the default open termio structure which is used to configure a device when open.
TCSETDF	$5448	Same as TCSETAW command, but change both the default open termio structure and the working termio structure.
TCGETSYM	$5449	Get the firmware symbol table to link downloadable code.

TABLE 8-2. IOCTL COMMANDS (cont.)

Ioctl Commands	Value	Description
TCWHAT	$544A	Get all SCCS IDs of the firmware files.
TCGETDS	$544C	Current status of DCD, CTS, DTR, PR_FAULT, PR_POUT, and PR_SELECT.
TIOCGETP	$74Ø8	Get device's current termio structure by using sgttyb structure.
TIOCSETP	$74Ø9	Change device's termio structure by using sgttyb structure.

Ioctl Argument And Mode

These fields may be required by certain ioctl commands for further decoding as in the TCFLSH command.

Error

8

A zero in this field indicates no error. The following is a list of possible errors that may occur for this type of command:

EINVAL

Invalid specified parameter, for example; the line discipline number specified in termio is out of range.

ENOMEM

Not enough memory space supplied by the host to carry out a function, for example; the space for the TCGETSYM command.

ENXIO

No such a device or address, for example; request a non existent table as in ISP software.

EBUSY

The device or address is in use, for example; delete a translation table that is in use by other device in ISP software.

EEXIST

Such a device or address exists, for example; attempt to create a translation table that has been created by other device in ISP software.

Parameter Block

This block is used to transfer data between the host and the firmware. Depending on the ioctl command, it is used as a termio, termcb, sgttyb, or down load structure. If it is required by a command, the host driver will copy the structure from a task's address space into this field before sending to the firmware or copy this block to a task's buffer after receiving the packet.

8.7.2 IOCTL Command Example

To highlight the differences between IOCTL sub-commands, the next examples leave out a few lines of code which are identical to all IOCTL examples at the beginning of a sub-routine.

```
ioctl_packet.command = IOCTL;
ioctl_packet.command_pipe_number = channel_number;
ioctl_packet.status_pipe_number = ØxFF;
ioctl_packet.device_number = dev;
```

8.7.3 TCGETA And TCGETDF Commands

The TCGETDF command returns a device's default termio into the packet parameter block. The default termio is supplied by an INIT packet and is used by the GATE process to configure the device when an OPEN packet is received. The TCGETA command, on the other hand, returns a working termio structure which is a copy of the default termio structure after a device is opened. The working termio structure is identical to the default termio structure if the host has not changed it by using the TCSETA command after the device is opened.

Figure 8-21 illustrates an example of how to use these commands.

```
ioctl_get_termio(dev, termio_pointer)
struct termio *termio_pointer;
{
        ioctl_packet.ioctl_command = TCGETA;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error)
                error_handler();
        else
                copy(&ioctl_packet.parameter_block.tio,
                     termio_pointer, sizeof(struct termio));
}
```

FIGURE 8-21. TCGETA/TCGETDF COMMAND EXAMPLES

8.7.4 TCSETA, TCSETAW, TCSETAF, And TCSETDF Commands

The TCSETA command immediately changes the working termio structure and reconfigures the device based on the termio supplied on the packet parameter block.

The TCSETAW command performs the same function as the TCSETA command, but waits for the device's WRITE ring and OUTPUT ring to drain out completely before changing. This guarantees that pending output characters will be transmitted before critical configurations occur, such as changing the baud rate or character size.

The TCSETAF command performs exactly the same function as TCSETAW. In addition, it flushes the device's READ ring and INPUT ring after waiting for the output rings to drain, and before reconfiguring the device.

The TCSETDF command also performs the same function as the TCSETAW command, but both default termio and working termio are modified. In addition, open calls will be affected by this command.

Figure 8-22 illustrates an example of how to use these commands.

```
ioctl_set_termio(dev, termio_pointer)
struct termio *termio_pointer;
{
        ioctl_packet.ioctl_command = TCSETAF;
        copy(termio_pointer, &ioctl_packet.parameter_block.tio,
                        sizeof(struct termio));
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error)
                error_handler();
}
```

FIGURE 8-22. TCSETA, TCSETAW, TCSETAF, AND TCSETDF COMMAND EXAMPLES

8.7.5 TCSBRK Command

The TCSBRK command waits for the device's WRITE ring and OUTPUT ring to drain out completely before sending out a BREAK sequence. The BREAK sequence is transmitted by asserting the device's transmit data line longer than a character time. However, to ensure that the remote device's receiver, of which the baud rate is unknown to, recognizes a BREAK sequence at the lowest baud (50 baud), the firmware asserts this line for a long period of time (about 250 ms).

Figure 8-23 illustrates an example of how to use this command. The ioctl argument should be set to zero. Otherwise, the BREAK sequence is never sent and the command becomes a command that is used to wait for the output ring to drain.

```
ioctl_send_break(dev)
{
        ioctl_packet.ioctl_command = TCSBRK;
        ioctl_packet.ioctl_argument = Ø;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error != Ø)
                invalid_device();
}
```

FIGURE 8-23. TCSBRK COMMAND EXAMPLE

8.7.6 TCXONC Command

The TCXONC command allows the host to control the output of a device such as suspend or resume an output, send an XON or XOFF, depending on the value of ioctl argument field. Table 8-3 provides the function associated to the ioctl argument.

TABLE 8-3. TCXONC ARGUMENT FUNCTIONS

Ioctl Argument	Function
Ø	Suspend the output, stop the current transmitting character stream until the next resume command.
1	Resume a suspended output, allow characters in the output rings to be transmitted.
2	Send XOFF and negate RTS (if hardware handshake is enabled), tell a remote side to stop sending characters to this port until XON is sent.
3	Send XON character and assert RTS (if hardware handshake is enabled), tell a remote side to start transmitting characters.
4	Assert RTS line on a device.
5	Negate RTS line on a device.
6	Assert DTR line on a device.
7	Negate DTR line on a device.
other	EINVAL error code will be returned in packet error field.

Figure 8-24 illustrates an example of how to use this command.

```
ioctl_send_xoff(dev)
{
        ioctl_packet.ioctl_command = TCXONC;
        ioctl_packet.ioctl_argument = 2;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error != Ø)
                invalid_argument_or_device();
}
```

FIGURE 8-24. TCXONC COMMAND EXAMPLE

8.7.7 TCFLSH Command

The TCFLSH command allows the host to flush device input rings or output rings or both depending on the value of the ioctl argument field. Once a ring is flushed, all characters will be discarded, and the put index is set equal to the get index to indicate an empty ring. Table 8-4 lists what rings are to be flushed according to the value of the ioctl argument field.

TABLE 8-4. TCFLSH ARGUMENT FUNCTIONS

Ioctl Argument	Function
Ø	Flush the input rings including the READ ring and the INPUT ring.
1	Flush the output rings including the WRITE ring and the OUTPUT ring.
2	Flush both input and output rings.
other	EINVAL error code will be returned in packet error field.

Figure 8-25 illustrates an example of how to use this command.

```
ioctl_flush_output(dev)
{
        ioctl_packet.ioctl_command = TCFLSH;
        ioctl_packet.ioctl_argument = 1;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error != Ø)
                invalid_argument_or_device();
}
```

FIGURE 8-25. TCFLSH COMMAND EXAMPLE

8.7.8 TCGETHW And TCSETHW Commands

These two commands are used to obtain current information about the hardware handshake option and how to change the option.

If the hardware handshake is enabled every time the firmware sends an XON character, it asserts the device RTS. Likewise, when the firmware sends the XOFF character to stop a remote side, it negates the RTS. However, the software handshake (XON/XOFF) can be disabled separately by clearing the IXOFF bit in the device working termio structure.

In the TCGETHW command, the information is returned in the packet parameter block. A non zero value in this field indicates that the option is enabled.

In the TCSETHW command, a zero in the ioctl argument field turns the option off while a non-zero value in the argument field turns the option on.

Figure 8-26 illustrates an example on how to use these commands.


```
ioctl_hardware_handshake(dev)
{
        ioctl_packet.ioctl_command = TCGETHW;
        bpp_send(&channel, &ioctl_packet, WAIT);
        if (ioctl_packet.parameter_block.param != Ø)
                printf("hardware handshake is enabled");
        else
        {
                printf("hardware handshake is disabled");

                /* enable the handshake option */
                ioctl_packet.ioctl_command = TCSETHW;
                ioctl_packet.ioctl_argument = 1;
                bpp_send(&channel, &ioctl_packet, WAIT);

        }
}
```

FIGURE 8-26. TCGETHW/TCSETHW COMMAND EXAMPLES

8.7.9 TCGETDL Command

The TCGETDL command returns information about the unused area of the local memory so that the host knows where to download a chunk of code for execution.

For security reasons, the device number is restricted to the printer device which is number 8, otherwise, EINVAL error code is returned. The information about the downloadable area can be found in the dl_info structure of the packet parameter block with the ipc_address, and the count field indicating a starting address and the size (in bytes) of the area, respectively.

Figure 8-27 illustrates an example of how to use this command.

```
ioctl_get_downloadable_area(dev)
{
        ioctl_packet.ioctl_command = TCGETDL;
        bpp_send(&channel, &ioctl_packet, WAIT);
        if (ioctl_packet.error != 0)
                invalid_device();

        printf("free area at address %x, and size %d (in bytes)",
                ioctl_packet.parameter_block.dl_info.ipc_address,
                ioctl_packet.parameter_block.dl_info.count);
}
```

FIGURE 8-27. TCGETDL COMMAND EXAMPLE

8

8.7.10 TCDLOAD Command

The TCDLOAD command allows the host to download a block of data or code into the MVME332XT local memory (the downloadable area obtained by the TCGETDL command).

The host has to set up information such as source, destination, and size of the download block in the dl_info structure of the packet parameter block. The host_address field specifies the source of the data block residing on the dual-port memory. The ipc_address field specifies the destination where the data block will be sent to and the count field specifies the number of bytes to be transferred.

The firmware will make sure that the data block fits completely into the downloadable area before copying the data. Otherwise, the EINVAL error code will be returned and no transfer will take place.

As mentioned earlier, the source buffer has to reside on the dual-port memory. If the buffer is smaller than a request, multiple commands can be performed to satisfy the request as shown in the example.

For security reasons, the device number is restricted to the printer device which is number 8, otherwise, the EINVAL error code is returned.

Figure 8-28 illustrates an example of how to use this command.

```
ioctl_download(dev, task_buffer, firmware_address, count)
char *task_buffer;      /* task buffer address */
char *firmware_address; /* MVME332XT downloadable area address */
{
        extern char download_buffer[BUF_SIZE]; /* on dual-port RAM */

        ioctl_packet.ioctl_command = TCDLOAD;
        ioctl_packet.parameter_block.dl_info.host_address =
                                                download_buffer;
        ioctl_packet.parameter_block.dl_info.ipc_address =
                                                firmware_address;
        ioctl_packet.parameter_block.dl_info.count = BUF_SIZE;

        while (count > 0)
        {
                copy(task_buffer, download_buffer,
                                minimum(count, BUF_SIZE));

                bpp_send(&channel, &ioctl_packet, WAIT);
                if (ioctl_packet.error != 0) {
                        invalid_device_or_address();
                        return(ioctl_packet.error);
                }
                count -= BUF_SIZE;
                task_buffer += BUF_SIZE;
                ioctl_packet.parameter_block.dl_info.ipc_address +=
                                BUF_SIZE;
        }
}
```

FIGURE 8-28. TCDLOAD COMMAND EXAMPLE

8.7.11 TCLINE Command

The TCLINE command instructs the firmware to copy a line discipline table, previously downloaded by TCDLOAD command, to its internal table, so that the new line discipline routines can take affect. The table consists of several set of pointers pointing to routines that are called by the OCP, ICP, GATE, or IOCTL process to perform some specific translation for the device. A set 0 is called line discipline 0, and so forth.

The ipc_address field in the dl_info structure of the packet parameter block specifies the address of the line discipline table. The count field specifies the number of sets in the table (or the number of lines, not the number of bytes).

The firmware also makes sure that the address specified in the ipc_address field is in range of the downloadable area before copying the table. Otherwise, the EINVAL error code will be returned and no transfer will take place.

For security reasons, the device number is also restricted to the printer device which is number 8, otherwise, the EINVAL error code is returned.

Figure 8-29 illustrates an example of how to use this command.

```
ioctl_line_switch_table(dev, table_address, nlines)
char *table_address; /* MVME332XT downloaded table address */
int  nlines;         /* number of line disciplines */
{
        ioctl_packet.ioctl_command = TCLINE;
        ioctl_packet.parameter_block.dl_info.ipc_address =
                                                table_address;
        ioctl_packet.parameter_block.dl_info.count = nlines;

        bpp_send(&channel, &ioctl_packet, WAIT);
        if (ioctl_packet.error != Ø)
                invalid_device_or_address();
}
```

FIGURE 8-29. TCLINE COMMAND EXAMPLE

8.7.12 TCEXEC Command

The TCEXEC command instructs the firmware to execute a function address previously downloaded into the local memory by the TCDLOAD command. The function must end with an RTS instruction (ReTurn from Subroutine) unless a new set of firmware is executed in place of the current one.

This command is useful for some purpose like fixing a problem in the firmware, debugging aid, or performing some special hardware control functions.

The ipc_address field in the dl_info structure of the packet parameter block specifies the address of the function. The firmware also makes sure that such an address is in the range of the downloadable area before executing. Otherwise, the EINVAL error code will be returned.

For security reasons, the device number is also restricted to the printer device which is number 8, otherwise, the EINVAL error code is returned.

Figure 8-3Ø illustrates an example of how to use this command.

```
ioctl_execute_function(dev, function_address)
int (*function_address)(); /* MVME332XT downloaded function address */
{
        ioctl_packet.ioctl_command = TCEXEC;
        ioctl_packet.parameter_block.dl_info.ipc_address =
                                                function_address;

        bpp_send(&channel, &ioctl_packet, WAIT);
        if (ioctl_packet.error != Ø)
                invalid_device_or_address();
}
```

FIGURE 8-3Ø. TCEXEC COMMAND EXAMPLE

8.7.13 TCGETVR Command

The TCGETVR command allows the host to obtain the version number and revision number of the firmware. The param field of the packet parameter block contains this information when the packet is returned. The least significant byte of the field is the revision number and the version number is in the next byte.

For security reasons, the device number is also restricted to the printer device which is number 8, otherwise, the EINVAL error code is returned.

Figure 8-31 illustrates an example of how to use this command.

```
ioctl_get_version(dev)
{
        ioctl_packet.ioctl_command = TCGETVR;

        bpp_send(&channel, &ioctl_packet, WAIT);
        if (ioctl_packet.error != Ø)
                invalid_device_or_address();

        printf("Firmware Version = %d, Revision = %d",
                ioctl_packet.parameter_block.param >> 8,
                ioctl_packet.parameter_block.param & Øxff);
}
```

FIGURE 8-31. TCGETVR COMMAND EXAMPLE

8.7.14 TCGETSYM Command

The TCGETSYM command allows the host to obtain the symbol table of the firmware in the format suitable for the UNIX linkage editor (ld) so that the host can link this with its downloadable code before downloading to the firmware.

Figure 8-32 illustrates a typical format of the symbol table returned by the firmware in the form of displayable character strings. It consists of two parts, the first part indicates where the downloadable area is and how big it is, the second part is a list of symbol definitions. The UNIX linkage editor (or loader) will use this information to resolve the undefined symbols in the user code to produce an executable code for downloading.

```
MEMORY
{
        dl_area (RW) : o = 0x00f20000, l = 0x00010000
}
SECTIONS
{
        GROUP : {
                .text {}
                .data {}
                .bss  {}
        } > dl_area
}

reset_vector = 0x00fc0000;
ctmain       = 0x00fc04ec;
......       = ..........
fwmain       = 0x00fc3dfa;
```

FIGURE 8-32. TYPICAL SYMBOL TABLE

The host specifies the address and size of the buffer to receive the symbol table in the host_address field of the packet parameter block. The buffer must reside on the dual-port memory and, if it is smaller than the table, multiple commands can be sent to satisfy a request.

It is required that the host should set the ipc_address field to zero to start with, then check this field until it becomes -1 which is the end-of-file (EOF) mark to complete a user request. When the packet is returned, this field is updated by the firmware to the index of the next entry in the symbol table so that the subsequent commands will reach the end of the table. A zero in this field instructs the firmware starts to read the beginning of the symbol table. The count

field is also modified by the firmware to indicate the number of characters returned in the buffer.

For security reasons, the device number is also restricted to the printer device which is number 8, otherwise, the EINVAL error code is returned.

Figure 8-33 illustrates an example of how to use this command.

```
ioctl_get_symbol_table(dev, task_buffer, buffer_size)
char *task_buffer;
{
    extern char dual_port_buffer[BUF_SIZE]; /* on dual-port RAM */

    ioctl_packet.ioctl_command = TCGETSYM;
    ioctl_packet.parameter_block.dl_info.ipc_address = Ø;
    ioctl_packet.parameter_block.dl_info.host_address =
                                            dual_port_buffer;
    while (buffer_size > Ø) {
         ioctl_packet.parameter_block.dl_info.count = BUF_SIZE;
         bpp_send(&channel, &ioctl_packet, WAIT);
         if (ioctl_packet.error != Ø)
                 invalid_device_or_address();
         if (ioctl_packet.parameter_block.dl_info.ipc_address == -1)
                 break;

         count = ioctl_packet.parameter_block.dl_info.count;
         copy(dual_port_buffer, task_buffer, count);
         buffer_size -= count;
         task_buffer += count;;
    }
}
```

FIGURE 8-33. TCGETSYM COMMAND EXAMPLE

8.7.15 TCWHAT Command

The TCWHAT command performs exactly the same function as the TCGETSYM command, excepts that it returns a list of the firmware files with the file version number. This command is useful to keep track of the versions of files included in the firmware.

Figure 8-34 illustrates a typical format of a file listing returned by the firmware in the form of displayable character strings.

```
@(#)close2.c        7.2
@(#)ctl2.c          7.1
@(#)event2.c        7.3
@(#)open2.c         7.4
@(#)ocp2.c          7.2
@(#)icp2.c          7.1
.......             ...
```

FIGURE 8-34. TYPICAL FILE LISTING

Figure 8-35 illustrates an example of how to use this command. The required parameters are identical to the TCGETSYM command's parameters.

```
ioctl_get_file_listing(dev, task_buffer, buffer_size)
char *task_buffer;
{
    extern char dual_port_buffer[BUF_SIZE]; /* on dual-port RAM */

    ioctl_packet.ioctl_command = TCWHAT;
    ioctl_packet.parameter_block.dl_info.ipc_address = Ø;
    ioctl_packet.parameter_block.dl_info.host_address =
                                            dual_port_buffer;
    while (buffer_size > Ø) {
         ioctl_packet.parameter_block.dl_info.count = BUF_SIZE;
         bpp_send(&channel, &ioctl_packet, WAIT);
         if (ioctl_packet.error != Ø)
                 invalid_device_or_address();
         if (ioctl_packet.parameter_block.dl_info.ipc_address == -1)
                 break;

         count = ioctl_packet.parameter_block.dl_info.count;
         copy(dual_port_buffer, task_buffer, count);
         buffer_size -= count;
         task_buffer += count;;
    }
}
```

FIGURE 8-35. TCWHAT COMMAND EXAMPLE

8.7.16 TIOCGETP Command

The TIOCGETP command returns the device's current configuration in the format of "sgttyb" structure into the packet parameter block. This command is almost identical to the TCGETA command, except that the sgttyp structure is returned instead of the termio structure. The sgttyp structure contains all informations found in a termio

structure. The firmware will convert its internal version of termio to the sgttyp structure when it receives this command.

Figure 8-36 illustrates an example of how to use this command.

```
ioctl_get_sgttyp(dev, sgttyp_pointer)
struct sgttyp *sgttyp_pointer;
{
        ioctl_packet.ioctl_command = TIOCGETP;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error)
                error_handler();
        else
                copy(&ioctl_packet.parameter_block.sgt,
                     sgttyp_pointer, sizeof(struct sgttyp));
}
```

FIGURE 8-36. TIOCGETP COMMAND EXAMPLE

8.7.17 TIOCSETP Command

The TIOCSETP command allows the host to reconfigure the device with a sgttyp structure. It will wait for the device's output rings (WRITE ring and OUTPUT ring) to drain out completely, then flush the input rings, and finally change the device configuration. As a result, the internal termio structure is also modified based on the information supplied in the sgttyp structure.

Figure 8-37 illustrates an example of how to use this command.

```
ioctl_set_sgttyp(dev, sgttyp_pointer)
struct sgttyp *sgttyp_pointer;
{
        ioctl_packet.ioctl_command = TIOCSETP;
        copy(sgttyp_pointer, &ioctl_packet.parameter_block.sgt,
                 sizeof(struct sgttyp));
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error)
                invalid_device();
}
```

FIGURE 8-37. TIOCSETP COMMAND EXAMPLE

8.7.18 LDOPEN, LDCLOSE, LDCHG, LDGETT, And LDSETT Commands

These commands exist for compatibility with old versions of UNIX. It basically allows the host to open or close a device and to get or change the virtual terminal parameters, such as the position of a cursor.

The LDGETT command returns the current cursor information in the termcb structure of the packet parameter block. Likewise, the LGSETT command requires such a structure in the same place to change the cursor position.

Figure 8-38 illustrates a use of these commands.

```
ioctl_ldsett(dev, termcb_pointer)
struct termcb *termcb_pointer;
{
        ioctl_packet.ioctl_command = LDOPEN;
        bpp_send(&channel, &ioctl_packet, WAIT);

        ioctl_packet.ioctl_command = LDSETT;
        copy(termcb_pointer, &ioctl_packet.parameter_block.tcb,
                sizeof(struct termcb));
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.error)
                invalid_device();
}
```

FIGURE 8-38. LDOPEN/LDSETT COMMAND EXAMPLES

8.7.19 TCGETDS Command

This command returns the current status of a device's hardware signals such as DCD, CTS, DTR, PR_FAULT, PR_POUT, and PR_SELECT. The param field of the packet parameter block contains this information when the packet is returned. The corresponding bits are listed in Table 8-1 of the EVENT packet.

Figure 8-39 illustrates a use of this command.

```
ioctl_get_dev_status (dev)
{
        ioctl_packet.ioctl_command = TCGETDS;
        bpp_send(&channel, &ioctl_packet, WAIT);

        if (ioctl_packet.parameter_block.param &E_DCD)
                print&("DCD is negated");
}
```

NOTES: This command is only implemented with firmware version 8.4 or later and driver version 8.3 or later.

The status of RTS and DTR are returned in the E_LOST_CTS bit and the E_LOST_DSR bit of Table 8-1, respectively.

FIGURE 8-39. TCGETDS COMMAND EXAMPLE

CHAPTER 9
CONFIDENCE TEST

9.1 INTRODUCTION

The confidence test is called upon the system reset to perform a set of basic tests to ensure that the hardware is in good condition for the firmware to run. The test does not require any external cables or any special configuration except power. The information about the tests can be easily obtained by the host in the dual-port memory.

9.2 TEST FEATURES

The following features will be included in the confidence test:

1. Loop continuously on error.
2. Confidence Test Descriptor consists of Composite Status Word (CSW), Failed address, Expect pattern, and Read pattern.
3. Subtests for better fault isolation.
4. Burn-In test sequence.
5. Warm-start detection.

9.3 TEST FLOW

The confidence test is divided into several subtests which are sequentially executed. If an error occurs in any subtest, the subtest composes its error information into a word, writes it to the Composite Status Word of the Confidence Test Descriptor, displays it on the LEDs, sends it to the debug port (if a diagnostic board is connected), and then repeats the test forever until reset or power down.

Error information may be obtained by examining the contents of the composite status word with an off-board debugger or by observing it on the terminal connected to the diagnostic port.

As illustrated in Figure 9-1, the confidence test consists of the following subtests:

Basic CPU	Extended CPU
Local RAM	SIO (MK68564)
Local ROM	PIT (MC6823Ø)
Dual-Port RAM	On-Board CSR

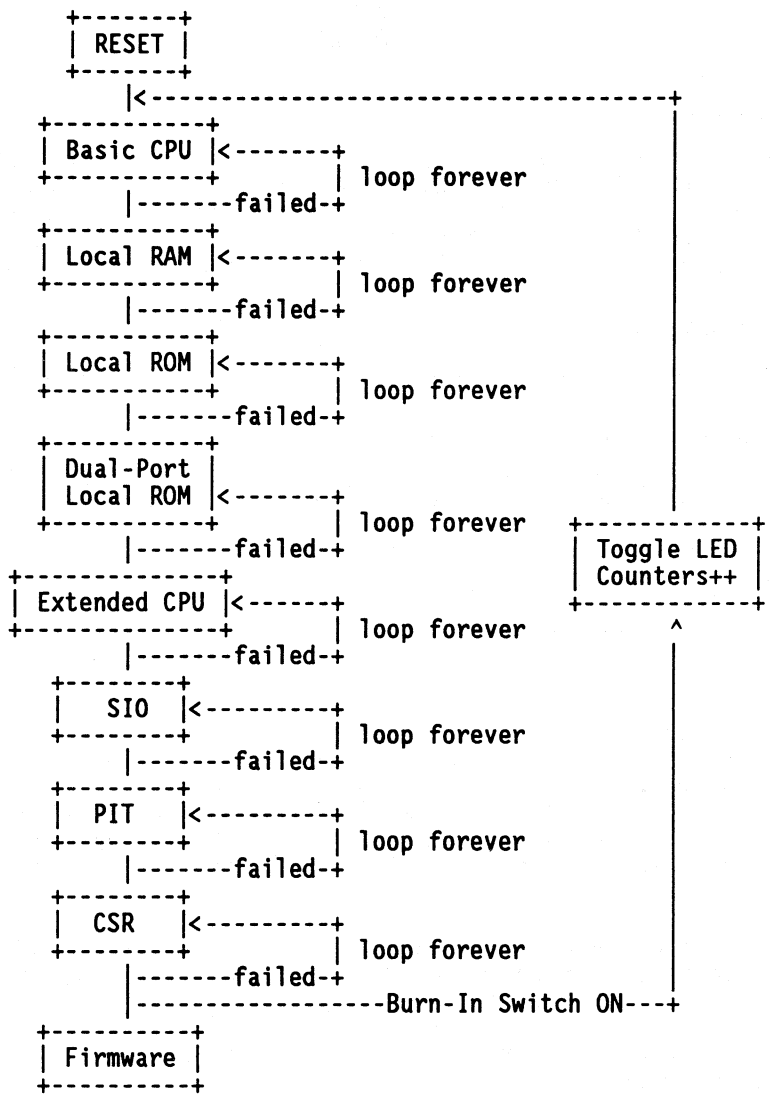


FIGURE 9-1. CONFIDENCE TEST FLOW

The details of these subtests are described in the following sections. Tests may be in turn divided into many sub-modules in the order of simple to complex functions for better fault isolation. The complex tests may rely on the simple test to minimize redundant code and time.

Note that the only way to turn off the **FAIL** LED is to successfully complete the confidence test without an error.

At the end of the test sequence, if switch S2 on the board is set to the burn-in mode, the test will toggle the LED to indicate it is alive, increment the loop counter in the Confidence Test Descriptor, then repeat the test again. During the burn-in test, if any error occurs, the LED will be lit steadily and the test will loop forever on the failed sub-test. But if the LED is completely blank, this means the local CPU is out of sequence due to a catastrophic hardware condition.

9.4 TEST OUTPUT DISPLAY

Prior to executing any test, the test number is sent to the terminal attached to the debug port of a diagnostic board. It is also written into the Composite Status Word of the Confidence Test Descriptor on the dual-port RAM.

If any test fails during the sequence, its description will also be displayed on both terminal and the dump area as shown below:

```
21ØØ Ram Walking bit  failed at ØØf21ØØØ expected=ØØØ8ØØØØ,
read=ØØØØØØØØ loop until reset
```

FIGURE 9-2. TYPICAL CONFIDENCE TEST ERROR MESSAGE

9.5 CONFIDENCE TEST DESCRIPTOR

The Confidence Test Descriptor is a reserved area on the dual-port memory after the IPC_CSR space (at offset $1Ø of the board base address). It is a structure that the confidence test uses to update the information regarding failures occurring during the test, or the number of repeated loops, so that when a failure occurs, a user can obtain this information by using an off-board debugger command.

Figure 9-3 illustrates the confidence test descriptor. It consists of the Composite Status Word (CSW), Loop Counter (LC), Fault Address (FA), Expect Data, and Read Data. The details of each field are described in the following sections.

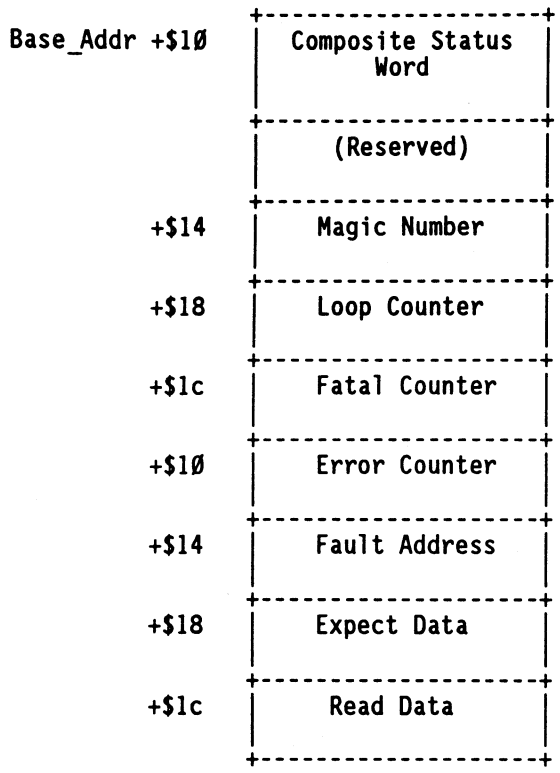


FIGURE 9-3. CONFIDENCE TEST DESCRIPTOR

9.5.1 Composite Status Word

The composite status word is a 16-bit value that is updated by each subtest to indicate a subtest has started, successfully completed, or failed. It will provide information such as subtest number, submodule number, and status number. Its format is illustrated in Figure 9-4.

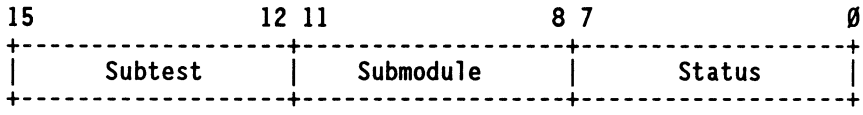


FIGURE 9-4. COMPOSITE STATUS

TABLE 9-1. SUBTEST/SUBMODULE NUMBER

Sub-Test	Sub-Module	Subtest Name	Sub-Module Name
Ø	Ø	RESET	
1	1	Basic CPU	Simple Instruction
	2		Complex Instruction
2	1	RAM	Walking Bit
	2		Byte/Word/Long
	4		March
3	1	ROM	CRC checking
4	1	Dual-Port RAM	Walking Bit
	2		Byte/Word/Long
	4		March
5	2	Extended CPU	Complex Addressing
	3		Exception
6	1	MK68564	Register Test
	2		Tx/Rx Polling
	3		Tx/Rx Interrupt
	4		Baud Rate
7	1	MC6823Ø	Register Test
	2		Timer Counters
	3		Timer Interrupt
	4		Printer Interrupt
8	1	CSR	Register Test
	2		Attention Interrupt

SUBTEST

This field should be cleared upon the RESET. A non-zero number indicates a subtest has started, the subtest names are provided in Table 9-1.

SUBMODULE

This field should be cleared upon the RESET. A non-zero number indicates a submodule has started, the submodule names are provide in Table 9-1.

STATUS

This field indicates a status of a sub-module. Each sub-module will use this field to supply additional information, such as the subroutine the code is currently executed, or the port number. A ZERO always indicates a successful sub-module (no error).

9.5.2 Magic Number

This field is used to detect the warm/cold start condition. If the test finds a predefined magic pattern in this field, it will bypass the whole sequence of the confidence test unless the BURN-IN condition is set. At the current time, the string of "W332" is written into this field when the test completes at the first time.

9.5.3 Loop Counter

The loop counter field is a longword (32 bits) and is used to count the number of loops that the confidence test has passed. It is useful for burn-in test since the burn-in sequence repeats the confidence test indefinitely.

9.5.4 Fatal Counter

This field records the number of fatal errors the test encounters during testing.

9.5.5 Error Counter

This field records the number of non-fatal or soft errors the test encounters during testing.

9.5.6 Failed Address

The Failed-Address field is used to indicate the target test location that cause the failure during the test.

9.5.7 Expect Data

This field is used to indicate the data that the test is expecting as a result of the test.

9.5.8 Read Data

This field indicates that the data has obtained from the test. If it is different from the "Expect Data" field, the test fails.

9.6 BASIC CPU TEST

This is the first test of the confidence test sequence, it is called upon RESET. The purpose of this test is to ensure that CPU is able to execute a minimum set of instructions which may be used in the next level of sub-test or sub-module. The test should never attempt to access any RAM space. All instructions should use the CPU's internal registers to manipulate its data, if necessary.

This test is in turn divided into two sub-modules: simple instruction set and complex instruction set.

9.6.1 Simple Instruction

The simple instruction set consists of ADD, SUB, OR, AND, EXOR, NEG, CLR, NOT, SWAP, TST, NOP, MOVE, MOVE, CMP, EXG, LEA, and BRANCH. Those instructions are organized in such a manner as to produce a final result by using all internal registers as data. This final result is checked to determine whether the test has passed or failed.

9.6.2 Complex Instruction

The complex instruction set consists of SHIFT, MUL, DIV, DBcc, Scc, and MOVEC. Once again, those instructions are organized in such a manner as to produce a final result by using all internal registers as data. This final result is checked to determine whether the test has passed or failed.

9.7 LOCAL RAM TEST

This is the second test of the confidence test. It will be executed before any other tests that use RAM for variables. Since a user may want to examine the information left over from the last operation, the test is restricted to write to a specific area allocated at the link time, called TEST-RAM.

9.7.1 Walking Bit

This test mainly checks the circuitry related to all data bit lines from the CPU to the on-board RAM, such as data paths, input/output buffers etc. It is intended to check the data paths from the CPU to the RAM array only, not every bit of the RAM array; therefore, a small portion of the TEST-RAM is tested.

9.7.2 Byte/Word/Long

The purpose of this test is to check out all the signals such as DTACK and the board's decoder when the CPU accesses to the RAM. Any byte/word swapping will be detected in this test.

9.7.3 March

The main purpose of this test is to check the circuitry associated with all address lines, such as address paths, address muxes/buffers, etc. Its algorithm exercises every bit of the TEST-RAM in both states, low and high.

9.8 LOCAL ROM TEST

Due to a large amount of software stored in the ROMs, a CRC checking is employed in this test. This algorithm will detect more error bits and byte swapping at very high percentage. The last four bytes of the ROM space are reserved to program the CRC code. This code will be used to compare with the result of CRC calculation of the rest of ROM to determine if the test passes or fails.

9.9 DUAL-PORT RAM TEST

This test consists of a set of subtests as described in the local RAM test, but the target testing is the entire area of the dual-port RAM except the IPC_CSR and CTDES.

9.10 EXTENDED CPU TEST

This test is the second part of the CPU test, it attempts to execute several instructions or addressing modes that may be required to access the RAM.

9.10.1 Complex Addressing

The complex addressing mode is illustrated in Table 9-2. The test may organize its data in such an order as to produce a final result which is used to check against a known value to determine if the test has passed or failed. For more information on the addressing modes, refer to the "Effective Address" section of the MC68010 User's Manual.

TABLE 9-2. ADDRESSING MODES

Parameter	Description
(An)+	Address Register Indirect with Post_increment.
-(An)	Address Register Indirect with Pre_decrement.
(bd,An,Xn)	Address Register Indirect with Index and Base Displacement.
(bd,PC,Xn)	PC Indirect with Index and Base Displacement.

9.10.2 Exception

This test executes several instructions that cause the exception operations to occur and ensure that the CPU is able to handle such situations in a proper way. The exceptions tested are TRAP, ADDRESS-ERROR, BUS-ERROR, A-LINE and F-LINE Instructions, Illegal Instructions, and RTE.

9.11 MK68564 TEST

The purpose of this test is to check out the functionality of the MK68564 Serial Input/Output (SIO) chip in a loop-back mode, including register test, transmit character, receive character, TX interrupt, and RX interrupt.

9.11.1 MK68564 Register Test

The SIO chip consists of 32 internal registers, most of which are writable/readable. This test will perform a walking-bit test on each register to check out the interface between the CPU and SIO and ensure that data can be stored or retrieved from those registers.

9.11.2 MK68564 Tx/Rx Polling

This test verifies that a character stream sent to the transmitter of the SIO chip can be received on the receiver via an internal loop back. The CPU will poll the status bit in the SIO status register to determine the character coming, the interrupt is masked until another test.

9.11.3 MK68564 Tx/Rx Interrupt

The main purpose of this test is to verify that the interrupts generated by the SIO are seen by the CPU. This test may fail because the SIO's IRQ line is shorted or open, the interrupt vector is wrong, the interrupt acknowledge cycle is bad, or the device is faulty. The interrupts occur when the transmitter becomes empty, or when the receiver gets a character in its FIFO. The test should distinguish and verify the condition for each one.

9.12 MC68230 TEST

This test verifies the functionality of the MC68230 Parallel Interface/Timer (PIT) device. It consists of a register test, counter test, printer interrupt test, and timer interrupt test.

9.12.1 MC68230 Register Test

The walking bit test is performed by this test on every writable/readable bit of the PIT's register set to check out the interface logic between the CPU and the PIT.

9.12.2 MC68230 Counter Test

This test ensures that all counters are able to count down from a preloaded value. The counters are linked together so that when they reach all zeros, an interrupt condition will be generated, and an interrupt is issued if it is unmasked.

9.12.3 M6823Ø Timer Interrupt

In this test, the timer interrupt is enabled and the counter is loaded to count down as in the above test to assert its IRQ signal. This test ensures that the IRQ signal can be seen by the local CPU.

9.12.4 MC6823Ø Printer Interrupt

In this test, the printer output lines are isolated from the printer connector so that the test pattern will not be send to the printer device. When data written into the printer double registers is transmitted, a printer interrupt is generated. The local CPU will verify that it can see such an interrupt.

9.13 CSR TEST

There are a few readable/writable control registers on the MVME332XT processor. This test will perform a walking-bit test on every one of them. It will skip several bits that are not changeable.

9.14 ATTENTION INTERRUPT TEST

This test ensures that a ONE in the ATTENTION bit of the IPC_CSR will generate a level 1 interrupt to the local CPU. It also verifies that the ATTENTION_MASK bit in a control register is able to mask such an interrupt.

CHAPTER 10
LINE DISCIPLINES

10.1 INTRODUCTION

There are several major line disciplines that are currently supported by the on-board firmware to allow the user to select the features needed for his/her application. Each line can be selected at the system initialization or by using "ioctl" system calls (see termio(7)). All lines can be downloaded by the host to perform user application specifics. The firmware uses an internal line switch table on its local memory to allow the host to add or to replace any entry.

10.2 LINE 0 - STANDARD UNIX LINE DISCIPLINE

Line 0 is a standard UNIX line discipline 0. It conforms to all features specified in the termio(7) and stty(1) commands as described in the SYSTEM V/68 documentation. It consists of:

1. Output processing: character mapping and character filling.
2. Input processing: character mapping, software handshaking, and character signaling.
3. Control and Local mode: baud rate, character size, echoing, hardware handshaking, and parity control.
4. Virtual Terminal handling for 7 terminal types: Tec Scope, Dec vt61, Dec vt100, Tektronix 4023, TTY Mod 40/1, Hewlett-Packard 45, and TTY Mod 40/2B.

10.3 LINE 1 - PURE RAW LINE DISCIPLINE

Line 1 is a scaled down version of line 0, but it is faster since no input and output translations are performed. It is useful for applications that use raw data. It only checks for software handshake (XON/XOFF) and signaling characters. It supports:

1. Input processing: software handshaking and character signaling.
2. Control and Local mode: baud rate, character size, hardware handshaking, and parity control.

10.4 LINE 2 - INTERNATIONAL SUPPORT PACKAGE (ISP) LINE DISCIPLINE

This line is a superset of line 0, but it includes a capability to translate from one language to another for the ISP. It allows the user to download a language translation table conforming to the XEROX character set standard to the local RAM for subsequent output

or input translation. This line is slower than the above lines since it has to perform table look-up for every input or output character.

10.5 LINES 3 THROUGH 6 - USER DOWNLOADABLE LINE DISCIPLINE

These lines are available for the user who needs to perform a specific application that does not fit into the UNIX standard I/O. These lines are only accessible after the new line switch table consisting of their entries is downloaded into the local RAM with the TCDLOAD and TCLINE commands.

10.6 LINE SWITCH TABLE

As mentioned earlier, the firmware uses a table to switch from one line to another. This table can be downloaded by the host to replace or add entries in the table. It consists of seven sets of pointers pointing to functions that perform character processing for input, output, and control flow.

Each function is passed a pointer to the device **tty structure** containing all information about the device including the termio structure, INPUT ring buffer, OUTPUT ring buffer, READ ring pointer, WRITE ring pointer, and user buffers. In addition, a pointer to the received envelope is also passed to three routines specified in fields **l_open**, **l_ioctl**, and **l_close**. The details of the **tty structure** will be discussed in the next section.

Figure 10-1 illustrates the format of a typical line switch table. A zero in each field instructs the firmware to use its default pointer which resides in the on-board ROMs. A non-zero pointer points to the downloaded user's functions.

	Open	Icp	Ocp	Ioctl	Close	Ctl	Gate
Line Ø	openØ	icpØ	ocpØ	ioctlØ	closeØ	ctlØ	gateØ
Line 1	openØ	icp1	ocp1	ioctlØ	closeØ	ctlØ	gateØ
Line 2	open2	icp2	ocp2	ioctl2	close2	ctl2	gate2
Line 3	open3	icp3	ocp3	ioctl3	close3	ctl3	gate3
Line 4	Ø	Ø	Ø	Ø	Ø	Ø	Ø
Line 5	Ø	Ø	Ø	Ø	Ø	Ø	Ø
Line 6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

```
struct line_switch {          /* C declaration                */
       int  (*l_open)();      /* open function pointer        */
       int  (*l_icp)();       /* ICP process pointer          */
       int  (*l_ocp)();       /* OCP process pointer          */
       int  (*l_ioctl)();     /* ioctl function pointer       */
       int  (*l_close)();     /* close function pointer       */
       int  (*l_ctl)();       /* CTL process pointer          */
       int  (*l_gate)();      /* GATE process pointer         */
} linesw[7];
```

FIGURE 10-1. LINE SWITCH TABLE FORMAT

10

The following is a description of each field of the line switch table.

l_open

This field contains a pointer pointing to an OPEN sub-routine that is called by the **Bpp_Receiver** when an OPEN packet arrives. It performs functions necessary for the open sequence such as asserting the device RTS and DTR, enabling the device interrupt, or initializing a data structure.

Note that the open command is not a process, it is executed in the context of the attention interrupt so that only few kernel calls can be executed in this routine, such as **_signal** and **_wakeup**. The **_wait** and **_sleep** calls are prohibited since an innocent running process can be put to sleep forever. To wait or to sleep on a certain event, it queues the OPEN packet to the GATE process' packet queue, then wakes the process up so that the process can call the kernel primitives.

Figure 10-2 illustrates a typical call to this sub-routine from the **Bpp_Receiver** and the open routine itself (assume that line-3 is selected).

```
Bpp_Receiver()
{
        envelope = get_envelope(&channel);
        packet = envelope->packet_pointer;
        tty = &ttytab[packet->device_number];
        if (packet->command == OPEN)
                (*linesw[tty->t_tio.c_line].l_open)(tty, envelope);
}

open3(tty, envelope)                    /* line-3 open() example       */
struct tty *tty;
struct envelope *envelope;
{
         if (need_to_wait) {            /* hand it to the GATE process */
                queue_envelope(&tty->t_gate_env, envelope);
                _signal(&tty->t_gate_sem, PREEMPTED);
         } else
                device_open(tty);       /* enable the device           */
}
```

FIGURE 10-2. L_OPEN EXAMPLE

l_icp

The function pointed to by this field is executed in the context of the ICP process to perform input processing for the characters received in the device INPUT ring. The results of this operation will be put in the READ ring on the dual-port RAM ready for the host to use.

The ICP process is awakened by the **Timer_Handler** when there is a character in the associated INPUT ring. Upon running, the ICP process calls this function, then goes back to sleep waiting for the next signal.

Figure 10-3 illustrates a typical call to this sub-routine from the **ICP Process** and the icp routine itself (assume that line-3 is selected).

```
ICP()                                   /* ICP process                  */
{
        tty = gettty();
        for (;;) {                              /* forever loop         */
                wait(&tty->t_icp_sem);  /* wait for a signal            */
                (*linesw[tty->t_tio.c_line].l_icp)(tty);
        }
}

icp3(tty)                               /* line-3 icp() example         */
struct tty *tty;
{
        c = get_character(&tty->t_iring);   /* from INPUT ring          */
        c = translation(c);
        put_character(&tty->t_rring, c);    /* put into READ ring       */
        /* check for any one waiting for character */
        if (tty->t_read_wakeup) {
                envelope = dequeue_envelope(&tty->t_read_wakeup);
                bpprtn(envelope);               /* return the packet    */
        }
}
```

FIGURE 10-3. L_ICP EXAMPLE

l_ocp

The function specified in this field is called by the OCP process to perform output processing for the characters written by the host in the WRITE ring. The result of this operation will be put in the associated OUTPUT ring ready for transmitting to the port.

Same as the ICP process, the OCP process is awakened by the **Timer_Handler** when there is a character in the WRITE ring. At the completion of the function call, it goes back to sleep waiting for the next signal.

Figure 10-4 illustrates a typical call to this sub-routine from the **OCP Process** and the ocp routine itself (assume that line-3 is selected).

```
OCP()                                   /* OCP process                  */
{
        tty = gettty();
        for (;;) {                              /* forever loop         */
                _wait(&tty->t_ocp_sem);         /* wait for a signal    */
                (*linesw[tty->t_tio.c_line].l_ocp)(tty);
        }
}

ocp3(tty)                               /* line-3 ocp() example         */
struct tty *tty;
{
        c = get_character(&tty->t_wring);  /* from the WRITE ring  */
        c = translation(c);
        put_character(&tty->t_oring, c);   /* put into OUTPUT ring */
        /* check for any one waiting for character                   */
        if (low_water_mark && tty->t_write_wakeup) {
                envelope = dequeue_envelope(&tty->t_write_wakeup);
                bpprtn(envelope);               /* return the packet    */
        }
}
```

FIGURE 10-4. L_OCP EXAMPLE

l_ioctl

The sub-routine in this entry is called by the *Bpp_Receiver* when an IOCTL packet is received to carry out various functions for the IOCTL system calls. Some system calls such as TCGETA, TCGETDL, or even TCSETA can be performed quickly in this routine since they do not have to wait for any conditions. But some such as TCSETAW or TCSETAF will be handed off to the **CTL process** since they have to wait for the output rings to drain out completely before changing the device configuration.

Figure 10-5 illustrates a typical call to this sub-routine from the **Bpp_Receiver** and the ioctl routine itself (assume that line-3 is selected).

```
Bpp_Receiver()
{
        if (packet->command == IOCTL)
             (*linesw[tty->t_tio.c_line].l_ioctl)(tty, envelope);
}

ioctl3(tty, envelope)                   /* line-3 ioctl() example      */
struct tty *tty;
struct envelope *envelope;
{
        if (need_to_wait) {             /* hand it to the CTL process  */
                 queue_envelope(&tty->t_ctl_env, envelope);
                 _signal(&tty->t_ctl_sem, PREEMPTED);
        } else
                 decode_ioctl_command(envelope, tty);
}
```

FIGURE 10-5. L_IOCTL EXAMPLE

l_close

This field is the counter part of the **l_open** field. It is called by the **Bpp_Receiver** when the CLOSE packet arrives to disable a device interrupt or negate the device modem signals.

If the close needs to wait for the OUTPUT rings to drain out, the routine transfers this packet to the GATE process, then notifies the process by calling the **_signal** kernel primitive.

Figure 10-6 illustrates a typical call to this sub-routine from the **Bpp_Receiver** and the close routine itself (assume that line-3 is selected).

```
Bpp_Receiver()
{
        if (packet->command == CLOSE)
              (*linesw[tty->t_tio.c_line].l_close)(tty, envelope);
}

close3(tty, envelope)                  /* line-3 close3() example     */
struct tty *tty;
struct envelope *envelope;
{
        if (need_to_wait) {            /* hand it to the GATE process */
                queue_envelope(&tty->t_gate_env, envelope);
                _signal(&tty->t_gate_sem, PREEMPTED);
        } else
                device_close(tty); /* disable the device              */
}
```

FIGURE 10-6. L_CLOSE EXAMPLE

l_ctl

This field points to a function called by the CTL process to perform any IOCTL commands that wait for certain conditions. The CTL process is awakened by the sub-routine specified in the **ioctl** field as a result of calling the **_signal** kernel primitive. Upon running, the CTL process calls the function, then returns to sleep by invoking the **_wait** kernel primitive.

This function, when running, gets a packet from the IOCTL packet queue in the **tty** structure, decodes the command, then executes it.

Figure 10-7 illustrates a typical call to this sub-routine from the **CTL Process** and the gate routine itself (assume that line-3 is selected).

```
CTL()                                   /* CTL process                  */
{
        tty = gettty();
        for (;;) {                              /* forever loop         */
                _wait(&tty->t_ctl_sem);         /* wait for a signal    */
                (*linesw[tty->t_tio.c_line].l_ctl)(tty);
        }
}

ctl3(tty)                               /* line-3 ctl() example         */
struct tty *tty;
{
        envelope = dequeue_envelope(&tty->t_ctl_env);
        ttywait(tty);   /* wait for output rings to drain               */
        if (envelope->packet_pointer->command == TCSETAW) {
                ttyflush(F_READ);
        (*tty->t_param)(tty);                   /* reconfigure the device*/
        bpprtn(envelope);                       /* return the packet    */
}
```

FIGURE 10-7. L_CTL EXAMPLE

l_gate

This field is called by the GATE process upon a signal from the routines specified in the Open or Close field. When the GATE process has a chance to run, it calls the function and goes back to sleep.

The function gets a packet from the GATE packet queue in the device **tty structure**, decodes the command, executes the command, and finally returns the packet back to the host by calling the firmware function **bpprtn()**.

Figure 10-8 illustrates a typical call to this sub-routine from the **GATE Process** and the gate routine itself (assume that line-3 is selected).

```
GATE()                                  /* GATE process                 */
{
        tty = gettty();
        for (;;) {                      /* forever loop                 */
                wait(&tty->t_gate_sem);  /* wait for a signal           */
               (*linesw[tty->t_tio.c_line].l_gate)(tty);
        }
}

gate3(tty)                              /* line-3 gate() example        */
struct tty *tty;
{
        envelope = dequeue_envelope(&tty->t_gate_env);
        ttywait(tty);   /* wait for output rings to drain               */
        if (envelope->packet_pointer->command == OPEN) {
                device_open(tty);       /* enable device                */
        else
                device_close(tty);      /* disable device               */
        bpprtn(envelope);               /* return the packet            */
}
```

FIGURE 10-8. L_GATE EXAMPLE

10.7 TTY STRUCTURE

Associated with each device, the firmware statically allocates in its local RAM a data structure called **tty structure**, that contains all information about the device. A pointer to such a structure is passed to every function specified in the line switch table so that the function is able to obtain the location of the device ring buffers and other information.

Some fields in the **tty structure** are reserved for the on-board lines. Some are used by the device interrupt handler and some are available to the user for any purpose.

Figure 10-9 illustrates a **tty structure** in C language style. It consists of a pointer to a function that handles the device dependent functions (t_proc) and a pointer to the device configuration sub-routine (t_param). Details of those routines are also discussed in this section.

```
struct tty {
   unsigned char   t_state;          /* internal state                */
   unsigned char   t_cstate;         /* control state                 */
   struct   termio t_tio;            /* terminal control block        */
   unsigned short  t_devnum;         /* logical device number         */
   unsigned short  t_event;          /* event states                  */
   unsigned short  t_pending_event;  /* event pending flag            */
   unsigned char   reserved_0[22];

   int           (*t_proc)();        /* routine for device functions*/
   union
         MK564     *sio_addr;        /* pointer to Serial Chip        */
         PIT       *pit_addr;        /* or pointer to Printer Chip    */
   } t_devaddr;                      /* device address                */
   unsigned char   reserved_1[512];
   unsigned char   t_ract[256];      /* receive character action table */

   unsigned short  t_wsize;          /* WRITE ring size = real-1      */
   unsigned short  t_lwm_w;          /* WRITE ring Low_Water_Mark     */
   unsigned short  t_rsize;          /* READ ring size = real-1       */
   unsigned short  t_lwm_r;          /* READ ring Low_Water_Mark      */
   struct   ring  *t_wring;          /* WRITE ring pointer            */
   struct   ring  *t_rring;          /* READ ring pointer             */
   struct   ring   t_oring;          /* OUTPUT ring structure         */
   struct   ring   t_iring;          /* INPUT ring structure          */

   int           (*t_param)();       /* routine for device functions*/
   unsigned char   reserved_2[28];
   struct   termio t_def_tio;        /* open's default termio         */

   unsigned char   t_user[1024];     /* user buffer, not used by FW */

   unsigned short  t_write_wakeup;   /* WRITE_WAKEUP envelope queue header*/
   unsigned short  t_read_wakeup;    /* READ_WAKEUP envelope queue header */
   unsigned short  t_event_env;      /* EVENT envelope queue header */
   unsigned short  t_ctl_env;        /* IOCTL envelope queue header */
   unsigned short  t_gate_env;       /* GATE envelope queue header  */

   SEMAPHORE       t_icp_sem;        /* semaphore for ICP process   */
   SEMAPHORE       t_ocp_sem;        /* semaphore for OCP process   */
   SEMAPHORE       t_ctl_sem;        /* semaphore for CTL process   */
   SEMAPHORE       t_gate_sem;       /* semaphore for GATE process  */

   unsigned char   t_init_data[64]; /* inited space                 */
   unsigned char   t_avail[0x108]   /* room for future expansion    */

} ttytab[9];
```

FIGURE 10-9. TTY STRUCTURE FORMAT

The following is a discussion of each field.

t_state

This field contains the current state of the device. Some bits are set by the open routine and some are set to instruct the device interrupt handler to send an XON or XOFF character. Table 10-1 provides the bit definitions of this field.

TABLE 10-1. T_STATE BIT DEFINITION

Name	Value	Description
CARR_ON	0x01	Software copy of carrier-present.
TTENABLE	0x02	Output is enabled, else disabled.
TTXON	0x04	Send an XON character.
TTXOFF	0x08	Send an XOFF character.
NOT_EMPTY	0x20	OUTPUT ring is not empty.
TBLOCK	0x40	Remote transmitter is being blocked.
TTBUSY	0x80	Device transmitter is busy.

t_cstate

This field contains the information about the open and close commands. Table 10-2 provides the bit positions of this field.

TABLE 10-2. T_CSTATE BIT DEFINITION

Name	Value	Description
ISOPEN	0x01	Device is being opened.
NOCLOSE	0x02	Do not close a device.
WCLOSE	0x04	Close is pending or in progress.
HWFC	0x08	Hardware handshake is enabled.

t_tio

This field is a working termio structure for a device. The device is configured based on the information in this structure. Initially, it is a copy of the default termio structure t_def_tio when the first open is called, but can be modified by some IOCTL commands such as TCSETA, TCSETAF, TCSETDF, and TCSETP.

t_devnum

The device number associated with this tty structure is recorded in this field. Device number Ø to 7, and 8 correspond to serial port Ø to 7, and the printer port. The device dependent handler uses this field to detect the printer device for special handling.

t_event and t_pending_event

The **t_event** field is set by the device interrupt handler to indicate an event. The **Timer_Handler** will wake up the ICP process associated with this tty structure to return the event back to the host in the device's event packet. But if there is no event packet since it is pending on the host side, the ICP routine should save all events into the **t_pending_event** field and clear it to zero so that when the **Bpp_Receiver** receives the EVENT packet, it can return such pending events back to the host immediately.

For more information about the event codes, refer to the EVENT packet section.

t_proc

This field is a pointer to a device dependent function to control the physical device such as the serial chip (MK68564) or the printer chip (MC6823Ø). The calling of this function has a format as below:

```
(*tty->t_proc)(tty, command);
```

Table 1Ø-3 is a list of all commands supported in the serial driver.

TABLE 1Ø-3. T_PROC COMMANDS

Name	Value	Description
T_OUTPUT	Ø	Enable the device transmitter to output characters.
T_TIME	1	Stop sending BREAK sequence.
T_SUSPEND	2	Suspend the output, disable the transmitter.
T_RESUME	3	Resume the output, re-enable the transmitter.
T_BLOCK	4	Send an XOFF, negate RTS if hardware handshake enabled.
T_UNBLOCK	5	Send an XON, assert RTS if hardware handshake enabled.
T_RFLUSH	6	Flush both INPUT ring and READ ring, and send an XON if previous character was an XOFF.
T_WFLUSH	7	Flush both OUTPUT ring and WRITE ring, and resume the transmitter.
T_BREAK	8	Start sending a BREAK sequence until the T_TIME command is called.
T_PARM	11	Indirectly call device configuration routine (i.e., m564param()).

t_devaddr

Depending on the value of the **t_devnum** field, this can be a pointer to the printer device which is the base address of the MC6823Ø device, or to the serial MK68564 device. The device dependent routines use this field to interface to the device. It is set up by the board initialization routine.

t_ract[]

This is a Receive Action Table and is used by the device interrupt handler to act on a received character. For example, if a character is an XOFF, it disables the device transmitter to prevent further output. The device interrupt handler uses the value of the character to index into the table to find the action code. The action codes are listed in Table 1Ø-4.

TABLE 10-4. T_RACT[] ACTION CODES

Name	Value	Description
A_NOP	0	No special action on the character.
A_XOFF	1	Suspend the output, the received character is an XOFF.
A_XANY	2	Resume the output on any received character if the output is suspended and the XANY bit set in the termio structure.
A_XON	3	Resume the output, the received character is an XON.
A_INTR	4	Set the INTR event bit (E_INTR) in the tp-> t_event, the received character is an INTERRUPT character.
A_QUIT	5	Set the QUIT event bit (E_QUIT) in the tp-> t_event, the received character is a QUIT character.
A_SWTCH	6	Set the SWTCH event bit (E_SWTCH) in the tp-> t_event, the received character is a SWITCH character.

Note that the ioctl routine or process should set up this table based on the information in the termio structure. For example, if a QUIT character is changed, the ioctl routines may include the following lines of C code:


```
tty->t_ract[old_quit_character] = A_NOP;
tty->t_ract[tp->tio.c_cc[VQUIT]] = A_QUIT;
```

t_wsize

This field contains the mask bits for the WRITE ring. It is used as a modular number for the ring index to guarantee that the index into the ring is always in range. For example, the following C code shows how to get a character from a WRITE ring by using the **get** index:

```
c = tty->t_wring->data[tty->t_wring->get++ & tty->t_wsize];
```

This field is equal to the size of the ring minus 1. Since the ring size is a power of 2, minus 1 yields a mask field. It is set up when the INIT packet is received.

t_lwm_w

The WRITE ring **low water mark** is calculated when the INIT packet is received, then saved in this field for later use in the OCP process. It is equal to a quarter of the WRITE ring size.

t_rsize

This field is identical to the **t_wsize** field, but is used for READ ring operations.

t_lwm_r

This is the low water mark for the READ ring. It is set to half of the ring size.

t_wring and **t_rring**

The WRITE ring and READ ring *pointers* specified in the INIT packet are converted to the local accessible addresses and saved in these fields, respectively. They always point to the dual-port memory.

t_oring and **t_iring**

These are the device's OUTPUT ring structure and INPUT ring structure. They are currently equal in size (2Ø48 bytes) and reside in the local RAM since the **tty structure** is in the local RAM.

t_param

This is a pointer to a function that configures a device based on the information in the **t_tio** field such as baud-rate, character size, number of parity bits, even or odd parity, hardware handshake, or hangup a device if the baud-rate is Ø. The calling convention is shown below:

```
(*tty->t_param)(tty);
```

t_def_tio

This is a default open termio that will be copied to the t_tio field when the first OPEN packet is received. It can be obtained or changed with the TCGETDF or TCSETDF command, respectively.

t_user[]

This block is free for user buffers or data structures. It is initialized to zero at the board initialization and never touched again by the firmware.

t_write_wakeup

This field contains an index into the first envelope of the WRITE_WAKEUP packet received by the **Bpp_Receiver** on the dual-port memory. A zero indicates that there is no packet pending in the queue.

Notice that when accessing the envelope, the OCP process translates the index to a local accessible address by adding it to the base address of the local dual-port memory address ($f30000). Likewise, to access the next envelope, the OCP uses only the lower word (bit 15-0) of the link pointer in the current envelope to translate to the address of the next envelope in the queue.

t_read_wakeup, t_even_env,

t_ctl_env, and t_gate_env

These fields are the same as the **t_write_wakeup** field, but for the READ_WAKEUP, EVENT, IOCTL, and OPEN or CLOSE packet queue.

t_icp_sem, t_ocp_sem,

t_ctl_sem, and t_gate_sem

These are the semaphores that the ICP, OCP, CTL, and GATE processes sleep on. The open, close, and ioctl function specified in the line switch table uses these semaphores to wake the appropriate process up by calling the **_signal** kernel primitive. The calling convention is shown as below:

```
#define  NO_PREEMPTED     0
#define  PREEMPTED        1
_signal(&tty->t_ctl_sem, NO_PREEMPTED);
```

Figure 10-14 illustrates the format of a semaphore structure. It should be initialized to zero. If the count field becomes negative, it means that some processes are waiting for a condition on this semaphore as a result of calling the **_wait** kernel primitive. For more information about semaphore, refer to the ADC Kernel Firmware Manual.

```
typedef struct semaphore {
        short    count;          /* resource counter for semaphore */
        long     reserved0;      /* kernel use only, do not change */
        long     reserved1;      /* kernel use only, do not change */
} SEMAPHORE;
```

FIGURE 10-10. SEMAPHORE STRUCTURE

t_init_data[]

This block is a copy of the initialized data specified in the *Init Info* structure of the INIT packet. It is usable for the downloadable line disciplines to obtain a default parameter from the device driver's initialization routine.

t_avail[]

This space is free for the downloadable line disciplines. The firmware initializes it to zero and does not access it again.

10.8 FIRMWARE FUNCTION SUB-ROUTINES

Table 10-5 provides a list of the firmware functions callable by downloadable line disciplines. The addresses of these functions can be obtained by using a TCGETSYM command packet or by the **m332xctl** utility command available under SYSTEM V/68 (refer to Appendix L).

TABLE 10-5. FIRMWARE FUNCTION SUB-ROUTINES

Name	Description
open0	Line 0 OPEN function, called by the Bpp_Receiver.
close0	Line 0 CLOSE function, called by the Bpp_Receiver.
ioctl0	Line 0 IOCTL function, called by the Bpp_Receiver.
gate0	Line 0 GATE function, called by the GATE process.
ctl0	Line 0 CTL function, called by the CTL process.
icp0	Line 0 ICP function, called by the ICP process.
ocp0	Line 0 OCP function, called by the OCP process.
icp1	Line 1 ICP function, called by the ICP process.
ocp1	Line 1 OCP function, called by the OCP process.
bpprtn	Return a packet to the host.
ttywait	Wait for both WRITE ring and OUTPUT ring to drain. Should be called by a process.
ttyflush	Flush input rings or output rings or both.
change_ract	Change Receive-Action-Table based on termio structure.
bcopy	Copy a block of data to another place in byte mode.
wcopy	Copy a block of data to another place in word mode.
lcopy	Copy a block of data to another place in long word mode.

TABLE 10-5. FIRMWARE FUNCTION SUB-ROUTINES (cont.)

Name	Description
bzero	Clear a block of memory in byte mode.
lzero	Clear a block of memory in long word mode.
bfill	Fill a block of memory with a byte pattern.
lfill	Fill a block of memory with a long word pattern.
spl[0-7]	Set process interrupt level to 0,...,7, return old level.
splx	Set process interrupt level to X, return old level.
splattn	Mask the attention interrupt.
splpr	Mask the printer device interrupt.
spltimer	Mask the tick timer interrupt.
spltty	Mask the serial device interrupt.
splhi	Mask all interrupts.
getvbr	Return the content of CPU's VBR (Vector Base Register).
getsr	Return the content of CPU's SR (Status Register).
setvec	Set up an interrupt handler for a vector.
strncmp	String compare.
strcopy	String copy.
strlen	String length.
m564putc	Send a character to a port using polling mode, useful only for debugging.
m230putc	Send a character to the printer port (for debugging only).
printf	Formatted print a message into DUMP-AREA on dual-port memory.
sprintf	Same as printf but into specified buffer.
print	Printf and sprintf core function.

The following sections will describe the detail and the calling convention of each function.

10.8.1 OPEN0,CLOSE0,IOCTL0,GATE0,CTL0,ICP0,OCP0,ICP1, And OCP1 Functions

SYNOPSIS:

```
ctl0(tty);
icp0(tty);
ocp0(tty);
icp1(tty);
ocp1(tty);
gate0(tty);

open0(tty, envelope);
close0(tty, envelope);
iocl t0(tty, envelope);

struct tty *tty;
struct envelope *envelope;
```

DESCRIPTION:

These are a set of functions that perform character processing for the standard UNIX line discipline 0. Line 1 shares the same functions with line 0 excepts **icp1()** and **ocp1()**.

Downloadable lines may call these after or before executing some specific code if they want to conform to the UNIX interface. If no special actions need to be taken, these functions can be placed directly into the line switch table as in the case of line 1 (refer to the "Line Switch Table" section).

EXAMPLES:

```
open3(tty, envelope)
struct tty *tty; struct envelope *envelope;
{
        /* perform special action for line 3 */
        bzero(tty->t_user, sizeof (tty->t_user));

        open0(tty, envelope);
}
```

10.8.2 BPPRTN Function

SYNOPSIS:

```
bpprtn(envelop);
struct envelope *envelope;
```

DESCRIPTION:

This function will return the packet back to the host's status pipe of the channel specified in the **status_pipe_number** field of the packet pointed to by the envelope pointer. It will then issue an interrupt to notify the host by using the vector and level established in the INIT packet.

EXAMPLES:

```
ctl3(tty)
struct tty *tty;
{
        envelope = dequeue_envelope(&tty->t_ctl_env);
        bpprtn(envelope);
}
```

10.8.3 TTYWAIT Function

SYNOPSIS:

```
ttywait(tty);
struct tty *tty;
```

DESCRIPTION:

This function will put the calling process to sleep until the WRITE ring, the OUPUT ring, and the device's transmitter FIFO become empty.

Note that this function can be called ONLY by a process, not by the routines specified in the **l_open**, **l_close**, and **l_ioctl** field of the line switch table, since it calls the kernel **_delay** primitive which will swap the running process out to the sleep queue.

EXAMPLES:

```
ctl3(tty)
struct tty *tty;
{
        ttywait(tty);          /* wait for output rings to drain */
        (*tty->t_param)(tty);      /* then reconfigure the device    */
}
```

10.8.4 TTYFLUSH Function

SYNOPSIS:

```
ttyflush(tty, flags);
struct tty *tty;

/* option flags */
#define FREAD     0x01    /* flush both INPUT ring and READ ring   */
#define FWRITE    0x02    /* flush both OUTPUT ring and WRITE ring */
```

DESCRIPTION:

This function flushes the output rings and/or the input rings depending on the option flags set in its argument. All characters in those rings are discarded since it resets the input rings' **put** index equal to **get** index and the output rings' **get** index equal to the **put** index to empty the rings.

In the case of flushing the output rings, it also resumes a suspended transmitter allowing the next write to be transmitted. In the case of flushing the input rings, it sends an XON character and asserts the RTS signal if the remote side was previously blocked. This will allow the remote side to continue sending the data over to the device.

This function is called by the **ioctl** routine when the TCFLSH command packet is received. It may be called in the **close** routine to clean up the rings.

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;  structure envelope *envelope;
{
      if (envelope->packet_pointer->ioctl_command == TCFLSH)
            ttyflush(tty, envelope->packet_pointer->ioctl_argument);
}
```

10.8.5 CHANGE_RACT Function

SYNOPSIS:

```
change_ract(tty);
struct tty *tty;
```

DESCRIPTION:

This function sets up the Receive-Action-Table (RAT) based on the information in the t_tio field of the tty structure. It is called when a termio structure (t_tio) is modified so that the device interrupt handler is able to respond to the new configuration such as new interrupt character or break suppressing.

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;  structure envelope *envelope;
{
        packet = envelope->packet_pointer;
        if (packet->ioctl_command == TCSETA) {
                copy(&packet->parameter_block.tio, &tty->t_tio,
                                                sizeof (struct termio));
                change_ract(tty);
        }
}
```

10.8.6 BCOPY, WCOPY, And LCOPY Functions

SYNOPSIS:

```
bcopy(src, des, nbytes);
char *src, *des;
unsigned short nbytes;

wcopy(src, des, nwords);
short *src, *des;
unsigned short nwords;

lcopy(src, des, nlwords);
long *src, *des;
unsigned short nlwords;
```

DESCRIPTION:

These are very fast **copy** functions and are used to duplicate a block of memory pointed to by the **src** pointer to another pointed to by the **des** pointer. For the sake of speed, they are written in assembly code for maximum optimization.

Note that **wcopy** is very useful to copy a structure to another since most compilers always round a structure to a word boundary. Therefore, the result of the structure size divided by word size always yields an integer number.

Also note that any pointer to an ODD address passed to the **wcopy()** or **lcopy()** results in an address error exception which turns on the FAIL LED, dumps the CPU register set on the dump area, and finally halts the CPU.

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;  structure envelope *envelope;
{
        wcopy(&packet->parameter_block.tio, &tty->t_tio,
                        sizeof (struct termio)/sizeof (short));
}
```

10.8.7 BZERO, LZERO, BFILL, And LFILL Functions

SYNOPSIS:

```
bzero(ptr, nbytes);
char *ptr;   unsigned short nbytes;

lzero(ptr, nlwords);
long *ptr;   unsigned short nlwords;

bfill(ptr, nbytes, pattern);
char *ptr;   unsigned short nbytes;
unsigned char pattern;

lfill(ptr, nlwords, pattern);
char *ptr;   unsigned short lwords;
unsigned long pattern;
```

DESCRIPTION:

Same as the **copy** functions, these functions are also written in assembly code and are used to initialized a block of memory to zero or to a byte pattern or to a longword pattern. They are useful for table initialization as shown in the following example.

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;  structure envelope *envelope;
{
        /* initialize RAT table to no special action characters    */
        lfill(tty->t_ract, 256/sizeof(long),
                        (A_NOP<<24)+(A_NOP<<16)+(A_NOP<<8)+A_NOP);

        /* then make the CTL-S character as a XOFF character       */
        tty->t_ract[0x11] = A_XOFF;
}
```

10.8.8 SPL[0-7],SPLX,SPLATTN,SPLPR,SPLTIMER,SPLTTY, And SPLHI Functions

SYNOPSIS:

```
spl0(); spl1(); spl2(); spl3(); spl4(); spl5(); spl6(); spl7();
splattn(); splpr(); spltimer(); spltty(); splhi();

splx(SR);
unsigned short SR;
```

DESCRIPTION:

These are a set of functions that change the processor interrupt level to the level implied in the function name and then return the previous content of the processor's status register (SR). For example, spl3 will return the current value of SR and change the interrupt level to 3 to mask all interrupts equal to or less than level 3.

Splattn(), splpr(), spltimer(), spltty(), and splhi() are acronyms to spl1(), spl4(), spl5(), spl6(), and spl7, respectively. They are used to mask the attention interrupt, printer device interrupt, tick timer interrupt, or serial device interrupt.

Splx() is different in the way of calling. It requires a new value as an argument which will be loaded into the processor's status register (SR). It also returns the previous value of SR before changing to the new value.

10

Note that spl1() or splattn() can be used to protect a critical region between processes since the kernel will not swap the running process out if the current CPU interrupt level is not equal to zero.

EXAMPLES:

```
icp3(tty)
struct tty *tty;
{
        /* mask attention interrupt and prevent process swapping    */
        old_level = splattn();  /* and save the current SR    */
        .....
        splx(old_level);        /* restore the previous level */
}
```

10.8.9 GETVBR And GETSR Functions

SYNOPSIS:

```
unsigned long *
getvbr();

unsigned short
getsr();
```

DESCRIPTION:

Getvbr() and getsr() return the content of the CPU vector base register (VBR) and status register (SR), respectively.

EXAMPLES:

```
open3(tty, envelope)
struct tty *tty; struct envelope *envelope;
{
        extern unsigned long *getvbr, buserr_handler();

        /* check for the first time open after close                */
        if ((tty->t_cstate & ISOPEN) == 0) {
                /* replace the bus-error handler with the new one  */
                *(getvbr() + 2) = buserr_handler;
        }
}
```

10.8.10 SETVEC Function

SYNOPSIS:

```
setvec(vector_number, handler, handler_argument)
int    vector_number;
int    (*handler)();
int    handler_agument;
```

DESCRIPTION:

This is a generic function allowing an interrupt handler written completely in C to be executed when an interrupt on the vector specified in the argument occurs. It also passes an argument supplied by the caller to the interrupt handler routine so that a common interrupt handler can be developed for all similar devices.

Note that this works for the interrupt vector only, not for the bus-error or address error vector (vector 2 and 3), since such exceptions have a different format on the stack frame.

EXAMPLES:

Assume that the function below will be downloaded into the MVME332XT's local memory, then executed to replace a handler for all serial device interrupts.


```
download_initialize()
{
        extern new_sio_intr();

        vector = 0x55;
        for (device_number = 0; device_number < 8; device_number++) {
                setvec(vector, new_sio_intr, device_number)
                vector += 4;
        }
}

/*
 * New handler, get a character from the device, then put it into
 * the INPUT ring ready for the ICP to use.
 */
new_sio_intr(device_number)
int device_number;
{
        tty = &ttytab[device_number];  /* get tty pointer          */
        c = get_char(tty);
        put_character(&tty->iring, c);
}
```

10.8.11 STRNCMP, STRCOPY, And STRLEN Functions

SYNOPSIS:

```
int
strncmp(str1, str2, nchars)
char *str1, *str2;
int  nchars;

int
strlen(str)
char *str

strcopy(str_des, str_src)
char *str_des, *str_src;
```

DESCRIPTION:

These functions are identical to the string functions specified in the SYSTEM V/68 Programmer's Reference Manual.

Strncmp() performs a lexicographical comparison of the strings pointed to by **str_src** and **str_des** up to **nchars** characters and returns an integer less than, equal to, or greater than zero, when **str1** is less than, equal to, or greater than **str2**, respectively.

Strlen() returns the number of characters in **str**, not including the terminating null character.

Strcopy() copies string **str_src** to string **str_des** until the null character has been copied.

10

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        /* put a message into the host buffer                    */
        strcopy(packet->parameter_block.dl_info.host_addr,
                        "hello world!!\n");
}
```

10.8.12 M546PUTC And M230PUTC Functions

SYNOPSIS:

```
m564putc(device_number, c)
int device_number;
char c;

m230putc(8, c)
char c;
```

DESCRIPTION:

These functions send a character **c** to the device specified in the **device_number** in polling mode to guarantee that the character has been transmitted before the function returns. They are useful for printing debug messages when tracing the code.

EXAMPLES:

```
ioctl3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        /* print a message on the printer port                    */
        for(i = 0; i < 14; i++)
                m230putc(8, "hello world!!\n"[i]);
}
```

10.8.13 PRINTF And SPRINTF Functions

SYNOPSIS:

```
printf(format, arg0, ..., argn);

char *
sprintf(buffer, format, arg0, ..., argn);

char *format, buffer;
int  arg0, argn;
```

DESCRIPTION:

These functions are identical to those in the SYSTEM V/68 Programmer's Reference Manual, except that the output of printf() will be placed in the dump area of the dual-port memory. There is no floating point format supported in these functions.

Sprintf() places the output into the supplied buffer, then returns the address of the last character in the buffer.

EXAMPLES:

```
printf("today is %d/%d/%d\n", month, date, year);
```

10.8.14 PRINT Function

SYNOPSIS:

```
print(format, arg, putc, putc_arg);
char *format, **arg;
int   (*putc)(), putc_arg;
```

DESCRIPTION:

Print() is a core function of printf() and sprintf(). It parses the format string and uses the **putc()** function passed to it as an argument to output a character. It is useful for a user to customize the output routine as shown in the example.

EXAMPLES:

The **pprintf()** performs the exact function as printf(), but all outputs will be on the printer port.

```
pprintf(format, args);
char *format;
int  args;
{
        extern   m230putc();

        print(format, &args, m230putc, 8);
}
```


The **dprintf()** also performs the same function as printf(), but the outputs will be sent to the device specified in the argument.

```
dprintf(device, format, args);
int  device;
char *format;
int  args;
{
        extern   m230putc(), m564putc();

        if (device < 8)      /* serial device                    */
                print(format, &args, m564putc, device);
        else
        if (device == 8)     /* printer device                   */
                print(format, &args, m230putc, device);
        else
                error("invalid device number");
}
```

10.9 FIRMWARE STATIC VARIABLES

Most of the firmware variables are located in the tty structure except some that are global to all devices. Table 10-6 provides a list of global variables of which the address can be obtained by using the TCGETSYM command packet or by the **m332xctl** command available on SYSTEM V/68 (refer to Appendix L).

TABLE 10-6. FIRMWARE GLOBAL VARIABLES

Name	Description
ttytab	Table of tty structures for all devices.
linesw	Line discipline switch table.
linecnt	Number of lines available in the line switch table.
maxline	Maximum number of lines the line switch table can have.
romlines	Number of lines in the on-board ROMs.
attn_status	The content of the CPU status register (SR) before the attention interrupt.
timer_status	The content of the CPU status register (SR) before the tick timer interrupt.

The next sub-sections describe the details of the above variables.

10.9.1 TTYTAB Table

SYNOPSIS:

```
struct tty ttytab[9];
```

DESCRIPTION:

Ttytab[] is an array of the devices' tty structures, one for each device number, and resides in the local memory which is accessible only for the MVME332XT CPU.

The firmware uses the device number found in the packet to index into this array to obtain the pointer to the device tty structure as shown in the below example.

EXAMPLES:

```
bpp_receiver();
{
                struct tty *tty;

                tty = &ttytab[packet->device_number];
}
```

10.9.2 LINESW, LINECNT, MAXLINE, And ROMLINES Variables

SYNOPSIS:

```
struct line_switch linesw[7];

short linecnt, maxline, romlines;
```

DESCRIPTION:

Linesw[] is an array of seven line disciplines in which line 0 corresponds to index 0 and resides in the local memory. The firmware uses this to switch from one line to another when the **c_line** field in the working termio (tty-> t_tio) is changed.

Linecnt specifies the number of line disciplines currently set in the line switch table. The CTL process uses this to validate the **c_line** field in the packet parameter block when a TCSETA or TCSETP command is received.

Maxline specifies maximum number of lines the line switch table can have. The CTL process uses this to validate the parameter of the TCLINE command.

EXAMPLES:

```
CTL()                                   /* CTL process                  */
{
        (*linesw[tty->t_tio.c_line].l_ctl)(tty);
}

ctl3(tty)
struct tty *tty;
{
        if (packet->parameter_block.tio.c_line > lincnt)
            packet->error = ENXIO;
}
```

10.9.3 ATTN_STATUS And TIMER_STATUS Variables

SYNOPSIS:

```
unsigned short attn_status;
```

DESCRIPTION:

Attn_status or **timer_status** records the content of the CPU's Status Register (SR) before getting into the attention interrupt or timer interrupt handlers, respectively.

It is useful for the **Bpp_Receiver** or the **Timer_Handler** to inform the **_signal** kernel primitive whether it wants to preempt the running process or not. If the CPU is running in the context of an interrupt handler indicated by these variables, do not swap the process.

EXAMPLES:

```
open3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        if (attn_status & 0x0700)   /* attention interrupt   */
                                    /* occurs inside another */
                                    /* interrupt handler     */
                _signal(&tty->t_gate_sem, NO_PREEMPT);
        else
                _signal(&tty->t_gate_sem, PREEMPT);
}
```

10.10 KERNEL FUNCTION PRIMITIVES

Table 10-7 provides a list of kernel function primitives. The details of these can be found in the ADC Kernel Firmware Manual.

TABLE 10-7. KERNEL FUNCTION PRIMITIVES

Name	Description
_aging	Age the ready list (called by the tick timer interrupt).
_can_timeout	Cancel an event timeout (used as a watchdog timer).
_create	Create a process ready to run.
_cycle	Voluntarily relinquish a time slide.
_delay	Put the running process to sleep for a number of ticks.
_deq	Dequeue an item from a linked list.
_dispatch	Activate a process from the ready list to run.
_enq	Enqueue an item into a linked list.
_exit	Remove the running process from existence.
_freemem	Deallocate a block of memory, return it to free list.
_get_user	Returns the user value from the running process' PCB.
_getbuf	Allocate a buffer from a fixed buffer poll.
_getmem	Allocate a block of memory from the free list.
_halt	Mask all interrupts then halt the CPU.
_kerinit	Initialize the kernel data structure.
_link	Create a link list from a memory poll.
_mask	Mask the processor interrupt level to kernel level.
_nullmgr	Is a place holder for an idle system.
_prenq	Enqueue a process onto the ready list based on priority.
_put_user	Load a used value into a specified PCB.
_putbuf	Deallocate a buffer to a fixed buffer pool.
_receive	Receive a standard message.
_recvptr	Receive a short form message.
_send	Send a standard message.
_sendptr	Send a short form message.
_set_timeout	Start a watchdog timeout.
_signal	Signal a resource/event semaphore.
_sleep	Sleep on an event address.
_stop	Unmask all interrupts, stop the CPU until interrupted.

TABLE 10-7. KERNEL FUNCTION PRIMITIVES (cont.)

Name	Description
_swap	Swap a process image to another.
_timer_int	Kernel timer interrupt housekeeping.
_unmask	Unmask the processor interrupt level.
_wait	Wait on a resource/event semaphore.
_wakeup	Wake up all processes waiting on an event address.

The next sub-sections describe some frequently used primitives.

10.10.1 _DELAY Primitive

SYNOPSIS:

```
_delay(nticks);
unsigned long nticks;     /* number of timer ticks to delay process */
```

DESCRIPTION:

This will put the calling process to sleep for the number of timer ticks specified in **nticks**, then dispatch another process pending in the ready to run list.

In the current implementation of the firmware, the tick time is configured to 10 milliseconds.

EXAMPLES:

```
ctl3(tty)
struct tty *tty;
{
        /* assert RTS for 200ms then negate it */
        tty->t_devaddr.sio->xmtctl |= RTS;
        _delay(200/10);
        tty->t_devaddr.sio->xmtctl &=  RTS;
}
```

10.10.2 _WAIT Primitive

SYNOPSIS:

```
_wait(semaphore);
struct semaphore {
        short   count;
        long    reserved0;
        long    reserved1;
} *semaphore;
```

DESCRIPTION:

This will put the calling process into a wait list, if the count field becomes zero or negative, waiting for a _signal call from another process or interrupt handler. Otherwise, the count field is decremented by one and returns immediately to the calling process.

Note that the semaphore structure should be initialized to zero before the first call to this kernel function.

EXAMPLES:

```
ICP()
{
        for (;;) {
                _wait(&tty->t_icp_sem);
                (*linesw[tty->t_tio.c_line].l_icp)(tty);
        }
}
```

10.10.3 _SIGNAL Primitive

SYNOPSIS:

```
_signal(semaphore, preempt);
struct semaphore {
        short   count;
        long    reserved0;
        long    reserved1;
} *semaphore;
int preempt;
```

DESCRIPTION:

This will wake up a process, if any available, waiting on a semaphore, then dispatch the process immediately to run if the process priority is higher than the running process and the **preempt** flag is set. The **count** is also incremented by one. A positive value in the **count** field indicates that there is no process waiting on this semaphore.

EXAMPLES:

```
bpp_receiver()
{
        if (attn_status & 0x0700)  /* attention interrupt occurs   */
                                   /* inside another interrupt     */
                                   /* handler                      */
                _signal(&tty->t_ctl_sem, NO_PREEMPT);
        else
                _signal(&tty->t_ctl_sem, PREEMPT);
}
```

10

10.10.4 _SLEEP Primitive

SYNOPSIS:

```
_sleep(address);
int address;
```

DESCRIPTION:

This will put the calling process to sleep on an address waiting for another process or interrupt handler to call the _wakeup with the corresponding address that it sleeps on.

EXAMPLES:

```
OCP()
{
        while (OUTPUT ring is not empty)
                _sleep(&tty->t_oring);
}
```

10.10.5 _WAKEUP Primitive

SYNOPSIS:

```
_wakeup(address);
int address;
```

DESCRIPTION:

This will will wake up all processes waiting on the address specified in the **address** field.

EXAMPLES:

```
sio_transmitter()
{
        if (OUTPUT_ring is empty)
             _wakeup(&tty->t_oring);
}
```

10.11 DOWNLOAD LINE DISCIPLINE EXAMPLE

Assume that the "line3.c" file is the file containing all the routines for line discipline 3, which will be downloaded and executed by the MVME332XT. The format for this example is illustrated in Figure 10-11.

```
/***
 ***          line3.c : line discipline 3 example.
 ***/

#include " header.h"          /* include data structure declarations */


/* line-3 open() example */

open3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        if (need_to_wait) {         /* hand it to the GATE process   */
                queue_envelope(&tty->t_gate_env, envelope);
                _signal(&tty->t_gate_sem, PREEMPTED);
        } else
                device_open(tty);  /* enable the device              */
}


/* line-3 icp() example */

icp3(tty)
struct tty *tty;
{
        c = get_character(&tty->t_iring);   /* from INPUT ring       */
        c = translation(c);
        put_character(&tty->t_rring, c);    /* put into READ ring    */
        /* check for any one waiting for character                   */
        if (tty->t_read_wakeup) {
                envelope = dequeue_envelope(&tty->t_read_wakeup);
                bpprtn(envelope);           /* return the packet     */
        }
}
```

FIGURE 10-11. DOWNLOAD LINE DISCIPLINE EXAMPLE (PAGE 1 OF 4)

```
/* line-3 ocp() example */

ocp3(tty)
struct tty *tty;
{
        c = get_character(&tty->t_wring);   /* from the WRITE ring  */
        c = translation(c);
        put_character(&tty->t_oring, c);    /* put into OUTPUT ring */
        /* check for any one waiting for character */
        if (low_water_mark && tty->t_write_wakeup) {
                envelope = dequeue_envelope(&tty->t_write_wakeup);
                bpprtn(envelope);           /* return the packet    */
        }
}


/* line-3 ioctl() example */

ioctl3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        if (need_to_wait) {         /* hand it to the CTL process   */
                queue_envelope(&tty->t_ctl_env, envelope);
                _signal(&tty->t_ctl_sem, PREEMPTED);
        } else
                decode_ioctl_command(envelope, tty);
}


/* line-3 close3() example */

close3(tty, envelope)
struct tty *tty;
struct envelope *envelope;
{
        if (need_to_wait) {         /* hand it to the GATE process  */
                queue_envelope(&tty->t_gate_env, envelope);
                _signal(&tty->t_gate_sem, PREEMPTED);
        } else
                device_close(tty);          /* disable the device   */
}
```

FIGURE 10-11. DOWNLOAD LINE DISCIPLINE EXAMPLE (cont.)
(PAGE 2 OF 4)

```
/* line-3 ctl() example */

ctl3(tty)
struct tty *tty;
{
        envelope = dequeue_envelope(&tty->t_ctl_env);
        ttywait(tty);     /* wait for output rings to drain          */
        if (envelope->packet_pointer->command == TCSETAW) {
                ttyflush(F_READ);
        (*tty->t_param)(tty);          /* reconfigure the device     */
        bpprtn(envelope);              /* return the packet          */
}


/* line-3 gate() example */

gate3(tty)
struct tty *tty;
{
        envelope = dequeue_envelope(&tty->t_gate_env);
        ttywait(tty);     /* wait for output rings to drain          */
        if (envelope->packet_pointer->command == OPEN) {
                device_open(tty);      /* enable device              */
        else
                device_close(tty);     /* disable device             */
        bpprtn(envelope);              /* return the packet          */
}


/*
 * linetable[] and linecount are only required by the " m332xctl"
 * utility.  A zero in the linetable field indicates the use of the
 * default value (only applies to new versions of the firmware, i.e.,
 * versions later than 7.3).
 */
```

FIGURE 10-11. DOWNLOAD LINE DISCIPLINE EXAMPLE (cont.)
(PAGE 3 OF 4)

```
#ifdef OLD_FW_VERSION                    /* less than or equal to 7.3    */

extern  openØ(), icpØ(), ocpØ(), ioctlØ(), closeØ(), ctlØ(), gateØ(),
        icpl(), ocpl();

struct linesw linetable[] = {
        /* line Ø */ openØ, icpØ, ocpØ, ioctlØ, closeØ, ctlØ, gateØ,
        /* line 1 */ openØ, icpl, ocpl, ioctlØ, closeØ, ctlØ, gateØ,
        /* line 2 */ openØ, icpØ, ocpØ, ioctlØ, closeØ, ctlØ, gateØ,
        /* line 3 */ open3, icp3, ocp3, ioctl3, close3, ctl3, gate3,
};

#else                                    /* greater than 7.3             */

struct linesw linetable[] = {
        /* line Ø */     Ø,     Ø,     Ø,      Ø,      Ø,     Ø,     Ø,
        /* line 1 */     Ø,     Ø,     Ø,      Ø,      Ø,     Ø,     Ø,
        /* line 2 */     Ø,     Ø,     Ø,      Ø,      Ø,     Ø,     Ø,
        /* line 3 */ open3, icp3, ocp3, ioctl3, close3, ctl3, gate3,
};

#endif

long linecount = sizeof linetable/sizeof struct linesw;
```

FIGURE 1Ø-11. DOWNLOAD LINE DISCIPLINE EXAMPLE (cont.)
(PAGE 4 OF 4)

10

10.12 DOWNLOAD PROCEDURE UNDER SYSTEM V/68

Under SYSTEM V/68, the "m332xctl" utility can be used to obtain the firmware's symbol table and to download the code to the MVME332XT.

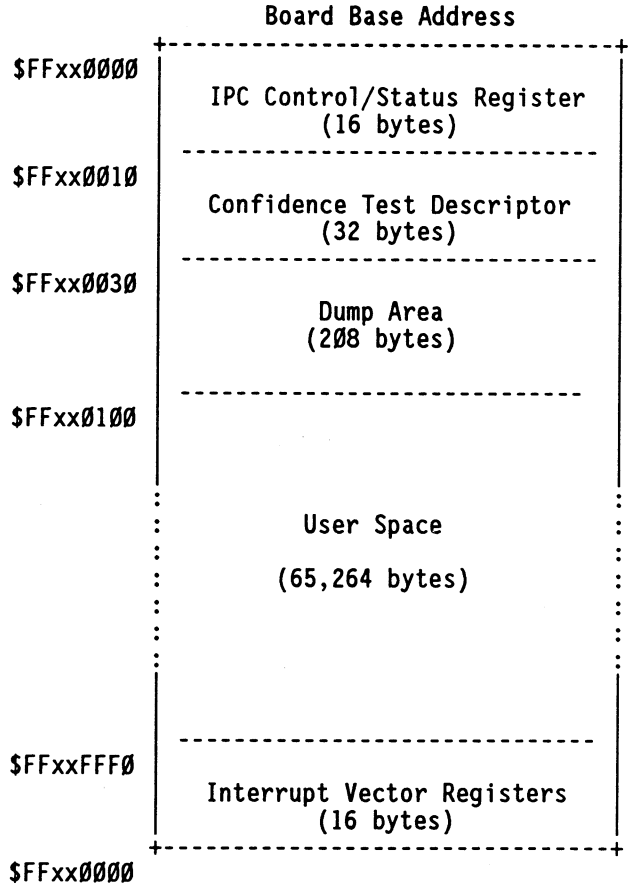
Figure 10-12 illustrates an example of how to compile, link, and download the file "line3.c":

```
# compile line3.c with 68010 C compiler to produce
# relocatable file " line3.o".
cc010 -c -O -o line3.o line3.c

# get the symbol table from the first MVME332XT,
# then put into file " symtab.ld".
/etc/m332xctl -s /dev/m332x08 > symtab.ld

# linking to produce executable file " line3".
ld -o line3 symtab.ld line3.o

# download " line3"  to the first MVME332XT
/etc/m332xctl -d line3 -l /dev/m332x08
```

FIGURE 10-12. DOWNLOAD PROCEDURE

APPENDIX A - DUAL PORT MEMORY MAP

Figure A-1 illustrates the map of the MVME332XT dual-port memory. It consists of the IPC Control/Status Register, confidence test descriptor, dump area, user space, and interrupt vector registers. The user space is allocated by the host. It is initialized to zero when reset.

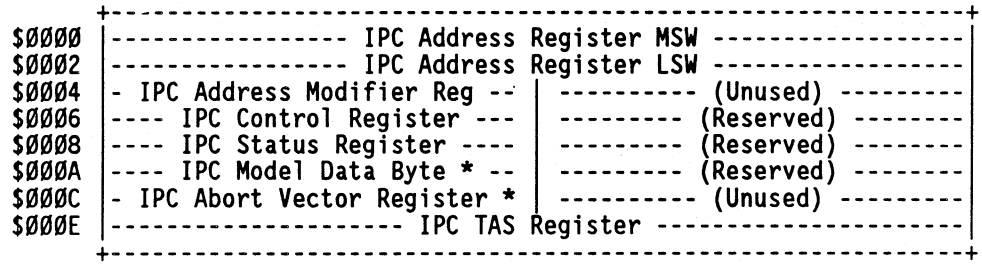


Where: xx is configurable in 8-position switch

FIGURE A-1. DUAL-PORT MEMORY MAP

APPENDIX B - IPC CONTROL/STATUS REGISTER SPACE

B



* Unused

FIGURE B-1. CSR REGISTER MAP

This CSR space is accessible from the VMEbus as well as from local RAM space. This permits both the host CPU and the IPC CPU to read and write any location in this CSR space.

CSR address offset: $Ø6

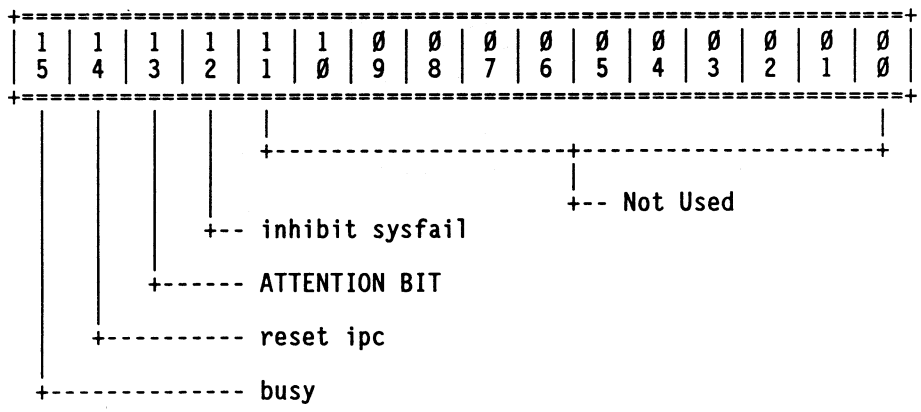


FIGURE B-2. CSR CONTROL REGISTER BIT ASSIGNMENTS

B

CSR address offset: $ØE

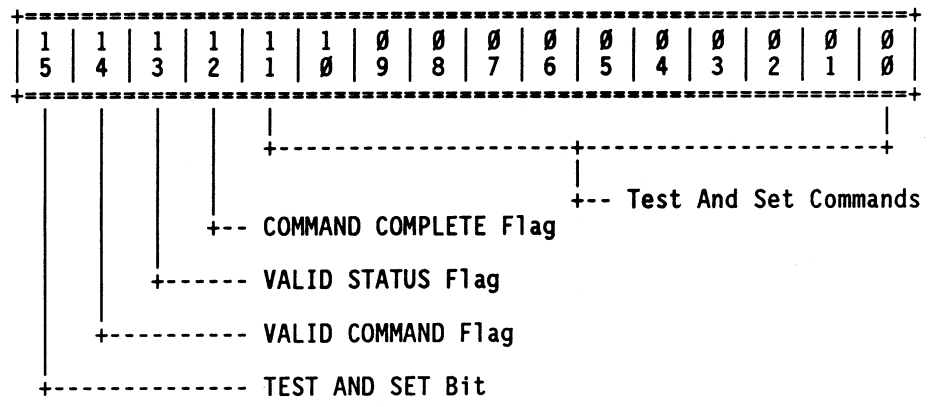


FIGURE B-3. CSR COMMAND INTERFACE FLAGS

TABLE B-1. CSR TEST AND SET COMMANDS

Command Field	CSR Command
Ø	reserved
1	Create Channel
2	Delete Channel
3	unassigned
:	:
$FFF	unassigned

APPENDIX C - CONFIDENCE TEST ERROR CODES

These codes reflect the status of the MVME332XT if a failure occurs during the power-up or reset confidence check. The descriptions indicate the last operation successfully performed if that code appears in the display on the diagnostic serial port card and in the Composite Status Word (CSW) in the Confidence Test Descriptor.

TABLE C-1. CONFIDENCE TEST ERROR CODES

CSW	Test Description
$0000	RESET
$1100	Basic CPU Simple Instruction Test
$1200	Basic CPU Complex Instruction Test
$2100	Local RAM Walking Bit Test
$2200	Local RAM Byte/Word/Long Test
$2400	Local RAM March Test
$3100	Local ROM CRC checking Test
$4100	Dual-Port RAM Walking Bit Test
$4200	Dual-Port RAM Byte/Word/Long Test
$4400	Dual-Port RAM March Test
$5200	Extended CPU Complex Addressing Test
$5300	Extended CPU Exception Test
$6100	MK68564 Register Test
$6200	MK68564 Tx/Rx Polling Test
$6300	MK68564 Tx/Rx Interrupt test
$6400	MK68564 Baud Rate Test
$7100	M68230 Register Test
$7200	M68230 Timer Counters Test
$7300	M68230 Timer Interrupt Test
$7400	M68230 Printer Interrupt Test
$8100	Local CSR Register Test
$8200	Attention Interrupt Test

APPENDIX D - CHANNEL HEADER STRUCTURE

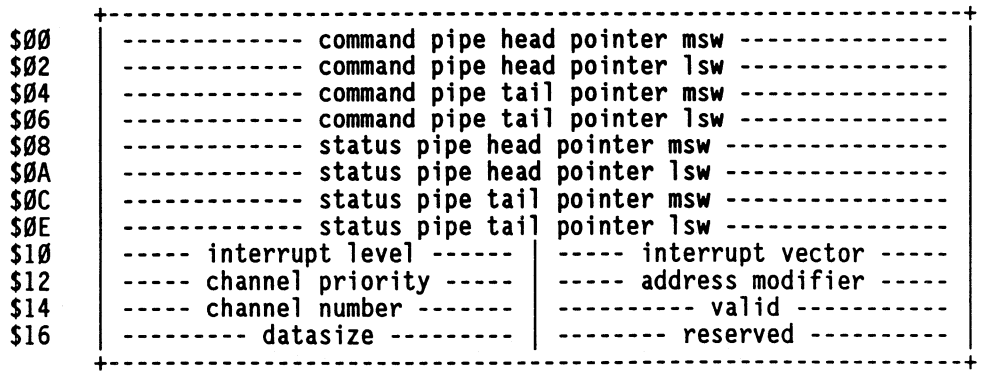


FIGURE D-1. COMMAND CHANNEL HEADER STRUCTURE

APPENDIX E - BPP ENVELOPE AND PACKET STRUCTURES

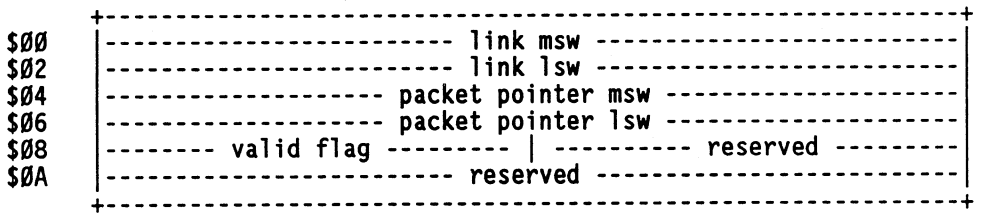


FIGURE E-1. BPP ENVELOPE FORMAT

REQUIRED PARAMETERS:

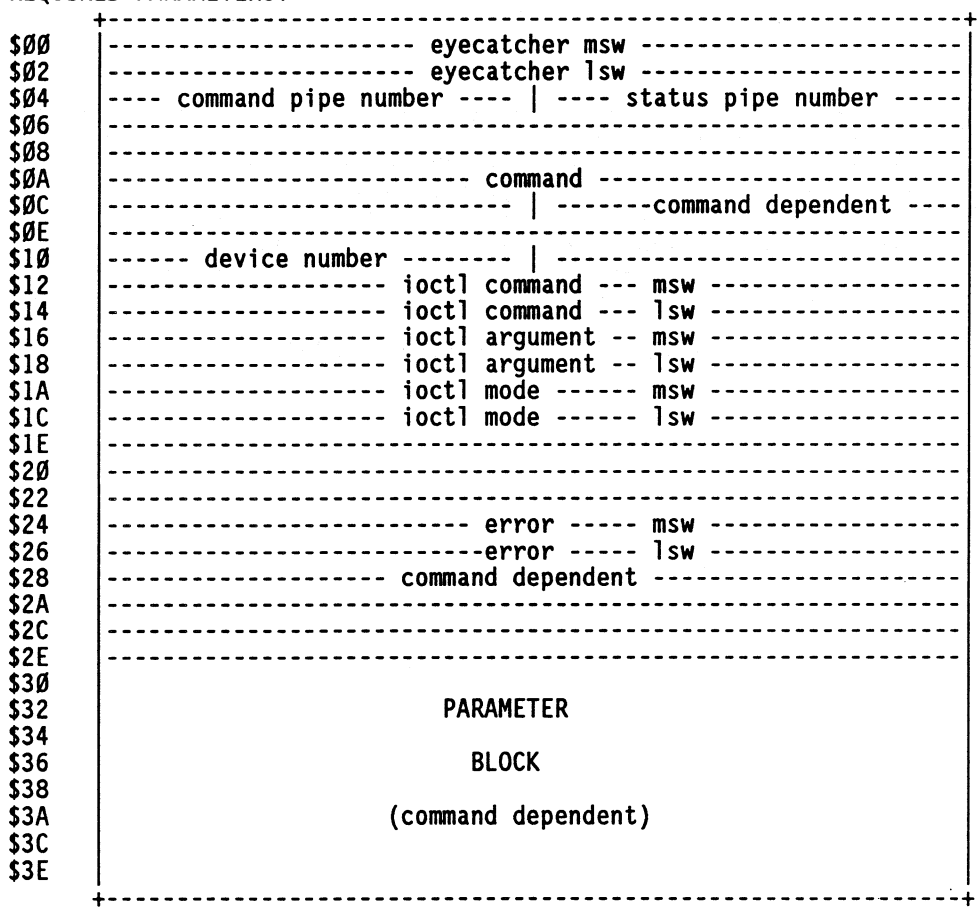


FIGURE E-2. IPC PACKET FORMAT

APPENDIX F - DEVICE NUMBER

TABLE F-1. DEVICE NUMBER ASSIGNMENT

Device Number	Physical Device
$Ø	Serial Port Ø
$1	Serial Port 1
$2	Serial Port 2
$3	Serial Port 3
$4	Serial Port 4
$5	Serial Port 5
$6	Serial Port 6
$7	Serial Port 7
$8	Printer Port

APPENDIX G - MVME332XT COMMAND SUMMARY

TABLE G-1. MVME332XT COMMAND SUMMARY

Command Code	Operation
$ØØ	Init
$Ø1	Read_Wakeup
$Ø2	Write_Wakeup
$Ø3	Open
$Ø4	Ioctl
$Ø5	Close
$Ø6	Event

TABLE G-2. IOCTL COMMANDS

Ioctl Commands	Value	Description
LDOPEN	$44ØØ	Set device internal state to open.
LDCLOSE	$44Ø1	Clear device internal state, flush all rings.
LDCHG	$44Ø2	No operation, return no error.
LDGETT	$44Ø8	Get current virtual terminal information into the termcb structure in the packet.
LDSETT	$44Ø9	Set virtual terminal parameters to the new one.
TCGETA	$54Ø1	Get a device's current termio structure.
TCSETA	$54Ø2	Change a device's termio structure to the new one.
TCSETAW	$54Ø3	Same as TCSETA but wait for device's WRITE ring and OUTPUT ring to drain.
TCSETAF	$54Ø4	Same as TCSETAF but flush the input rings after waiting for the output rings to drain.
TCSBRK	$54Ø5	Transmit a Break Sequence on output (as long as 25Øms).
TCXONC	$54Ø6	Suspend or resume the output, send XON or XOFF, assert or negate RTS or DTR depend on ioctl argument field.
TCFLSH	$54Ø7	Flush WRITE ring and OUTPUT ring, READ ring and INPUT ring or both pairs depend on ioctl argument field.
TCSETHW	$544Ø	Enable or Disable hardware handshake feature.
TCGETHW	$5441	Get current information of hardware handshake.

TABLE G-2. IOCTL COMMANDS (cont.)

Ioctl Commands	Value	Description
TCGETDL	$5442	Get the downloadable address and size of MVME332XT local memory.
TCDLOAD	$5443	Download a block of data or code into MVME332XT local memory.
TCLINE	$5444	Copy a line discipline switch table previously downloaded into the internal data structure of MVME332XT.
TCEXEC	$5445	Instruct the firmware to execute a functional address previously downloaded into MVME332XT local RAM.
TCGETVR	$5446	Get the MVME332XT firmware version and revision number.
TCGETDF	$5447	Same as TCGETA command, but get the default open termio structure which is used to configure a device when open.
TCSETDF	$5448	Same as TCSETAW command, but change both the default open termio structure and the working termio structure.
TCGETSYM	$5449	Get the firmware symbol table to link downloadable code.
TCWHAT	$544A	Get all SCCS IDs of the firmware files.
TCGETDS	$544C	Get the current status of DCD, CTS, DTR, PR_FAULT, PR_POUT, and PR_SELECT.
TIOCGETP	$74Ø8	Get device's current termio structure by using sgttyb structure.
TIOCSETP	$74Ø9	Change device's termio structure by using sgttyb structure.

APPENDIX H - MVME332XT ERROR CODE

TABLE H-1. MVME332XT ERROR CODES

Name	Code	Descriptions
EIO	5	I/O Error. Some physical I/O error. This error indicates that an abnormal hardware condition has occurred that prevents future access to the device.
ENXIO	6	No such device or address. I/O on a device which does not exist; or I/O is beyond the limit of the device.
ENOMEM	12	Not enough space. Some commands such as create a table for ISP require allocation of the local memory. If this request is not satisfied, this error will be returned to the host.
EEXIST	17	Device or address exists. Attempt to create a existing table for ISP will receive this error code.
EINVAL	22	Invalid argument. One or more command parameters are invalid.

APPENDIX I - MVME332XT COMPONENT PLACEMENT

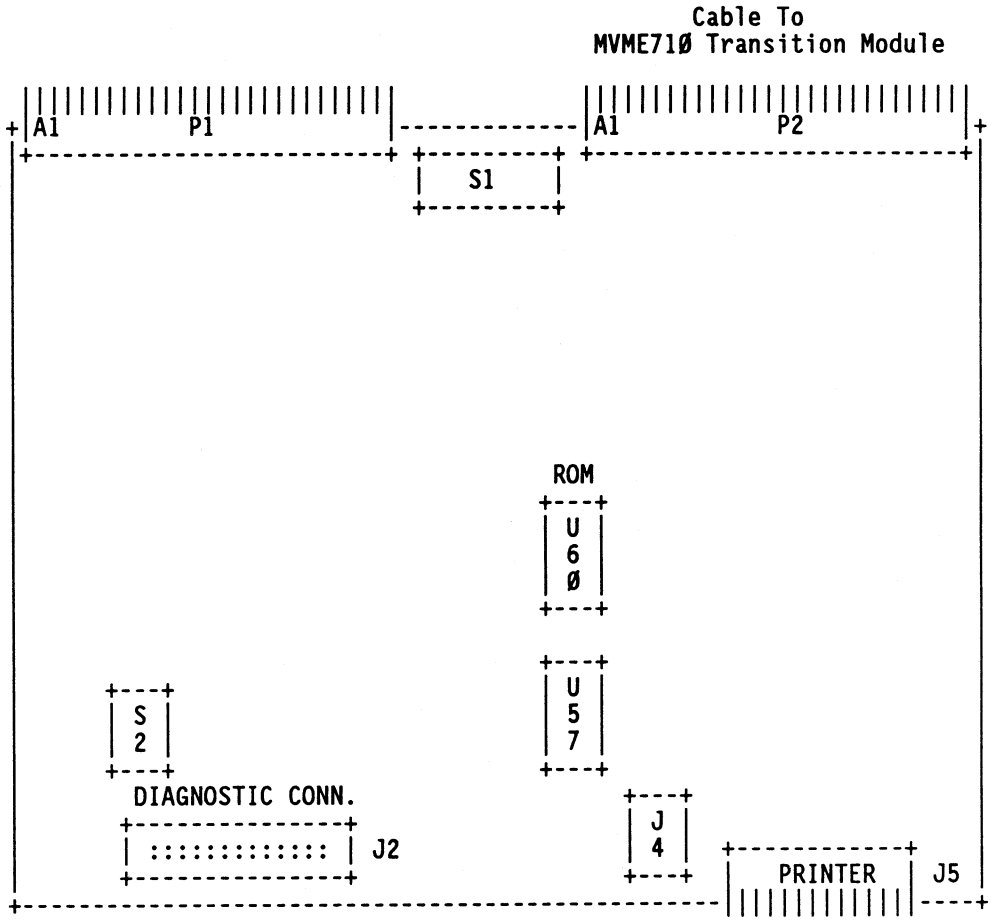


FIGURE I-1. MVME332XT CABLE/SWITCH DIAGRAM

Figure I-1 illustrates the locations of the plugs, switches, and sockets on the MVME332XT that the users must be aware of. The view is from the component side of the MVME332XT board. The use of each of these is given in the following table.

The MVME71Ø transition module is cabled to the P2 connector. The printer cable is plugged into the Centronics standard connector J4.

The "even" ROM contains data bytes that maps to even addresses in the memory. The "odd" ROM contains data bytes that maps to odd addresses in the memory.

TABLE I-1. BOARD SWITCH AND COMPONENT LOCATIONS

Location	Description
J2	Connector for the diagnostic board.
J4	ROM size select jumper (1-2 for 27512, 2-3 for 27256).
J5	Centronics 36-pin connector for printer.
U6Ø	Firmware ROM (Even).
U57	Firmware ROM (Odd).
S1	Board Base Address Switches.
S2	Firmware Select Mode (burn-in mode, FAT test mode).

The address mapping switches determine where the MVME332XT board base address appears in the host memory space (on the VMEbus). The most significant bit is closest to P2. The switches correspond to bits 23 through 16 of the address lines. The default setting of the switches is $Ø1111ØØØ ($78) which represents a base address of $FF78ØØØØ (note that the ON position is Ø).

I

The firmware mode selection switch S2 allows a user to select several modes supported in the firmware to perform burn-in sequence, final assembly test (FAT), or the firmware. Table I-1 provides the position of S2 for firmware modes.

TABLE I-2. S2 SELECTION

S2				
1	2	3	4	Description
OFF	OFF	OFF	OFF	Burn-In test mode
OFF	OFF	OFF	ON	FAT test mode
OFF	OFF	ON	ON	Firmware mode (Factory Setting)
	others			Reserved

APPENDIX J - GENERAL TERMINAL INTERFACE (TERMIO)

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. Common features of this interface are presented in this section.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by getty and become a user's standard input, output, and error files. The first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the control terminal for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a fork(2). A process can break this association by changing its process group using setpgrp(2).

A terminal associated with one of these files ordinarily operates in the full-duplex mode. Characters may be typed at any time and are only lost when the system's character input buffers become completely full or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, all the saved characters are discarded without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read is suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, one line at most is returned. It is not necessary, however, to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character # erases the last character typed, except that it does not erase beyond the beginning of the line. By default, the character @ kills (deletes) the entire input line and optionally outputs a newline character. Both these characters operate on a key-stroke basis, independent of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case, the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates an interrupt signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see signal(2).

QUIT (Control-| or ASCII FS) generates a quit signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it is not only terminated but a core image file (called core) is created in the current working directory.

SWTCH ASCII NUL is used by the job control facility, shl(1), to change the current layer to the control layer.

ERASE (#) erases the preceding character. It does not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a newline, and the EOF is discarded. Thus, if there are no characters waiting, i.e., the EOF occurred at the beginning of a line, zero characters are passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, similar to NL. Normally, it is not used.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, SWTCH, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a hangup signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished printing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they are printed, it is suspended when its output queue exceeds some limit. When the queue has drained to some threshold, the program is resumed.

Several ioctl(2) system calls apply to terminal files. The primary calls use the following structure, defined in < termio.h> :

```
#define     NCC  8
struct     termio {
      unsigned  short  c_iflag;     /* input modes */
      unsigned  short  c_oflag;     /* output modes */
      unsigned  short  c_cflag;     /* control modes */
      unsigned  short  c_lflag;     /* local modes */
                 char  c_line;      /* line discipline */
      unsigned  char   c_cc[NCC];   /* control chars */
};
```

CONTROL CHARACTERS

The special control characters are defined by the array c_cc. The relative positions and initial values for each function are as follows:

Ø	VINTR	DEL
1	VQUIT	FS
2	VERASE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	reserved	
7	VSWTCH	NUL

Refer to the section "LOCAL MODES" for information about enabling and disabling the functions of these characters. As stated in that section and shown in termio.h, if canonical processing is not set, positions 4 and 5 contain values for VMIN and VTIME, respectively.

INPUT MODES

The c_iflag field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map uppercase to lowercase on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and, therefore, not read by any process. Otherwise, if BRKINT is set, the break condition generates an interrupt signal and flushes both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

J

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to seven bits; otherwise, all eight bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise, if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received uppercase alphabetic character is translated into the corresponding lowercase character.

If IXON is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. All start/stop characters are ignored and not read. If IXANY is set, any input character restarts output that has been suspended.

If IXOFF is set, the system transmits START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

OUTPUT MODES

The c_oflag field specifies the system treatment of output:

OPOST	ØØØØØØ1	Postprocess output.
OLCUC	ØØØØØØ2	Map lowercase to uppercase on output.
ONLCR	ØØØØØØ4	Map NL to CR-NL on output.
OCRNL	ØØØØØ1Ø	Map CR to NL on output.
ONOCR	ØØØØØ2Ø	No CR output at column Ø.
ONLRET	ØØØØØ4Ø	NL performs CR function.
OFILL	ØØØØ1ØØ	Use fill characters for delay.
OFDEL	ØØØØ2ØØ	Fill is DEL, else NUL.
NLDLY	ØØØØ4ØØ	Select newline delays:
NLØ	Ø	
NL1	ØØØØ4ØØ	
CRDLY	ØØØ3ØØØ	Select carriage-return delays:
CRØ	Ø	
CR1	ØØØ1ØØØ	
CR2	ØØØ2ØØØ	
CR3	ØØØ3ØØØ	
TABDLY	ØØ14ØØØ	Select horizontal-tab delays:
TABØ	Ø	
TAB1	ØØØ4ØØØ	
TAB2	ØØ1ØØØØ	
TAB3	ØØ14ØØØ	Expand tabs to spaces.
BSDLY	ØØ2ØØØØ	Select backspace delays:
BSØ	Ø	
BS1	ØØ2ØØØØ	
VTDLY	ØØ4ØØØØ	Select vertical-tab delays:
VTØ	Ø	
VT1	ØØ4ØØØØ	
FFDLY	Ø1ØØØØØ	Select form-feed delays:
FFØ	Ø	
FF1	Ø1ØØØØØ	

J

If OPOST is set, output characters are post-processed as indicated by the remaining flags; otherwise, characters are transmitted without change.

If OLCUC is set, a lowercase alphabetic character is transmitted as the corresponding uppercase character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column Ø (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer is set to Ø and the delays specified for CR are used. Otherwise, the NL character is assumed to do just the line-feed function; the column pointer remains unchanged. The column pointer is also set to Ø if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases, a value of Ø indicates no delay. If OFILL is set, fill characters are transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL; otherwise, it is NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about two seconds.

J

Newline delay lasts about Ø.1Ø seconds. If ONLRET is set, the carriage-return delays are used instead of the newline delays. If OFILL is set, two fill characters are transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about Ø.1Ø seconds, and type 3 is about Ø.15 seconds. If OFILL is set, delay type 1 transmits two fill characters and type 2 transmits four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about Ø.1Ø seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters are transmitted for any delay.

Backspace delay lasts about Ø.Ø5 seconds. If OFILL is set, one fill character is transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

CONTROL MODES

The c_cflag field describes the hardware control of the terminal:

CBAUD	00000017	Baud rate:
B0	0	Hang up
B50	00000001	50 baud
B75	00000002	75 baud
B110	00000003	110 baud
B134	00000004	134.5 baud
B150	00000005	150 baud
B200	00000006	200 baud
B300	00000007	300 baud
B600	00000010	600 baud
B1200	00000011	1200 baud
B1800	00000012	1800 baud
B2400	00000013	2400 baud
B4800	00000014	4800 baud
B9600	00000015	9600 baud
EXTA	00000016	External A
EXTB	00000017	External B
CSIZE	00000060	Character size:
CS5	0	5 bits
CS6	00000020	6 bits
CS7	00000040	7 bits
CS8	00000060	8 bits
CSTOPB	00000100	Send two stop bits, else one.
CREAD	00000200	Enable receiver.
PARENB	00000400	Parity enable.
PARODD	00001000	Odd parity, else even.
HUPCL	00002000	Hang up on last close.
CLOCAL	00004000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal is not asserted. Normally, this disconnects the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used; otherwise, one stop bit is used. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled, and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set; otherwise, even parity is used.

If CREAD is set, the receiver is enabled; otherwise, no characters are received.

If HUPCL is set, the line is disconnected when the last process with the line open closes it or terminates, i.e., the data-terminal-ready signal is not asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. If it is not set, modem control is assumed.

The initial hardware control value after open is B3ØØ, CS8, CREAD, HUPCL.

LOCAL MODES

The c_lflag field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (Ø) provides the following:

ISIG	ØØØØØØ1	Enable signals.
ICANON	ØØØØØØ2	Canonical input (erase and kill processing).
XCASE	ØØØØØØ4	Canonical upper/lower presentation.
ECHO	ØØØØØ1Ø	Enable echo.
ECHOE	ØØØØØ2Ø	Echo erase character as BS-SP-BS.
ECHOK	ØØØØØ4Ø	Echo NL after kill character.
ECHONL	ØØØØ1ØØ	Echo NL.
NOFLSH	ØØØØ2ØØ	Disable flush after interrupt or quit.

J

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus, these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., Ø377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read is not satisfied until at least VMIN characters have been received or the timeout value VTIME has expired. This allows fast bursts of input to be read efficiently while still allowing single character input. The VMIN and VTIME values are stored in the positions for the EOF and EOL characters, respectively. The VTIME value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an uppercase letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	use:
`	\'
|	\!
^	\^
{	\(
}	\)
\	\\

For example, A is input as \a, \n as \\n, and \N as \\n.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which clears the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character is echoed after the kill character to emphasize that the line is deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character is echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters is not done.

The initial line-discipline control value is all bits clear.

I/O SYSTEM CALLS

The primary ioctl(2) system calls have the form:

```
ioctl(fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA Get the parameters associated with the terminal and store in the termio structure referenced by arg.

TCSETA Set the parameters associated with the terminal from the structure referenced by arg. The change is immediate.

TCSETAW Wait for the output to drain before setting new parameters. This form should be used when changing parameters that affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

Additional ioctl(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK Wait for the output to drain. If arg is Ø, then send a break (zero bits for Ø.25 seconds).

TCXONC Start/stop control. If arg is Ø, suspend output; if 1, restart suspended output; if 2, send XON character and assert RTS if hardware handshake is enabled; if 3, send XOFF character and negate RTS if hardware handshake is enabled. In addition, if arg is 4, assert RTS; if 5, negate RTS; if 6, assert DTR; if 7, negate DTR.

TCFLUSH If arg is Ø, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

APPENDIX K - MVME332XT DEVICE DRIVER INTERFACE

DESCRIPTION

The MVME332XT driver provides a general interface to the MVME332XT VMEbus Communication Controller module. The MVME332XT controller supports up to eight asynchronous serial communication ports and one Centronics compatible printer port. The MVME332XT driver supports up to 16 MVME332XT controllers per system.

Each peripheral device connected to the MVME332XT has the same major device number. The MVME332XT firmware presents a generic serial and printer device interface to the driver, which can treat them as identical devices except for error message generation and interpretation. The driver distinguishes a serial device from the printer device by its device unit number. Device unit numbers Ø through 7 are allocated for the eight serial devices and the printer is designated unit 8. The least significant 4 bits of the minor device field are interpreted as the device unit number. Therefore, 16 minor device numbers are required per MVME332XT controller. The four highest order bits of the minor device number are interpreted as the controller number.

read(2) Processing

The MVME332XT driver is much simpler than traditional serial I/O controller drivers since the line discipline functions are performed by the MVME332XT firmware. As such, the MVME332XT driver simply controls port modes via *ioctl(2)* calls and performs character buffer I/O to and from the MVME332XT shared RAM space via *copyout()* and *copyin()* subroutines. The MVME332XT firmware supplies the processed character data to the driver through the MVME332XT shared RAM on *read*(2) calls. The driver can then transfer the processed characters directly to the user buffer without incurring line discipline overhead.

When attempting to read from a MVME332XT device that has no data currently available:

> If O_NDELAY is set from a *open*(2) or *fcntl*(2) system call, the read returns a Ø.
>
> If O_NDELAY is clear, the read blocks until data becomes available.

write(2) Processing

Writes to a MVME332XT device proceed in a similar manner. The driver transfers unprocessed character data directly to the MVME332XT shared RAM with *write*(2) calls. The MVME332XT firmware performs any character processing required to complete the character I/O.

open(2) Processing

When the *open*(2) system call is made on an MVME332XT serial device or printer, the following processing will occur:

1. If the minor device number is illegal, then the *open*(2) will fail, returning the error status **ENXIO**.
2. If **O_NDELAY** flag is set, the open returns without waiting for the carrier (serial port case) or the printer select status (printer port case). In the serial port case, carrier status is indicated by the state of the channel's DCD signal. The analogous signal for the printer port is the SELECT line, which is asserted by the printer when it has been selected.
3. If **O_NDELAY** is clear, the open blocks until the carrier (serial port case) or printer select is present. If a signal is caught while waiting for the carrier or select, the open returns with error status **EINTR**.

close(2) Processing

Upon a *close*(2) of the MVME332XT device, the minor device number must be legal, or the *close*(2) will fail, returning the error status **ENXIO**.

ioctl(2) Processing

The MVME332XT driver supports the standard *ioctl*(2) commands that apply to terminal files and some conversion aid ioctl commands not described by *termio*(7). Several MVME332XT specific ioctl commands are also supported, which support hardware flow control and downloading of object code to the MVME332XT. Refer to the *m332xctl*(1M) manual pages and the **/usr/include/mvme332xt.h** file for more details on the MVME332XT specific *ioctl* commands.

The following *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
struct termio *arg
```

TCSETA Set the parameters associated with the terminal from the *termio* structure referenced by **arg**. The change is immediate.

TCSETAW Wait for the output to drain before setting the *termio* parameters. The **arg** contains a pointer to the *termio* structure.

TCSETAF Wait for the output to drain, then flush the input queue and set the *termio* parameters. The **arg** contains a pointer to the *termio* structure.

TCSETDF Set the default *termio* parameters. The **arg** contains a pointer to the *termio* structure.

TCGETDF Get the default *termio* parameters. The **arg** contains a pointer to the *termio* structure.

TCGETA Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.

The following conversion *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
struct termcb *arg
```

LDSETT Set the parameters associated with the terminal from the termcb structure referenced by **arg**.

LDGETT Get the parameters associated with the terminal and store in the termcb structure referenced by **arg**.

The following conversion *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
struct sgttyb *arg
```

TIOCSETP Set the parameters associated with the terminal from the sgttyb structure referenced by **arg**.

TIOCGETP Get the parameters associated with the terminal and store in the sgttyb structure referenced by **arg**.

The following *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
int arg
```

TCSBRK Wait for the output queue to drain. If **arg** is Ø, then send a break.

TCXONC Start/stop control. If **arg** is Ø, suspend output; if 1, restart suspended output; if 2, send a XOFF character and negate the RTS if hardware handshake is enabled; if 3, send a XON character and assert the RTS if hardware handshake is enabled; if 4, assert the RTS; if 5, negate the RTS; if 6, assert the DTR; and if 7, negate the DTR.

TCFLSH If **arg** is Ø, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

The following are MVME332XT specific *ioctl* commands. The *m332xctl*(1M) utility also provides a way for the user to perform downloads and set/get hardware handshake options. The following MVME332XT specific *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
struct dl_info *arg
```

The *dl_info* structure has the following format:

```
struct dl_info {
     unsigned long host_addr;     /* host address              */
     unsigned long ipc_addr;      /* IPC address               */
     unsigned long count;         /* number of bytes           */
                                  /* to be transferred         */
     unsigned long  extra_long;   /* reserved work area Ø      */
     unsigned short extra_short;  /* reserved work area 1      */
};
```

K

The downloadable area starting address is set or returned in *dl_info.ipc_addr* and the downloadable area size in bytes is returned in *dl_info.count*.

The commands using this form are:

TCGETDL Get download information from the MVME332XT. The **arg** is a pointer to a *dl_info* structure. The downloadable area information is returned in the *dl_info* structure.

TCDLOAD Download object code to the MVME332XT. The **arg** is a pointer to a *dl_info* structure described above. The *host_addr* points to the object code in the MVME332XT shared RAM. The *ipc_addr* points to the MVME332XT local RAM base address. The count is the number of bytes to be downloaded.

TCGETSYM Get symbol table from the MVME332XT. The **arg** is a pointer to a *dl_info* structure described above. The *host_addr* points to the buffer in the MVME332XT shared RAM for the MVME332XT to return the symbol table. The *ipc_addr* should be set to Ø for the first call of the TCGETSYM to indicate the beginning of the symbol table. It is updated by the MVME332XT for subsequent TCGETSYM command. At the end of the symbol table, the MVME332XT returns EOF in the *ipc_addr* field. The count is the number of bytes returned by the MVME332XT.

TCWHAT This command performs a what function (similar to the UNIX what utility) and returns a SCCS id (Source Code Control System ID). The **arg** is a pointer to a *dl_info* structure described above. The *host_addr* points to the buffer in the MVME332XT shared RAM. The *ipc_addr* should be set to Ø for the first call of the TCWHAT to indicate the start of the TCWHAT function. It is updated by the MVME332XT for subsequent TCWHAT command. At the end of dumping the SCCS id, the MVME332XT returns EOF in the *ipc_addr* field. The count is the number of bytes returned by the MVME332XT.

TCLINE Load line discipline linesw table to the MVME332XT internal data structure. The **arg** is set to point to the address of the *dl_info* structure. The *dl_info.ipc_addr* is where the linesw table starts. The *dl_info.count* is the number of lines the linesw tables contains.

The MVME332XT linesw structure is defined below:

```
struct linesw {
     int (*l_open)();   /* open function            */
     int (*l_icp)();    /* Input Character Process  */
     int (*l_ocp)();    /* Output Character Process */
     int (*l_ioctl)();  /* control function         */
     int (*l_close)();  /* close function           */
     int (*l_ctl)();    /* control process          */
     int (*l_gate)();   /* open/close process       */
};
```

TCEXEC Execute a user function that is downloaded by a previous ioctl **TCDLOAD** command. The **arg** needs to be set to point to the *dl_info* structure. The *dl_info.ipc_addr* is the execution function address.

The following MVME332XT specific *ioctl*(2) system call has the form:

```
ioctl(fildes, command, arg)
int arg
```

TCSETHW Set hardware flow control option. If **arg** is **1**, enable hardware flow control using the RTS/CTS signal pairs; if **arg** is **Ø**, disable hardware flow control.

The following MVME332XT specific *ioctl*(2) system calls have the form:

```
ioctl(fildes, command, arg)
int *arg
```

TCGETHW Return hardware flow control status. If the specified serial port has hardware flow control enabled, **1** is returned to the **arg** integer location; otherwise, **Ø** is returned.

TCGETVR Return MVME332XT firmware and driver version and revision number in the longword pointed to by **arg**. The driver version number is returned in the most significant byte, the driver revision number is in the second most significant byte, the firmware version number is in the third byte, and the firmware revision number is in the least significant byte.

Error Return Codes

There are several MVME332XT specific error codes returned in u.u_error by the MVME332XT when *open*(2), *close*(2), *read*(2), *write*(2), or *ioctl*(2) are called. The following error codes are defined in **/usr/include/sys/mvme332XT.h**:

ERR_CHAN_NO: Invalid Channel Number. The command or status channel number used in the packet is invalid.

ERR_CMD: Invalid Command. The command packet is invalid.

ERR_UNIT: Invalid Logical Unit Number. The device unit number is out of range; only **Ø** to **8** are allowed.

ERR_PARM: Invalid Parameter. The packet parameter is invalid.

Error Messages

The MVME332XT driver generates many different error messages. These error messages, printed in English, attempt to provide information to help the operator to diagnose problems. The error messages displayed by the MVME332XT have the following format:

MVME332XT: MESSAGE on controller *X*, unit *Y*

where **MESSAGE** is the error message describing the symptoms, *X* is the controller number, and *Y* is the unit number.

The following are descriptions for the returned **MESSAGE**. Note that some messages include a second line, which indicates that the associated MVME332XT controller has been disabled as a result of the error condition.

Create channel error
Controller X disabled

The channel, or communication link between the driver and the MVME332XT, was not successfully created. The driver must establish the channel interface before any commands can be dispatched to the MVME332XT. This error condition typically indicates a MVME332XT configuration problem or malfunction. The controller is subsequently marked bad by the driver and any further access attempts are disallowed.

Initialization error
Controller X disabled

An error was reported by the MVME332XT controller when the driver sent an initialization command to it. This condition will result if the driver attempts to size one of the MVME332XT read/write rings to a non base 2 value.

Unknown interrupt

An interrupt occurred from a MVME332XT controller that was marked bad or nonexistent.

Corrupted envelopes

This indicates channel corruption in the MVME332XT shared RAM.

PRINTER is deselected

This message indicates that the printer is deselected. Check the printer select switch.

PRINTER is out of paper

This indicates that the printer is out of paper. Check the printer paper supply.

PRINTER fault for unknown reason

This indicates a printer error other than the paper out or the deselected printer error conditions. Check the printer connections or refer to the printer manufacturer's user manual.

Free packet pool is empty

This error message is printed out when the driver needs to send a command to the MVME332XT device but has no free packet to use due to a MVME332XT or system level malfunction.

APPENDIX L - m332xctl CONTROL UTILITY

SYNOPSIS

m332xctl {-t | -r | -R | -D | -h < on|off|info> | -g |
-s | [-d < dlfile> {-x < sname>] . . . -1 | -e < fname> }}dev

DESCRIPTION

m332xctl provides a functional control interface to the MVME332XT Communications Controller. Note that *m332xctl* provides no support for the MVME332 hardware and firmware architecture. The following options and fields are interpreted by *m332xctl:*

-t Test the existence of the MVME332XT. Return Ø if it exists, else return ENXIO.

-r Get firmware and driver version and revision numbers. The designated **dev** should be the printer device.

-R Get firmware version number in short format. The designated **dev** should be the printer device.

-D Debug mode.

-g Get downloadable area information from the MVME332XT controller. The address and size of the download area is displayed. The designated **dev** should be the printer device.

-s Get symbol table of the MVME332XT firmware and display. The designated **dev** should be the printer device.

-h Hardware flow control handshaking can be enabled or disabled, and hardware flow control port status can be queried. Hardware flow control is implemented with the RS-232C RTS and CTS handshakes.

-d Download a coff file to the MVME332XT. The designated **dev** should be the printer device.

-x Exclude a section when downloading. Up to sixteen sections may be excluded for a particular download operation. This option must be preceded by the **-d** option in the command invocation.

-l Instruct the MVME332XT firmware to copy the downloaded line switch table to its internal data structure. This option must be preceded by the **-d** option in the command invocation.

-e Instruct the MVME332XT firmware to execute a user function in a downloaded file. This option must be preceded by the **-d** option in the command invocation.

dev MVME332XT serial I/O or printer device. **dev** should be the printer device for the **-g**, **-d**, **-r**, **-R**, **-s**, **-l**, and **-x** options.

dlfile Coff compatible file that is to be linked to the MVME332XT symbol table before downloading.

sname Section names to be excluded when **dlfile** is downloaded.

fname Function within the **dlfile** that is to be executed.

To obtain the MVME332XT firmware version and revision number, execute the following *m332xctl* command:

m332xctl -R /dev/m332x*XY*

This command issues a mesage of the form:

VR

where *V* and *R* are the MVME332XT firmware version and revision numbers, respectively.

The *m332xctl* command

m332xctl -h on /dev/m332x*XY*

enables hardware flow control option for the specified serial I/O port. The I/O device to be set is designated by the *XY* field throughout this document, where *X* and *Y* refer to the MVME332XT controller and port device numbers, respectively.

Hardware flow control for any MVME332XT serial port may be disabled by issuing

m332xctl -h off /dev/m332x*XY*.

Hardware flow control is implemented with the RS-232 RTS/CTS signal pairs. In this mode, a serial port transmitter is disabled when its CTS input negates and a receiver negates its RTS output when the associated receive channel character high water mark has been reached. A MVME332XT serial port hardware flow control

configuration may be determined with the following *m332xctl* command.

m332xctl -h info /dev/m332x*XY*

In this example, if hardware flow control is enabled for the specified port, the following message will be sent to standard output.

hardware handshake is enabled

If hardware flow control is disabled for the specified port, the following message will be sent to standard output.

hardware handshake is disabled

To get the start address and size of the MVME332XT download area, use the following *m332xctl* command.

m332xctl -g /dev/m332x*XY*

where **/dev/m332x*XY*** must be the MVME332XT printer device. This restricts downloading and download area information access to root.

The following information is displayed in response to the previous command:

Downloadable area start address = *AAAA*, size = *SSSS*

The downloadable coff file should be linked to the displayed start address before downloading to the MVME332XT firmware, using the following syntax:

m332xctl -d dlfile /dev/m332x*XY*

where **dlfile** is the coff file to be downloaded and **/dev/m332x*XY*** is the MVME332XT printer device name, required for security purposes.

To exclude sections of **dlfile** during the download operation, use

m332xctl -d dlfile -x sname1 . . . -x sname*n* /dev/m332x*XY*

where **sname1, . . . , sname*n*** are the section names that are to be excluded during the download operation. The *m332xctl* command supports up to 16 excluded section names using the syntax shown.

The MVME332XT firmware supports user supplied line disciplines via the *m332xctl* **-d** and **-l** options, which allow the downloaded line switch table to be copied to the MVME332XT firmware data structures, as follows:

m332xctl -d dlfile -l /dev/m332x*XY*

where **dlfile** is the download file name and **/dev/m332x*XY*** is the MVME332XT printer device special file name, as before. The downloaded **dlfile** must contain the following symbols:

Symbol	Description
- linetable :	Linesw lineswitch table
- linecount :	Number of lines to be downloaded

Refer to *mvme332xt.7* for discussion regarding linesw table structure. Notice that the linesw table structure defined in *mvme332xt.7* differs from that described in **/usr/include/sys/conf.h**. Intimate familiarity with the MVME332XT firmware architecture is required to successfully port a user developed line discipline.

To download a coff file, **dlfile**, to the MVME332XT and execute a downloaded function, **fname**, use the following syntax:

m332xctl -d dlfile -e fname /dev/m332x*XY*

where **dlfile** is the downloaded file, **fname** is the function to be executed by the MVME332XT firmware, and **/dev/m332x*XY*** is the MVME332XT printer device name. Refer to *mvme332xt(7)* for more information regarding special file naming conventions.

The **-D** option enables the debug mode. Option **-DD** enables the debug mode at level 2, which results in more comprehensive debug messages. Either mode is useful for monitoring a downloading operation and for debugging user developed lineswitch and function routines.

FILES

/dev/m332x*

SEE ALSO

termio(7), tty(7), ioctl(2), stty(1), mvme332xt(7).

APPENDIX M - IOCTL COMMAND OUTPUT (TCWHAT)

The following is a sample output of the TCWHAT command packet or "m332xctl -r" SYSTEM V/68 utility command.

```
@(#)close2.c          7.2
@(#)cs_topri.c        7.1
@(#)cs_tostd.c        7.1
@(#)ctl2.c            7.2
@(#)dl_icpØ.c         7.1
@(#)dl_ocpØ.c         7.1
@(#)dlinit.c          7.2
@(#)event2.c          7.3
@(#)freecore.c        7.1
@(#)gate2.c           7.1
@(#)getcore.c         7.1
@(#)icp_dual.c        7.1
@(#)icp_single.c      7.1
@(#)ioctl2.c          7.4
@(#)ispspace.c        7.1
@(#)linesw.c          7.1
@(#)lowcore.s         7.1
@(#)m564intr.s        7.1
@(#)m564param.c       7.3
@(#)ocp2.c            7.2
@(#)open2.c           7.1
@(#)ttre2i.c          7.1
@(#)ttri2e.c          7.1
@(#)ttri2xs.c         7.1
@(#)ttrxput.c         7.1
@(#)ttxfcs.c          7.1
@(#)tty2.c            7.1
```

M

APPENDIX N - IOCTL COMMAND OUTPUT (TCGETSYM)

The following is a sample output of the TCGETSYM command packet or "m332xctl -s" SYSTEM V/68 utility command. This formatted output can be included in the linkage editor (ld) command to produce a downloadable module for the MVME332XT.

```
MEMORY
{
        dl_area (RW) : o = 0x00f20688, l = 0x0000f978
}
SECTIONS
{
        GROUP : {
                .text : {
                }
                .data : {
                }
                .bss : {
                }
        } > dl_area
}
reset_vector        =       0x00fc0000;
ctstart             =       0x00fc04a2;
ct_exception        =       0x00fc04bc;
ctmain              =       0x00fc04ec;
ctinit              =       0x00fc0628;
burnin              =       0x00fc0682;
afail               =       0x00fc06b0;
print               =       0x00fc06dc;
printf              =       0x00fc0a92;
sprintf             =       0x00fc0aae;
memput              =       0x00fc0ae0;
putc                =       0x00fc0af0;
strncmp             =       0x00fc0b3c;
ctcpul              =       0x00fc0b7c;
ctram_walk          =       0x00fc0ce8;
ctram_byteword      =       0x00fc0d42;
ctram_march         =       0x00fc0da0;
ctram_nondes        =       0x00fc0e2a;
ctshram_walk        =       0x00fc0e8c;
_send               =       0x00fc21a2;
_put_user           =       0x00fc2396;
_dispatch           =       0x00fc23e2;
_signal             =       0x00fc250a;
```

N

```
_deq            =       0x00fc25e2;
lzero           =       0x00fc59c8;
lfill           =       0x00fc59f0;
spl7            =       0x00fc5a4c;
m230param       =       0x00fc623a;
m564putc        =       0x00fc6a00;
ocp0            =       0x00fc6ab6;
ttytab          =       0x00f0080c;
linesw          =       0x00f0e90c;
linecnt         =       0x00f0e9d0;
maxline         =       0x00f0e9d1;
termcnt         =       0x00f0e9d2;
ct_desc         =       0x00f30010;
```

N

APPENDIX O - FIRMWARE FUNCTION SUB-ROUTINES

Table O-1 provides a list of the firmware functions callable by downloadable line disciplines. The addresses of these functions can be obtained by using a TCGETSYM command packet or by the **m332xctl** utility command available under SYSTEM V/68 (refer to Appendix L of this manual).

TABLE O-1. FIRMWARE FUNCTION SUB-ROUTINES

Name	Description
openØ	Line Ø OPEN function, called by the Bpp_Receiver.
closeØ	Line Ø CLOSE function, called by the Bpp_Receiver.
ioctlØ	Line Ø IOCTL function, called by the Bpp_Receiver.
gateØ	Line Ø GATE function, called by the GATE process.
ctlØ	Line Ø CTL function, called by the CTL process.
icpØ	Line Ø ICP function, called by the ICP process.
ocpØ	Line Ø OCP function, called by the OCP process.
icp1	Line 1 ICP function, called by the ICP process.
ocp1	Line 1 OCP function, called by the OCP process.
bpprtn	Return a packet to the host.
ttywait	Wait for both WRITE ring and OUTPUT ring to drain. Should be called by a process.
ttyflush	Flush input rings or output rings or both.
change_ract	Change Receive-Action-Table based on termio structure.
bcopy	Copy a block of data to another place in byte mode.
wcopy	Copy a block of data to another place in word mode.
lcopy	Copy a block of data to another place in long word mode.
bzero	Clear a block of memory in byte mode.
lzero	Clear a block of memory in long word mode.
bfill	Fill a block of memory with a byte pattern.
lfill	Fill a block of memory with a long word pattern.

O

TABLE 0-1. FIRMWARE FUNCTION SUB-ROUTINES (cont.)

Name	Description
spl[Ø-7]	Set process interrupt level to Ø,...,7, return old level.
splx	Set process interrupt level to X, return old level.
splattn	Mask the attention interrupt.
splpr	Mask the printer device interrupt.
spltimer	Mask the tick timer interrupt.
spltty	Mask the serial device interrupt.
splhi	Mask all interrupts.
getvbr	Return the content of CPU's VBR (Vector Base Register).
getsr	Return the content of CPU's SR (Status Register).
setvec	Set up an interrupt handler for a vector.
strncmp	String compare.
strcopy	String copy.
strlen	String length.
m564putc	Send a character to a port using polling mode, useful only for debugging.
m23Øputc	Send a character to the printer port (for debugging only).
printf	Formatted print a message into DUMP-AREA on dual-port memory.
sprintf	Same as printf but into specified buffer.
print	Printf and sprintf core function.

APPENDIX P - KERNEL FUNCTION PRIMITIVES

Table P-1 provides a list of kernel function primitives. The details of these can be found in the ADC Kernel Firmware Manual.

TABLE P-1. KERNEL FUNCTION PRIMITIVES

Name	Description
_aging	Age the ready list (called by the tick timer interrupt).
_can_timeout	Cancel an event timeout (used as a watchdog timer).
_create	Create a process ready to run.
_cycle	Voluntarily relinquish a time slide.
_delay	Put the running process to sleep for a number of ticks.
_deq	Dequeue an item from a linked list.
_dispatch	Activate a process from the ready list to run.
_enq	Enqueue an item into a linked list.
_exit	Remove the running process from existence.
_freemem	Deallocate a block of memory, return it to free list.
_get_user	Returns the user value from the running process' PCB.
_getbuf	Allocate a buffer from a fixed buffer poll.
_getmem	Allocate a block of memory from the free list.
_halt	Mask all interrupts then halt the CPU.
_kerinit	Initialize the kernel data structure.
_link	Create a link list from a memory poll.
_mask	Mask the processor interrupt level to kernel level.
_nullmgr	Is a place holder for an idle system.
_prenq	Enqueue a process onto the ready list based on priority.
_put_user	Load a used value into a specified PCB.
_putbuf	Deallocate a buffer to a fixed buffer pool.
_receive	Receive a standard message.
_recvptr	Receive a short form message.
_send	Send a standard message.
_sendptr	Send a short form message.
_set_timeout	Start a watchdog timeout.
_signal	Signal a resource/event semaphore.
_sleep	Sleep on an event address.
_stop	Unmask all interrupts, stop the CPU until interrupted.

TABLE P-1. KERNEL FUNCTION PRIMITIVES (cont.)

Name	Description
_swap	Swap a process image to another.
_timer_int	Kernel timer interrupt housekeeping.
_unmask	Unmask the processor interrupt level.
_wait	Wait on a resource/event semaphore.
_wakeup	Wake up all processes waiting on an event address.