# Hardware Information



## Special Registers

There are nine specia[illegible]

| | |
|---|---|
| `FLAGS` | [illegible]ing all of the one-bit flags |
| `PDBR` | [illegible]Register |
| `INTVEC` | [illegible]errupt vector |
| `CGBR` | [illegible]er |
| `CGLEN` | [illegible] |
| `DEBUG` | [illegible]r = this value, a debug interrupt is signalled |
| `TIMER` | Reduced by 1 after each instruction, causes timer interrupt when zero |
| `SYSSP` | System stack pointer. If in system mode, equivalent to SP |
| `SYSFP` | System frame pointer, not so useful. |
| `USRSP` | User mode stack pointer. If in user mode, equivalent to SP |
| `USRFP` | User mode frame pointer. |



The assembler understands the names of these registers (put a `$` sign in front), they stand for the numbers 0 to 9 in instruction operands.



There are two instructions that directly access the special registers:

| | |
|---|---|
| `GETSR` | loads a special register value into a normal register |
| `SETSR` | stores a normal register value into a special register |



Example: how to set the `TIMER` register to 100:

```
LOAD    R1, 100
SETSR   R1, $TIMER
```



The value stored in `$PDBR` is always treated as a physical memory address.
The values stored in `$INTVEC` and `$CGBR` are physical addresses, but the entries in the interrupt vector and the call gate vector must be virtual addresses if virtual memory is turned on.
`$DEBUG`, `$SYSSP`, and `$SYSFP` are treated as virtual addresses when virtual memory is turned on.



## Flags

There are seven one-bit CPU flags, as follows

| | |
|---|---|
| `R` | Indicates that the CPU is running, not halted |
| `Z` | Zero. Set by some instructions to indicate a zero (or equal) result. |
| `N` | Negative. Set by some instructions to indicate a negative result. |
| `ERR` | Error. Used only by the `PERI` instruction. Zero means success. |
| `SYS` | Set when CPU is in system mode, Zero when in user mode. |
| `IP` | Interrupt in progress. Set to 1 to ignore interrupts. |
| `VM` | Virtual Memory. If zero, all memory accesses use physical addresses, if set, page tables must be correctly set up, all memory addresses are translated. |

The final three, `SYS`, `IP`, and `VM`, may only be modified when the CPU is in system mode.
At start-up, `SYS=1`, `IP=1`, `VM=0`.

The assembler understands the names of these flags (put a $ sign in front): they stand for the numbers 0 to 6 in instruction operands.

There are two instruc[illegible]ess the special registers:

GETFL loads t[illegible]lag into a register

SETFL sets a [illegible] register value (0 for off, non-0 for on)

The COMP and COMPZ [illegible]ear both Z and N, depending on the result.
The JCOND instructio[illegible]ave a particular combination of values.

All the flag values r[illegible], using the GETSR instruction on the $FLAGS special register. The flags occupy the least significant bits of the value, in the order shown above. R is the least significant bit, VM is bit 6 (equivalent value 64).

All the flag values may be set at once using the SETSR instruction on the $FLAGS special register.

Example: Turn the SYS flag off, and the VM flag on, leaving other flags untouched:

```
GETSR    R1, $FLAGS
CBIT     R1, $SYS
SBIT     R1, $VM
SETSR    R1, $FLAGS
```

The special instruction FLAGSJ sets all the flags at once, and causes an unconditional jump by setting the PC. The only real point of this weird instruction is that it lets you turn on virtual memory without crashing the system. As soon as the VM flag is turned on, virtual-to-physical address translation begins for all memory accesses, so in the example above, if the program counter = 101 for the first instruction the GETSR is fetched from physical location 101, the CBIT is fetched from physical location 102, the SBIT is fetched from physical location 103, then suddenly physical addresses are not used any more, and the next instruction is fetched from *virtual* address 104. Unless virtual address 104 maps to physical address 104 (which would not make much sense), everything fails. This sequence:

```
GETSR    R1, $FLAGS
CBIT     R1, $SYS
SBIT     R1, $VM
FLAGSJ   R1, xxx
```

is safe. Of course ‘xxx’ should be replaced by the correct virtual address for program continuation.

## Bit Ranges

The instructions EXBR, DPBR, etc extract or deposit a sequence of consecutive bits from within a single word. The desired bits are described by a single 32 bit value constructed thus:

5 least significant bits: number of bits in the range, with 00000 indicating 32.
5 next bits: the number of bits to the right of the range within its word.
22 most significant bits: the number of whole words to be skipped before extracting the bits.

`EXBR` and `DPBR` work directly on their operand, so the 22 most significant bits are ignored. `EXBRV` and `DPBRV` treat their operand as the address of the first word in a segment of memory.

## INTERRUPTS

There are interrupts [illegible] problem (such as a user mode program attempting a privileged operation) [illegible]upts that represent some useful notification (such as keyboard input ready [illegible]r reached zero). If interrupts are being processed (that is, the `IP` flag is 0, [illegible]cial register contains the address of a proper interrupt vector), then all interrupts are trappable, regardless of how fatal they are.

If interrupts are being ignored (`IP` flag is 1), then fatal interrupts still stop a program, but notification interrupts are just ignored.

If interrupts are being accepted (`IP`=0) and a particular interrupt arises, but the interrupt vector is invalid, a second interrupt, `INTRFAULT`, is signalled. This may also be trapped, but given that it is caused by the failure to correctly process another interrupt, it will probably turn out to be fatal.

Beware of this. Problems with regular programs (system or user mode) cause interrupts, and that is fine. The interrupt gives the system a chance to correct whatever condition caused it. BUT interrupt handling functions have no backup. If an interrupt handler causes a non-trivial interrupt, even a page fault, it will normally be fatal.

The `INTRFAULT` interrupt is the last chance to avoid a big crash. If you have a handling function for `INTRFAULT` stored in the interrupt vector, it will be called if a fatal interrupt occurs during interrupt processing, but it will not be able to return to processing the original interrupt after fixing the situation.

There are 14 interrupts defined, each with a name known to the assembler. Their names all begin with "`IV$`". An interrupt vector is really an array, and must be at least 14 words long. To be used, its address must be stored in the special register `INTVEC`. Each entry in the array is either zero (the corresponding interrupt will not be handled) or the address of an almost perfectly normal function that will be called automatically whenever the relevant interrupt occurs. The only special requirement is that interrupt handling functions must use `IRET` in all places instead of `RET`.

The defined interrupts are:

| | | |
|---|---|---|
| `IV$NONE` | = `0`: | (not a real interrupt code) |
| `IV$MEMORY` | = `1`: | Physical memory access failed |
| `IV$PAGEFAULT` | = `2`: | Page fault |
| `IV$UNIMPOP` | = `3`: | Unimplemented operation code (i.e. instruction opcode wrong) |
| `IV$HALT` | = `4`: | `HALT` instruction executed |
| `IV$DIVZERO` | = `5`: | Division by zero |
| `IV$UNWROP` | = `6`: | Unwritable instruction operand (e.g. `INC 72`) |
| `IV$TIMER` | = `7`: | Countdown timer reached zero |
| `IV$PRIVOP` | = `8`: | Privileged operation attempted by user mode program |
| `IV$KEYBD` | = `9`: | at least one keyboard character typed and ready |
| `IV$BADCALL` | = `10`: | Bad `SYSCALL` index (i.e. `<0` or `>=$CGLEN`) |

```
IV$PAGEPRIV   =  11:  User mode access to system mode page
IV$DEBUG      =  12:  PC=$DEBUG trap
IV$INTRFAULT  =  13:  Failure to process interrupt.
```

The `IV$` values are [illegible] interrupt vector where the handler function’s address should be stored.

Example: How to set [illegible]er that automatically prints a dot whenever a keyboard key is pressed, and a [illegible]r 5000 instructions have been executed...

```
  LOAD      R[illegible]
  STORE     R1, [IVEC+IV$TIMER]
  LOAD      R1, KBHANDLER
  STORE     R1, [IVEC+IV$KEYBD]
  LOAD      R1, IVEC
  SETSR     R1, $INTVEC
  LOAD      R1, 0
  SETFL     R1, $IP
  LOAD      R1, 5000
  SETSR     R1, $TIMER
         ......

TIMHANDLER:
  LOAD      R1, ‘*’
  CALL      PRINTCHARACTER       // which you would have to write somewhere
  LOAD      R1, 5000
  SETSR     R1, $TIMER
  IRET

KBHANDLER:
  LOAD      R1, '.'
  CALL      PRINTCHARACTER
```

NOTE this interrupt will be repeatedly signalled until the character is consumed.

```
  IRET

IVEC:
  .SPACE    16
```

<u>ACTIONS AUTOMATICALLY PERFORMED WHEN AN INTERRUPT OCCURS, IF IP FLAG IS 0.</u>

oldflags = `FLAGS` register
flag `SYS` turned on. (i.e. now using system `SP` and system stack)
flag `IP` turned on.
`PUSH R0`
`PUSH R1`
...

```
...
PUSH R11
PUSH R12
PUSH SP
PUSH [illegible]
PUSH [illegible]
PUSH [illegible] information if available
PUSH [illegible]ress
PUSH [illegible]sition in interrupt vector)
PUSH [illegible]
PUSH [illegible]
PC = memory[$INTVEC + interrupt code]
```

These are exactly the same as the SYSCALL actions, except for the three values pushed after the 16 registers. These are information that may be needed to correctly handle the interrupt.



Note that if the interrupt handler behaves like a normal function, and performs "PUSH FP" and "LOAD FP, SP" as its first actions, then those three pieces of information will be available at [FP+3], [FP+4], and [FP+5], the locations of the first three parameters to a function in BCPL.



```
let interrupt_handler(intcode, address, info) be
{ ...
  ireturn; }
```



The first parameter is always the interrupt code, the IV$ value for the interrupt.

For the following interrupts:



PAGEFAULT, PAGEPRIV,

the second parameter is the virtual address that caused the problem.

For this interrupt:



MEMORY,

the second parameter is the physical address that caused the problem.

For the following interrupts:

UNIMPOP, HALT, DIVZERO, UNWROP, PRIVOP, BADCALL, DEBUG,

the second parameter is the address of the instruction that caused the problem (i.e. PC value).

For this interrupt:

BADCALL,

the third parameter is the operand of the SYSCALL instruction that caused the problem.

For this interrupt:

INTRFAULT,

which is only caused by a fatal error during interrupt processing, the second parameter is left unchanged from the original interrupt's setting, and the third parameter is set to the interrupt code for the original interrupt.

Realise that if each process has its own system stack then each process must also have its own value for the system stack pointer, which must be saved and restored when processes are switched.

## INPUT AND OUTPU[illegible]

All interactions w[illegible]side of the CPU are controlled by the PERI instruction. There are four genera[illegible]ions supported:

Disc Operations: [illegible] access to the emulated disc drives, permitting whole blocks (128 words, w[illegible] as 512 bytes) to be transferred between memory and a specified location on the disc. These operations are necessary for file-system implementation.

Magnetic Tape Operations: These provide a realistic way of accessing files in the real (i.e. outside the emulator, probably unix) file system. Without these it would be very difficult and time consuming to get useful test data into your own file system implementations.



Terminal Operations: These allow characters to be read from the controlling keyboard or written to appear on the monitor.

Network operations: these allow the use of a simulated IP network with six-byte IP addresses, using the real UDP interface to support it. Datagrams of up to 1024 words may be sent or received.



Time Operations: reading the emulated hardware clock and telling you the date and time.



All IO operations are controlled in the same way. A small lump of memory is filled with information describing the operation to be performed, and with space to receive the results. The PERI instruction sends these few words to the appropriate piece of hardware. When the operation is complete, data returned by the hardware, if any, is stored back into the small lump of memory, a success-or-error code (zero or positive for success, negative for failure) is put into the instruction's main register, and execution continues. The ERR flag is also cleared for success and set for failure.



Example: Finding the total size of disc drive number one.



The DISCCHECK IO operation requires a two-word control structure. All IO control structures must have the required operation code, in this case $DISCCHECK, stored in the first word. This particular operation also requires the second word to contain the disc drive number.

```
              LOAD     R2,    control
              LOAD     R1,    $DISCCHECK
              STORE    R1,    [R2]
              LOAD     R1,    1
              STORE    R1,    [R2+1]
              PERI     R3,    control
              JCOND    ERR,   failed
              ...etc...
      control:         .SPACE 3
```

If the operation is not successful, the ERR flag will be set, and the program will jump to the "failed:" label to deal with the situation, and R3 will contain a negative number as an error code. If the operation is successful, then R3 will contain the total number of blocks in disc number 1.

Of course, control structures may be set up in advance, like this:

```
            PERI     R3,   control
            JCOND    ERR,  failed
            ...etc
    control:         [illegible]EDISC
            .DA[illegible]
```

This style requires fe[illegible]s slightly less flexible.
PERI is a privileged [illegible]t be executed in user mode.

If the operation co[illegible]d, nothing happens except that the error code -1 (ERR_BAD_CODE) is stored in the register and the ERR flag is set.

## Disc Operations



Disc drives are set up at system initialisation. The system.setup file describes the disc drives that are needed. An example line from system.setup is "disc 1 maindrive 6000", it means that disc drive number 1 should be at least 6000 blocks long, and will actually be kept in the real file maindrive.disc. If such a file does not exist, it is created. If the file does exist, it is used as-is. The size of maindrive.disc will of course be 6000*512 bytes. The disc file is not actually created until it is first accessed, and even then it is only made big enough to store the blocks that have so far been written. Reading from a block that has never been written is *not* an error.



### $DISCCHECK



Requires a 2 word control structure, as follows
0: the value $DISCCHECK
1: disc drive number



Error codes (returned in register):
-2, ERR_READ_PARAMS: memory access problem reading the control structure.
-3, ERR_DEV_NUMBER: drive number < 1 or > 8.

Successful result (returned in register):
disc size, in blocks, or
0 if the indicated disc does not exist.

### $DISCREAD

Requires a 5 word control structure, as follows
0: the value $DISCREAD
1: disc drive number
2: (disc address) the number of the first block to be read
3: the number of consecutive blocks to be read
4: (memory address) the address into which the data should be stored.
make sure that there are at least (number of blocks * 128) words of space there.

Error codes (returned in register):


`-2, ERR_READ_PARAMS:` memory access problem reading the control structure.
`-3, ERR_DEV_NUMBER:` indicated disc not available
`-4, ERR_POSITION:` attempt to read a block number < 0 or >= size.
`-5, ERR_MEMORY:` memory access problem reading the data

Successful result (returned in register):
number of blocks transferred from disc to memory.

### $DISCWRITE

Requires a 5 word control structure, as follows
0: the value `$DISCWRITE`
1: disc drive number


2: (disc address) the number of the first block to be written
3: the number of consecutive blocks to be written
4: (memory address) the address of the data to be written.
make sure that there are at least (number of blocks * 128) words of space there.


Error codes same as for `$DISCREAD` plus
`-6, ERR_DEV_FAILED:` real failure to write all the data


Successful result (returned in register):
number of blocks transferred from memory to disc.


## Magnetic Tape Operations



Real files in the outside operating system are made available in the guise of magnetic tapes. To access a real file, a program must first load that file onto a tape drive. It may then either read from the file sequentially in units of 128 word blocks, or it may write units of 128 word blocks into the file. Finally, the tape drive must be unloaded. Files/tapes are automatically rewound to the beginning when they are loaded.

Magnetic tape drives are referred to by their unit number in the range 1 to 8. All blocks on a tape must be exactly 128 words (512 bytes), except that the last one may be smaller because they correspond to real files whose sizes are fixed.

### $TAPECHECK

Requires a 2 word control structure, as follows
0: the value `$TAPECHECK`
1: tape unit number

Error codes (returned in register):
`-2, ERR_READ_PARAMS:` memory access problem reading the control structure.
`-3, ERR_DEV_NUMBER:` drive number < 1 or > 8.

Successful result (returned in register):

'R' if the tape is readable,
'W' if the tape is writable, or
0 if the indicated tape has not been loaded.



### $TAPEREWIND

Requires a 2 w[illegible], as follows

0: the valu[illegible]

1: the tape [illegible]

Error codes (returned in register):

| | | |
|---|---|---|
| -2, | ERR_READ_PARAMS: | memory access problem reading the control structure. |
| -3, | ERR_DEV_NUMBER: | tape unit not available. |



Successful result (returned in register):

1



### $TAPELOAD

Requires a 4 word control structure, as follows

0: the value $TAPELOAD

1: the tape unit number

2: pointer to a string containing the real file name on the host system

3: mode, either 'R' for read only or 'W' for write only



Error codes (returned in register):

| | | |
|---|---|---|
| -2, | ERR_READ_PARAMS: | memory access problem reading the control structure. |
| -3, | ERR_DEV_NUMBER: | tape unit not available. |
| -5, | ERR_MEMORY: | memory access problem reading the filename string |
| -7, | ERR_NOT_FOUND: | the file is not accessible. |
| -8, | ERR_BAD_PARAM: | mode is neither 'R' nor 'W'. |



Successful result (returned in register):

1

### $TAPELENGTH

Requires a 2 word control structure, as follows

0: the value $TAPELENGTH

1: the tape unit number

Error codes (returned in register):

| | | |
|---|---|---|
| -2, | ERR_READ_PARAMS: | memory access problem reading the control structure. |
| -3, | ERR_DEV_NUMBER: | tape unit not available. |
| -7, | ERR_NOT_FOUND: | the tape is not loaded or the associated file is not accessible. |

Successful result (returned in register):
Length in bytes of the real file on the host system.



$TAPEUNLOAD

Requires a 2 w[illegible], as follows
0: the valu[illegible]
1: the tape [illegible]

Error codes (r[illegible]
-2, ERR_RE[illegible]ory access problem reading the control structure.
-3, ERR_DE[illegible] tape unit not available.
-7, ERR_NOT_FOUND: tape unit was not loaded.

Successful result (returned in register):
1



$TAPEREAD



Reads the next block from tape into memory

Requires a 3 word control structure, as follows
0: the value $TAPEREAD
1: tape unit number
2: (memory address) the address into which the data should be stored.
make sure that there are at least 128 words of space there.



Error codes (returned in register):
-2, ERR_READ_PARAMS: memory access problem reading the control structure.
-3, ERR_DEV_NUMBER: tape unit not available
-5, ERR_MEMORY: memory access problem reading the data



Successful result (returned in register):
number of bytes transferred from tape to memory, or
0 if the end of the tape had already been reached.

$TAPEWRITE

Requires a 4 word control structure, as follows
0: the value $TAPEWRITE
1: tape unit number
2: (memory address) the address of the data to be written.
3: the number of bytes to be written

Error codes same as for $DISCREAD plus
-6, ERR_DEV_FAILED: real failure to write all the data

Successful result (returned in register):
number of bytes transferred from memory to tape.



Example: Reading the first 512 characters from a real unix file and displaying them.



```
LOAD     [illegible]
LOAD     [illegible]
STORE    [illegible]
LOAD     [illegible]                 // unit number
STORE    [illegible]
LOAD     [illegible]
STORE    R2,  [R1+2]
LOAD     R2,  'R'
STORE    R2,  [R1+3]                 // READ ONLY
PERI     R3,  control                // have the tape loaded
JCOND    ERR, failed

LOAD     R2,  $TAPEREAD
STORE    R2,  [R1]
LOAD     R2,  1                      // unit number
STORE    R2,  [R1+1]
LOAD     R2,  space                  // where to put those characters
STORE    R2,  [R1+2]
PERI     R3,  control                // read from the tape
JCOND    ERR, failed

LOAD     R2,  $TERMOUTC
STORE    R2,  [R1]
LOAD     R2,  512                    // number of characters
STORE    R2,  [R1+1]
LOAD     R2,  space                  // where those characters are
STORE    R2,  [R1+2]
PERI     R3,  control                // print

LOAD     R2,  $TAPEUNLOAD
STORE    R2,  [R1]
LOAD     R2,  1                      // unit number
STORE    R2,  [R1+1]
PERI     R3,  control                // close the real file
HALT

filename:
    .STRING "tests/file.txt"
control:
    .SPACE  4
space:
    .SPACE  128
```

## Terminal Operations

There are two essent[illegible] a bunch of characters from the keyboard and write a bunch of characters [illegible]ead function is compatible with interrupt-driven user input: when a progra[illegible] single stepping) and interrupts are enabled, every time a keyboard key is pr[illegible] is added to the end of the hardware keyboard buffer and a `$KEYBD` inte[illegible]The `$TERMINC` operation takes characters from the beginning of the hard[illegible]r.

Character codes a[illegible] as the key is pressed, the system does not wait until a whole line is available. This means that any special behaviour associated with particular keys (such as ENTER or BACKSPACE) must be programmed. The one exception is control-c; that will always interrupt a running program and return to single stepping mode.



### $TERMINC



Requires 3 word control structure, as follows

0: the value `$TERMINC`

1: the maximum number of characters to be read

2: (memory address) the address into which the characters should be stored.
make sure that there are at least ((maximum number + 1) / 4) words of space there.



Error codes (returned in register):

`-2, ERR_READ_PARAMS:` memory access problem reading the control structure.

`-5, ERR_MEMORY:` memory access problem storing the characters



Successful result (returned in register):
number of characters actually read.



Notes:

It is not an error to attempt to read when the keyboard buffer is empty.

If no characters are already in the keyboard buffer, it will not wait for input.

The characters received are packed four per word to make a proper string, and that string will be zero terminated. Strings are organised so that the first character goes in the least-significant bits of the first word. This means that if just a single character is read, the first word of the result will simply be its ASCII code.

Any characters left unread in the buffer will be received by the next `TERMINC`.

### $TERMOUTC

Requires 3 word control structure, as follows

0: the value `$TERMOUTC`

1: the number of characters to be printed

2: (memory address) the address at which the characters may be found.

Error codes are the same as for `$TERMINC`



Successful result (returned in register):

the number of characters actually printed

Notes:

- The cha[illegible] should be in the form of a proper string (packed four per [illegible]given memory location. The string does not *need* to be zero[illegible]
- If the nu[illegible]s specified to be zero, the string will be assumed to be zero[illegible]nlimited number of characters will be printed.
- If the nu[illegible]is specified to be non-zero, that number of characters will be printed, even if they include some zeros.
- If the number of characters is specified to be one, then the memory location may just contain the character's ASCII code; no extra formatting is required to make it into a string.



### $TERMINW and $TERMOUTW



These operations perform exactly as `TERMINC` and `TERMOUTC` with the following exceptions:

- The data is not formatted as a string. Reading or printing N characters requires exactly N words of memory, containing one ASCII code each.
- The input operation will *not* zero-terminate the array of characters.



## Network Operations



The emulator uses UDP to simulate an IP network. Simulated IP addresses are six bytes long, consisting of the real IP address of the computer and a two byte UDP port number that may be requested by the programmer. Packets of data consist of any number of bytes up to 1024. Network devices must be started before they can be used, and should be stopped when they are no longer needed.

### $NETSS

Start or stop an network device.

Requires 4 word control structure, as follows

0: the value `$NETSS`

1: the unit number. Currently up to two network interfaces are supported, numbered 1 and 2.

2: the value 0 to turn a device off, or 1 to turn it on.

3: A pointer to two words of memory. The first word should be zero, and the second should be the requested real UDP port number to use, or zero to let the system select an unused port.
When the call terminates, these two words will contain the six byte simulated IP address.

Error codes:



| | | |
|---|---|---|
| `-2,` | `ERR_READ_PARAMS:` | memory access reading or writing the control structure. |
| `-5,` | `ERR_MEMORY:` | error while accessing the two-word IP address.. |
| `-3,` | `ERR_DE`[illegible] | [illegible] < 1 or > 2, or when closing a device, unit not in use. |
| `-9,` | `ERR_IN`[illegible] | [illegible] starting a device, unit already in use. |

<u>$NETSEND</u>

Send a packet [illegible]
Requires 4 wo[illegible] as follows

0: the valu[illegible]
1: the unit number, 1 or 2.
2: A pointer to two words of memory, containing the destination IP address.
3: The number of bytes to be sent.
4: (memory address) the address at which those bytes may be found.



Error codes:



| | | |
|---|---|---|
| `-2,` | `ERR_READ_PARAMS:` | memory access reading or writing the control structure |
| `-5,` | `ERR_MEMORY:` | error accessing IP address or data to be sent. |
| `-3,` | `ERR_DEV_NUMBER:` | unit < 1 or > 2, or when closing a device, unit not in use. |
| `-8,` | `ERR_BAD_PARAM:` | number of bytes < 0 or > 1024 |
| `-6,` | `ERR_DEV_FAILED:` | the underlying unix call used to simulate IP transmission failed. |



<u>$NETRECV</u>



Receive a packet of data.
This is a non-blocking operation. If no packet has been received yet, it will immediately return a code of -1
Requires 4 word control structure, as follows



0: the value `$NETRECV`
1: the unit number, 1 or 2.
2: A pointer to two words of memory, which will be set to contain the source IP address of the packet received.
3: (memory address) the address at which the bytes received should be stored.

Error codes:

| | | |
|---|---|---|
| `-2,` | `ERR_READ_PARAMS:` | memory access reading or writing the control structure. |
| `-5,` | `ERR_MEMORY:` | error accessing IP address or the read-data buffer. |
| `-3,` | `ERR_DEV_NUMBER:` | unit < 1 or > 2, or when closing a device, unit not in use. |
| `-6,` | `ERR_DEV_FAILED:` | the underlying unix call used to simulate IP transmission failed. |
| `-11,` | `ERR_NO_DATA:` | (not an error) no data has bee received yet, just try again later. |

## Time Operations



### $SECONDS

Requires 1 wo[illegible] as follows
0: the valu[illegible]

Error codes:
`-2, ERR_RE`[illegible]ory access reading or writing the control structure.

Successful res[illegible]ter):
The num[illegible]ed since midnight (0000 hours) on 1st January 2000.

### $USECONDS



Requires 3 word control structure, as follows
0: the value `$SECONDS`
1: output only: receives the number of seconds
2: output only: receives the number of microseconds



Error codes:
`-2, ERR_READ_PARAMS:` memory access reading or writing the control structure.



Successful result (returned in the control structure):
The number of microseconds elapsed since midnight (0000 hours) on 1st January 2000, split into separate integers for seconds and microseconds.



### $DATETIME



Splits a date/time value into its human-oriented parts.

Requires 9 word control structure, as follows
0: the value `$DATETIME`
1: a time value of the kind returned by `$SECONDS`
2: output only: receives the year
3: output only: receives the month, 1 to 12
4: output only: receives the day of the month, 1 to 31
5: output only: receives the day of the week, 0 to 6, 0 = Sunday
6: output only: receives the hour, 0 to 23
7: output only: receives the minute, 0 to 59
8: output only: receives the second, 0 to 59

Error codes:
`-2, ERR_READ_PARAMS:` memory access reading or writing the control structure.

## Virtual Memory

Because the emulator uses 32 bit words instead of 8 bit bytes, the Intel scheme of splitting a virtual address into a 10 bit page table number, a 10 bit page number, and a 12 bit offset can not be used exactly.

A 12 bit offset m[illegible]ld be 4096 memory locations in a page, and that would mean that a page tabl[illegible]resses of 4096 pages instead of 1024, so we would not need so many of then[illegible]

In the emulator a[illegible]nsists of 2048 32-bit locations requiring only an 11 bit offset. That means th[illegible]old the addresses of 2048 pages, so 11 bits are required for page numbers. T[illegible] bits for the page table number, meaning that page directories only fill h[illegible] page.

A Virtual Address

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page Table Number | | | | | | | | | | Page Number | | | | | | | | | | | Offset | | | | | | | | | | |



There are two advantages to this changed layout: pages are smaller, so more of them are available without using up so much real memory, and page directories only fill half a page, so it is quite possible that you can store everything you need to know about a process in one single page.

Only the most significant 22 bits of the value stored in the Page Directory Base Register are looked at during virtual address translation. Page directories must occupy complete half-pages; their addresses must be multiples of 1024 (i.e. in binary they must end in 10 zeros).

Only the most significant 21 bits of the values stored in the Page Directories are looked at during virtual address translation. Page tables must occupy whole pages; their addresses must be multiples of 2048 (i.e. in binary they must end in 11 zeros).

The entries in page tables include two page status bits in the least significant bits. They are the Resident or Valid bit (in bit 0) and the System bit (in bit 1). The meaning of a page table entry depends upon the value of the Resident bit.

A Page Table Entry, which is the same as a Page Directory Entry.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Physical Page Address | | | | | | | | | | | | | | | | | | | | | Unassigned | | | | | | | | | S | R |

If the Resident bit is Zero, any access to this virtual page will immediately cause a `PAGEFAULT` interrupt, and the other 31 bits will not even be seen. They may be used for any purpose whatsoever.

If the System bit is 1, any attempted access to this virtual page while in User mode will result in a `PAGEPRIV` interrupt, and the access will not occur.

In all cases, bits 2 to 10 have no assigned meaning, and may be used for any purpose whatsoever.

## MEMORY ACCESS ALGORITHM

```
let A be the address in memory referenced by an instruction.
if $VM flag is OFF:
    Use physical memory at address A
otherwise, if $VM flag is ON:
    // A is a virtua[...] translated.
    let DIR be (A[...]          // most significant 10 bits
    let PG be (A>[...]          // next 11 bits
    let OFFS be A[...]          // least significant 11 bits
    let POS be DI[...]BR register
    read PTADDR [...]y address POS
    if PTADDR is [...]
        PAGEFAULT, translation abandoned
    PTADDR &= 0xFFFFF800        // zero out least significant 11 bits
    read PGADDR from physical memory address (PG + PTADDR)
    let R be PGADDR & 1         // least significant bit
    if R is Zero:
        PAGEFAULT, translation abandoned
    let S be (PGADDR & 2)>>1    // second least significant bit
    if S is One and $SYS flag is Zero:
        PAGEPRIV, translation abandoned
    PGADDR &= 0xFFFFF800        // zero out least significant 11 bits
    let PHYS be PGADDR + OFFS
    Use physical memory at address PHYS
```

## Privileged Operations

If any of the following instructions are executed when the $SYS flag is off (zero), a PRIVOP interrupt will be triggered and the operation will not be performed.

SETSR, PERI, IRET, PHLOAD, PHSTORE, FLAGSJ, CLRPP

If a HALT instruction is executed when the $SYS flag is off (zero), a HALT interrupt will be triggered and the processor will *not* be halted.

If a SETFL instruction is executed when the $SYS flag is off (zero), and it attempts to modify either the R, SYS, VM, or IP flag, a PRIVOP interrupt will be triggered and the operation will not be performed.